TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Methods and Algorithms for Efficient Programming of FPGA-based Heterogeneous Systems for Object Detection

Lester Kalms
Born on: 22nd December 1985 in Hannover

Dissertation

to achieve the academic degree

Doktor-Ingenieur (Dr. Ing.)

Supervisor and Examiner
Prof. Dr.-Ing. Diana Göhringer
Co-Examiner
Prof. Dr. Marco D. Santambrogio

Submitted on: 7th April 2022
Defended on: 28th June 2022

TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Statement of authorship

I hereby certify that I have authored this document entitled *Methods and Algorithms for Efficient Programming of FPGA-based Heterogeneous Systems for Object Detection* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Diana Göhringer

Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 7th April 2022

Lester Kalms

TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Abstract

Nowadays, there is a high demand for computer vision applications in numerous application areas, such as autonomous driving or unmanned aerial vehicles. However, the application areas and scenarios are becoming increasingly complex, and their data requirements are growing. To meet these requirements, it needs increasingly powerful computing systems. FPGA-based heterogeneous systems offer an excellent solution in terms of energy efficiency, flexibility, and performance, especially in the field of computer vision. Due to complex applications and the use of FPGAs in combination with other architectures, efficient programming is becoming increasingly difficult. Thus, developers need a comprehensive framework with efficient automation, good usability, reasonable abstraction, and seamless integration of tools. It should provide an easy entry point, and reduce the effort to learn new concepts, programming languages and tools. Additionally, it needs optimized libraries for the user to focus on developing applications without getting involved with the underlying details. These should be well integrated, easy to use, and cover a wide range of possible use cases. The framework needs efficient algorithms to execute applications on heterogeneous architectures with maximum performance. These algorithms should distribute applications across various nodes with low fragmentation and communication overhead and find a near-optimal solution in a reasonable amount of time. This thesis addresses the research problem of an efficient implementation of object detection applications, their distribution across FPGA-based heterogeneous systems, and methods for automation and integration using toolchains. Within this, the three contributions are the `HiFlipVX` object detection library, the `DECISION` framework, and the `APARMAP` application distribution algorithm.

`HiFlipVX` is an open-source HLS-based FPGA library optimized for performance and resource efficiency. It contains 66 highly parameterizable computer vision functions including neural networks, ideally for design space exploration. It extends the OpenVX standard for feature extraction, which is challenging due to unknown element size at design time. All functions are streaming capable to achieve maximum performance by increasing parallelism and reducing off-chip memory access. It does not require external or vendor libraries, which eases project integration, device coverage, and vendor portability, as shown for Intel. The library consumed on average 0.39 % FFs and 0.32 % LUTs for a set of image processing functions compared to a vendor library. A `HiFlipVX` implementation of the AKAZE feature detector computes between 3.56 and 4.13 times more pixels per second than the related work, while its resource consumption is comparable to optimized VHDL designs. Its neural network extension achieved a speedup of 3.23 for an AlexNet layer in comparison to a related work, while consuming 73 % less on-chip memory. Furthermore, this thesis proposes an improved feature extraction implementation that achieves a repeatability of 72.57 % when weighting complex cases, while the next best algorithm only achieves 62.99 %.

`DECISION` is a framework consisting of two toolchains for the efficient programming of FPGA-based heterogeneous systems. Both integrate `HiFlipVX` and use a joint OpenVX-based frontend to implement computer vision applications. It abstracts the underlying hardware and algorithm details while covering a wide range of architectures and applications. The first toolchain targets x86-based systems consisting of CPUs, GPUs, and FPGAs using OpenCL (Open Computing Language). To create a heterogeneous schedule, it considers device profiles, kernel profiles and estimates, and FPGA dataflow characteristics. It manages synchronization, memory transfers and data coherence at design time. It creates a runtime optimized program which excels by its high parallelism and a low overhead. Additionally, this thesis looks at the integration of OpenCL-based libraries, automatic OpenCL kernel generation, and OpenCL kernel optimization and comparison for different architectures. The second toolchain creates an application specific and adaptive NoC-based architecture. The streaming-optimized architecture enables the reusability of vision functions by multiple applications to improve the resource efficiency while maintaining high performance. For a set of example applications, the resource consumption was more than halved, while its overhead was only 0.015 % in terms of performance.

`APARMAP` is an application distribution algorithm for partition-based and mesh-like FPGA topologies. It uses a NoC (Network-on-Chip) as communication infrastructure to connect reconfigurable regions and generate an application-specific hardware architecture. The algorithm uses load balancing techniques to find reasonable solutions within a predictable and scalable amount of time. It optimizes solutions using various heuristics, such as Simulated Annealing and Tabu Search. It uses a multithreaded grid-based approach to prevent threads from calculating the same solution and getting stuck in local minimums. Its constraints and objectives are the FPGA resource utilization, NoC bandwidth consumption, NoC hop count, and execution time of the proposed algorithm. The evaluation showed that the algorithm can deal with heterogeneous and irregular host graph topologies. The algorithm showed a good scalability in terms of computation time for an increasing number of nodes and partitions. It was able to achieve an optimal placement for a set of example graphs up to a size of 196 nodes on host graphs of up to 49 partitions. For a real application with 271 nodes and 441 edges, it was able to achieve a distribution with low resource fragmentation in an average time of 149 ms.

Acknowledgements

First, I would like to express my deepest appreciation to my supervisor Prof. Dr. Diana Göhringer for giving me the opportunity to do this dissertation and for the good support throughout the whole time. I could not have undertaken this journey without your support, advice and great supervision. Special thanks goes to my second supervisor, Prof. Dr. Marco D. Santambrogio, for the support and review of my dissertation. Many thanks to my Fachreferent Prof. Dr. Jerónimo Castrillón-Mazo and your advice. Many thanks also to the committee members Prof. Dr. Akash Kumar and Prof. Dr. Martin Wollschlaeger.

I would like to extend my sincere thanks to the entire MCA team at Ruhr-Universität Bochum and the ADS team at Technische Universität Dresden. I have enjoyed working with each one of you and am grateful for the fruitful discussions, collaborations and insights. There are too many to name all. Yet I express my deep gratitude to Dr. Muhammed Al Kadi, Dr. Philipp Wehner, Dr. Jens Rettkowski and Dr. Salma Hesham, with whom I was able to come up with many questions, especially at the beginning of this dissertation. I had the pleasure of supervising many students from whom I could also learn a lot. Some of these works are mentioned at the end of this dissertation.

Of course, my biggest thanks go to my family, Camille, Dietmar and Nelson, without whom I would never have come this far in life. I am so happy for all their love. I also want to thank my closest friends who have been with me on my journey since childhood: Felix, Pascal, Johann, Maximilian, Nicolas, Lukas and Malte.

Contents

List of Figures IV

List of Tables VII

List of Listings IX

1 Introduction 1
- 1.1 Motivation and Problem Statement . . . 1
- 1.2 Own Contribution . . . 2
- 1.3 Thesis Outline . . . 4

2 Background 5
- 2.1 Object Detection . . . 6
  - 2.1.1 Pattern Recognition . . . 8
  - 2.1.2 Feature Detection, Description and Matching . . . 8
  - 2.1.3 Multiscale Feature Detection and Description Algorithms . . . 10
  - 2.1.4 ORB . . . 11
  - 2.1.5 (A)KAZE . . . 12
  - 2.1.6 FREAK . . . 14
  - 2.1.7 FPGA Implementations . . . 15
- 2.2 Heterogeneous Systems . . . 17
  - 2.2.1 FPGA-based Systems . . . 18
  - 2.2.2 Communication and Topology . . . 19
  - 2.2.3 FPGA-based Clusters . . . 21
  - 2.2.4 Summary . . . 23
- 2.3 Application Distribution . . . 25
  - 2.3.1 Metaheuristics . . . 25
  - 2.3.2 Mapping and Scheduling on Many-Core Systems . . . 29
  - 2.3.3 Design Space Exploration for Neural Networks on FPGA-based Cluster 31
  - 2.3.4 Clustering and Placement for Configurable Logic Blocks . . . 32
  - 2.3.5 Tuning, Clustering and Placement for Partial Reconfigurable Regions . 34
  - 2.3.6 Summary . . . 37
- 2.4 Programming Methods . . . 40
  - 2.4.1 General Purpose Approaches . . . 41
  - 2.4.2 Domain Specific Approaches . . . 48
- 2.5 Toolchains . . . 53
  - 2.5.1 Source-to-Source Compilers (Transpilers) . . . 54

2.5.2 High-Level Synthesis (HLS) Tools 59
2.5.3 FPGA-based OpenVX Tools 63
2.5.4 OpenCL-based Tools 66

3 `HiFlipVX`: Object Detection Library 69
3.1 `HiFlipVX` Library Functions 70
3.1.1 Image Pixelwise Functions 70
3.1.2 Image Filter Functions 72
3.1.3 Image Conversion Functions 82
3.1.4 Image Analysis Functions 83
3.1.5 Image Function Latency & HLS Directive Usage 87
3.1.6 Feature Functions 89
3.1.7 Neural Network Functions 95
3.2 Object Detection Algorithms 104
3.2.1 Proposed AKAZE and FREAK based Feature Extraction Algorithm . . . 104
3.2.2 FAST Corner Detector 107
3.2.3 Canny Edge Detector 108
3.2.4 ORB feature detection 109
3.2.5 AKAZE feature detection 113
3.2.6 FREAK feature description 118
3.2.7 MobileNets 123
3.3 Evaluation 125
3.3.1 Library Functions 125
3.3.2 Feature Extraction Algorithms 137
3.3.3 Neural Network Extension 149
3.4 Summary 156

4 `DECISION`: Vision Framework 159
4.1 Overview 159
4.2 OpenVX Graph Creation Module 161
4.2.1 OpenVX Application Flow 162
4.2.2 Data Extraction 164
4.3 Architecture Dependent OpenCL Kernel Optimizations 168
4.3.1 FPGA Bandwidth and Kernel Optimization 169
4.3.2 FPGA Example Implementation 170
4.3.3 CPU and (integrated) GPU Kernel Optimization 171
4.4 Automatic OpenCL Code Generation 172
4.4.1 Overview 173
4.4.2 PPCG Source Code Generation 174
4.4.3 Creating OpenCL Device and Host Code 175
4.4.4 Conversion from LLVM-IR to C-Code 176
4.4.5 Converting PHI Instructions 176
4.5 High-Performance Vision Toolchain 177
4.5.1 Overview 177
4.5.2 Library Module 183
4.5.3 Profiling Module 184
4.5.4 Mapping and Scheduling Module 186
4.5.5 Program Creation Module 188
4.5.6 Runtime System Module 192

4.6 Embedded System Vision Toolchain . . . 197
4.6.1 Module Overview . . . 198
4.6.2 Model Description . . . 200
4.6.3 Library Module . . . 203
4.6.4 Hardware Creation Module . . . 207
4.7 Evaluation . . . 208
4.7.1 Architecture Dependent OpenCL Kernel Optimizations . . . 209
4.7.2 Automatic OpenCL Code Generation . . . 212
4.7.3 High-Performance Vision Toolchain . . . 214
4.7.4 Embedded System Vision Toolchain . . . 219
4.8 Summary . . . 223

5 APARMAP: Application Distribution Algorithm 227
5.1 Overview . . . 228
5.2 Heuristics and Concepts . . . 229
5.2.1 Objective Function . . . 230
5.2.2 Local Search Function . . . 232
5.2.3 Grid-Based Solution Space . . . 232
5.2.4 Simulated Annealing . . . 233
5.2.5 History . . . 233
5.2.6 Tabu Search . . . 234
5.3 Algorithm . . . 236
5.3.1 Scheduling & Mapping Phase . . . 236
5.3.2 Load Balancing Phase . . . 237
5.3.3 Optimization Phase . . . 241
5.4 Evaluation . . . 242
5.4.1 Default Parameters . . . 243
5.4.2 Measurement Methodology . . . 244
5.4.3 Memory Usage . . . 244
5.4.4 Load Balancing & Optimization Phase Comparison . . . 244
5.4.5 Amount of Threads . . . 246
5.4.6 Load Balancing Phase Iterations . . . 247
5.4.7 Load Balancing Complexity . . . 248
5.4.8 Optimization Phase: MTM and STM amount per LTM . . . 249
5.4.9 Irregular and Heterogeneous Topologies . . . 249
5.4.10 Verifying Optimality and Scalability . . . 250
5.4.11 Real Application . . . 251
5.4.12 Comparison to Previous Work . . . 251
5.5 Summary . . . 252

6 Conclusion and Outlook 255
6.1 Summary of Contributions . . . 255
6.2 Future Work . . . 257

Bibliography 259

Student Work 281

A Appendix 283
A.1 High-Performance Vision toolchain: Transaction Patterns Between Kernels . 283

List of Figures

1.1 `DECISION` framework including the `HiFlipVX` library and the `APARMAP` algorithm. 2

2.1 Three different object detection techniques in the computer vision field. . . . 7
2.2 Feature extraction properties: feature types, scale-space & orientation 9
2.3 Bresenham circle used in the FAST detector [17]. 12
2.4 Comparison between the Gaussian and nonlinear diffusion scale-space [46]. 13
2.5 Illustration of the FREAK sampling pattern [24]. 15
2.6 An MPSoC using a NoC as communication infrastructure. 18
2.7 FPGA cluster with inter-FPGA connection using a direct or indirect network. . 20
2.8 A combined computing model for embedded and HPC. 24
2.9 Classification of metaheuristics including the most known examples. 27
2.10 A taxonomy of mapping methodologies [107]. 29
2.11 The flow of FPGA-based CAD tools. 33
2.12 Placement example of the DyNoC [142] architecture. 36
2.13 Subareas for the application distribution process on FPGA-based systems. . 38
2.14 The OpenCL platform model [153]. 43
2.15 The OpenCL memory model [153]. 44
2.16 The OpenMP fork-join model. 45
2.17 Available implementations of the SYCL specification [165]. 47
2.18 The OpenCL acceleration of OpenCV, adapted from [179]. 50
2.19 The four main steps of the polyhedral optimization. 56
2.20 The structure of the Polly tool. 58
2.21 The structure of the PPCG tool. 59
2.22 Classification of HLS tools based on the input language [222]. 62
2.23 Overview of the layers in the OpenVX framework [229]. 64

3.1 Sliding window approach for a 3 × 3 kernel. 73
3.2 Common filter structure. 75
3.3 Separable filter structure. 77
3.4 Fields of a `HiFlipVX` feature data type. 90
3.5 Input stages of the 3D-convolution function. 98
3.6 Computation stages of the 3D-convolution function. 99
3.7 Canny edge and FAST corner detector implementations. 108
3.8 Hardware design of the ORB detector. 109
3.9 Hardware design of the Retain Best block. 111
3.10 Integration of the ORB detector into the video processing system. 112
3.11 Overview of the ORB feature detection implementation. 113

3.12 Block diagram of the complete system for the AKAZE VHDL implementation. 114
3.13 Block diagram of a 5 × 5 window generator. 115
3.14 Block diagram of the AKAZE contrast factor and nonlinear scale-space creation.115
3.15 Block diagram of the AKAZE feature detection. 117
3.16 HLS implementation of the optimized AKAZE feature detector. 117
3.17 Integral Image top-level (left) and parallel prefix-sum of 4 pixel (right). 119
3.18 Top-level hardware design of the FREAK algorithm. 120
3.19 HW design of Boundary block, unrotated Pattern block and Intensity block. . 121
3.20 Design of the Orientation (top) and rotated Pattern block (bottom). 122
3.21 Block design of the MobileNets hardware implementation. 124
3.22 Compares relative LUT and FF utilization of pixelwise functions. 128
3.23 Resource utilization scalability results of filter functions. 131
3.24 LUT and FF utilization of conversion functions. 132
3.25 Resource utilization of analysis functions. 133
3.26 Resource utilization of feature functions. 135
3.27 Relative FF/LUT and absolute BRAM/DSP utilization compared to xfOpenCV. . 137
3.28 Repeatability comparison of the embedded AKAZE design. 144
3.29 Repeatability comparison of the `HiFlipVX` AKAZE design. 145
3.30 Relative resource utilization for neural network layers. 153

4.1 Overview and flow of the `DECISION` framework. 160
4.2 Small application graph illustrating the buffering problem. 166
4.3 Task graph of the AKAZE feature detection implementation. 168
4.4 Windowed (3 × 3) function implementation on an FPGA. 170
4.5 Example for the input pixels of a register buffer for an 8 × 4 grid. 171
4.6 Overview of the combined Polly and PPCG tool design. 173
4.7 LLVM to OpenCL: Modifications of Polly and PPCG. 174
4.8 Overview of the proposed High-Performance Vision toolchain. 178
4.9 Data movement and device, queue, and node types for memory management 180
4.10 Dataflow pipelining latency effects, adapted from [214]. 186
4.11 Example system to cover all possible types of data transfer. 191
4.12 Multithreaded OpenCL runtime system to execute OpenVX graph applications.195
4.13 All operations processed during the execution of a message. 197
4.14 Overview and flow of the Embedded System Vision toolchain. 199
4.15 Application distribution model for mesh-like partition-based architectures. . 200
4.16 Example platform model topologies. 201
4.17 Network interface and DMA controller example. 203
4.18 Control flit structure for adaptable network interfaces. 204
4.19 Computation time for different devices for a resolution of 1280 × 720. 211
4.20 Matrix multiplication speedup compared to a single-threaded CPU execution. 215
4.21 Median kernel pipelined and vectorized execution time for different devices. 218
4.22 Embedded vision example architecture. 221

5.1 Overview of the application distribution process. 227
5.2 Activation function (left) and bandwidth consumption (right). 231
5.3 Structure of the LTM in the history. 234
5.4 Plot of the node graph after load balancing. 238
5.5 Shows which steps of load balancing are executed in which iteration periods. 238
5.6 Computation for the initial node coordinates of a guest graph. 239

5.7 Optimization algorithm for the clustering and placement process. 242
5.8 Example graphs with 24, 36 and 48 nodes. 245
5.9 Load balancing and optimization phase comparison. 246
5.10 Results for different thread amount. 247
5.11 Results for different number of load balancing iterations. 247
5.12 Collision detection in the load balancing phase. 248
5.13 Results for different number of MTM and STM amount per LTM. 249
5.14 Results for irregular and heterogeneous topologies. 250
5.15 Verification of scalability and optimality. 250

A.1 Example system to cover all possible types of data transfers. 283
A.2 All possible data transfers with a host or IACC as destination 285
A.3 All possible data transfers with a DACC as destination 286

List of Tables

2.1 Different FPGA implementations for feature detection and description. 16
2.2 Two approaches for direct GPU-FPGA communication over PCI-e. 21
2.3 Comparison of five FPGA-based cluster. 22
2.4 Comparison of application distribution algorithms targeting FPGAs. 39
2.5 Existing C/C++ based parallel programming approaches. 41
2.6 Comparison of directive-based languages. 45
2.7 Comparison of SYCL with OpenCL and OpenMP [164]. 47
2.8 OpenVX or similar libraries with their conformance version and acceleration. 50
2.9 Comparison of FPGA-based computer vision domain specific libraries. 52
2.10 Comparison of FPGA-based OpenVX frameworks. 66
2.11 Comparison of OpenCL-based toolchains. 68

3.1 `HiFlipVX` image pixelwise functions. 71
3.2 `HiFlipVX` image filter functions. 73
3.3 Constants (uppercase) and variables (lowercase) used in `HiFlipVX`. 74
3.4 `HiFlipVX` image conversion functions. 82
3.5 `HiFlipVX` image analysis functions. 84
3.6 `HiFlipVX` feature functions. 90
3.7 `HiFlipVX` neural network functions. 96
3.8 Implemented activation functions. 102
3.9 FPGA development boards and their available resources. 126
3.10 Default configuration of the `HiFlipVX` library functions. 126
3.11 Synthesis and implementation results of pixelwise functions. 127
3.12 Synthesis and implementation results of filter functions. 129
3.13 Comparison of different combinations of feature detectors and descriptors. . 138
3.14 Resource utilization of the FREAK and integral image hardware implementation.140
3.15 Comparison of the FREAK with the related work. 141
3.16 Proposed AKAZE hardware designs. 142
3.17 Resource consumption of the proposed AKAZE implementation. 144
3.18 Best configuration for ORB detector + FREAK descriptor. 146
3.19 Comparison of ORB execution time between software and hardware. 147
3.20 Resource utilization of the proposed ORB designs. 147
3.21 Comparison of proposed ORB implementation to related work. 148
3.22 Default configuration of the library compile time parameters. 149
3.23 The MAPE (Mean Absolute Percentage Error) of the neural network extension. 150
3.24 Resource utilization & latency of synthesized & implemented design per layer. 151
3.25 The MAPE and resource utilization of the activation functions. 152

3.26 Vectorization tuning for the proposed MobileNets modules. 154
3.27 Proposed MobileNets modules executed separately on the ZCU104. 155
3.28 Comparison of SSD-MobileNets-V1 with the related work. 155

4.1 Actions and protocol for data transfer for memory management. 188
4.2 Test system components. 209
4.3 Resource usage and memory bandwidth for 1 read and 1 write DMA kernel. . 209
4.4 Compute time [ms] for different memory port widths of a 5 × 5 Gaussian kernel.210
4.5 Comparison of the implemented algorithm for different devices and resolutions.211
4.6 Devices used in the different measurements. 216
4.7 Device profiling results calculated with Equation (4.7). 216
4.8 Comparison of devices and kernels without data transfer in [µs]. 217
4.9 OpenCL synchronization and kernel offloading overhead. 218
4.10 Resource utilization of the NoC-based architecture for the example application.222
4.11 Resource utilization of the example application as pure AC design. 223

5.1 Variables of the application distribution algorithm. 229
5.2 Constants of the application distribution algorithm. 229
5.3 Memory usage of application distribution algorithm. 245

A.1 Distance calculation for data transfer between two devices 284

List of Listings

2.1 Simple matrix multiplication code including two statements (S0 and S1). 56

3.1 Integer square root function. Each stage computes 1 bit of the resulting vector. 72
3.2 Generic data type used for vectorization and to data types wider than 64 bit. 87
3.3 Example application showing how to connect two `HiFlipVX` functions. 88
3.4 General structure of the 3D-convolution function. 97
3.5 Pseudocode of the proposed AKAZE-FREAK algorithm. 106

4.1 OpenVX example code for an edge detector. 162
4.2 Instantiation of a `HiFlipVX`-based accelerator. 166
4.3 Program flow of Program Creation module to create all queue commands . . 189
4.4 Program flow of the Runtime System module. 193
4.5 Example C++ matrix multiplication function. 213
4.6 Generated OpenCL matrix multiplication kernel. 214

5.1 Main structure of the TS (Tabu Search) algorithm. 235

Acronyms

AC Accelerator

ACO Ant Colony Optimization

AES Advanced Encryption Standard

AGAST Adaptive Generic Accelerated Segment Test

AKAZE Accelerated KAZE

ALAP As-Late-As-Possible

AMBA Advanced Microcontroller Bus Architecture

API Application Programming Interface

ASAP As-Soon-As-Possible

ASIC Application-Specific Integrated Circuit

AST Abstract Syntax Tree

BRAM Block Random-Access Memory

BRIEF Binary Robust Independent Elementary Features

BRISK Binary Robust Invariant Scalable Keypoints

CAD Computer-Aided Design

CLB Configurable Logic Block

CLooG Chunky Loop Generator

CMP Common MidPoint

CNN Convolutional Neural Network

CORDIC COordinate Rotation DIgital Computer

CPU Central Processing Unit

CSA Carry-Save Adder

CU Compute Unit

CUDA Compute Unified Device Architecture

DACC Dedicated Accelerator

DAG Directed Acyclic Graph
DDR Double Data Rate
DFG Data Flow Graph
DLP Data-Level Parallelism
DMA Direct Memory Access
DMC Depth Map Computation
DoG Difference of the Gaussian
DoH Determinant of the Hessian
DPR Dynamic Partial Reconfiguration
DSE Design Space Exploration
DSL Domain-Specific Language
DSP Digital Signal Processor
DVFS Dynamic Voltage and Frequency Scaling
EOF End of Frame
EoL End of Line
EoM End of Message
FAST Features from Accelerated Segment Test
FED Fast Explicit Diffusion
FF Flip-Flop
FIFO First-in-First-out
flit flow control unit
FPGA Field-Programmable Gate Array
fps frames per second
FREAK Fast Retina Keypoint
FSM Finite-State Machine
GA Genetic Algorithm
GCC GNU C Compiler
GPL General-Purpose Language
GPU Graphics Processing Unit
HDL Hardware Description Language
HDMI High Definition Multimedia Interface
HEFT Heterogeneous Earliest Finish Time
HLS High-Level Synthesis

HPC High-Performance Computing

I/O Input/Output

IACC Integrated Accelerator

ICAP Internal Configuration Access Port

ICD Installable Client Driver

IFM Input Feature Map

ILP Instruction-Level Parallelism

IP Intellectual Property

IR Intermediate Representation

ISL Integer Set Library

k-NN k-Nearest Neighbors

LCM Least Common Multiple

LLVM Low Level Virtual Machine

LooPo Loop Parallelizer

LTM Long-Term Memory

LUT Lookup Table

M-LDB Modified-Local Difference Binary

M-SURF Modified SURF

MA Manager

MAPE Mean Absolute Percentage Error

MMU Memory Management Unit

MPI Message Passing Interface

MPSoC Multiprocessor System-on-Chip

MSI Modified Shared Invalid

MTL Matrix Template Library

MTM Medium-Term Memory

MWD Multi Window Display

NI Network Interface

NMS Non-Maximum Suppression

NoC Network-on-Chip

NUMA Non-Uniform Memory Access

ODE Ordinary Differential Equation

oFAST oriented FAST

OFM Output Feature Map

OpenACC Open Acceleration

OpenCL Open Computing Language

OpenGL Open Graphics Language

OpenMP Open Multi-Processing

ORB Oriented FAST and Rotated BRIEF

OS Operating System

PC Personal Computer

PCI-e Peripheral Component Interconnect Express

PDE Partial Differential Equation

PE Processing Element

PET Polyhedral Extraction Tool

PIPLib Parametric Integer linear Programming

PLUTO Polyhedral Parallelizer and Locality Optimizer

PoCC Polyhedral Compiler Collection

PPCG Polyhedral Parallel Code Generator

PPS Pixels Per Second

PRR Partial Reconfigurable Region

PSO Particle Swarm Optimization

PU Processing Unit

QAP Quadratic Assignment Problem

rBRIEF rotated BRIEF

RISC Reduced Instruction Set Computer

RSA RivestShamirAdleman

RTL Register Transfer Level

SA Simulated Annealing

SCEV SCalar EVolution

SCoP Static Control Part

SDK Software Development Kit

SIFT Scale Invariant Feature Transform

SIMD Single Instruction Multiple Data

SLAM Simultaneous Localization and Mapping

SoA State-of-the-Art

SoC System-on-Chip

SoF Start of Frame

SPIR Standard Portable Intermediate Representation

SR Subpixel Refinement

SSA Single Static Assignment

STL Standard Template Library

STM Short-Term Memory

SURF Speeded Up Robust Features

SVM Shared Virtual Memory

SVP Soft Vector Processor

TCL Tool command language

TDP Thermal Design Power

TLP Task-Level Parallelism

TOPS Terra Operations per Second

TS Tabu Search

UAV Unmanned Aerial Vehicle

UMA Uniform Memory Access

URAM Ultra Random-Access Memory

VCI Vector Custom Instruction

VHDL Very High-Speed Integrated Circuit Hardware Description Language

VLSI Very Large-Scale Integration

VOPD Video Object Plane Decoder

VPS Video Processing System

VPU Vision Processing Unit

WCET Worst-Case Execution Time

XML Extensible Markup Language

1 Introduction

1.1 Motivation and Problem Statement

Many application areas, such as autonomous driving, hand gesture recognition, medical x-ray imaging, advanced driver assistance or UAVs (Unmanned Aerial Vehicles), require object detection [1, 2, 3, 4, 5], which is a subfield in computer vision and image processing. However, the applied application areas and scenarios are becoming more and more complex. In addition, application requirements are increasing due to higher image resolutions, frame rates, color depths, and a larger number of input images for 2D and 3D vision.

To meet these demands, the corresponding algorithms require increasingly powerful computing systems. At the same time, energy consumption plays an important role in the present time [6]. This is true for small embedded systems as well as for large computing clusters. To counteract this, FPGA (Field-Programmable Gate Array)-based heterogeneous systems provide an excellent solution for many use cases in terms of energy efficiency, flexibility, and performance [7, 8]. Especially in the field of computer vision, these systems can show their strengths and benefits compared to CPU (Central Processing Unit)- and GPU (Graphics Processing Unit)-based acceleration [9, 10]. For example, through their streaming capability, which enables very large parallelism with little external memory access through deep pipelines.

Due to the complex algorithms and the use of FPGAs in combination with other architectures, efficient programming is becoming increasingly challenging. Thus, developers need a comprehensive framework with efficient automation, good usability, reasonable abstraction, and seamless integration of different tools and libraries. It should not only provide an easy entry point for the user, but also reduce the effort required to learn new concepts, programming languages and tools. Additionally, the framework should use optimized libraries and DSLs (Domain-Specific Languages) so that the user can focus on developing new applications without getting too involved with the underlying details. These should be well integrated and easy to use, but at the same time complex enough to cover as many use cases as possible. Furthermore, it should be possible to optimize an application based on various objectives such as performance or resource utilization.

The framework needs efficient algorithms to execute applications on heterogeneous architectures with maximum performance. These algorithms should be able to distribute applications across various heterogeneous nodes with low fragmentation and communication overhead. Even though such problems are complex and NP-hard [11], the algorithm should find a near-optimal distribution in a reasonable amount of time. The specified objectives can be diverse, such as high performance, energy efficiency, or resource utilization. Through the design time optimizations and flexibility of FPGAs, it should even be possible to generate an application-specific architecture, based on a set of applications.

1.2 Own Contribution

This thesis addresses the problem of efficient programming of object detection algorithms on FPGA-based heterogeneous systems using various methods and algorithms. It identified three main areas within this topic as particularly important and explored them. This includes the efficient implementation of object detection applications, their distribution across heterogeneous architectures, and methods for automation and integration using toolchains. This thesis addresses them in the following three contributions, which are also shown in Figure 1.1:

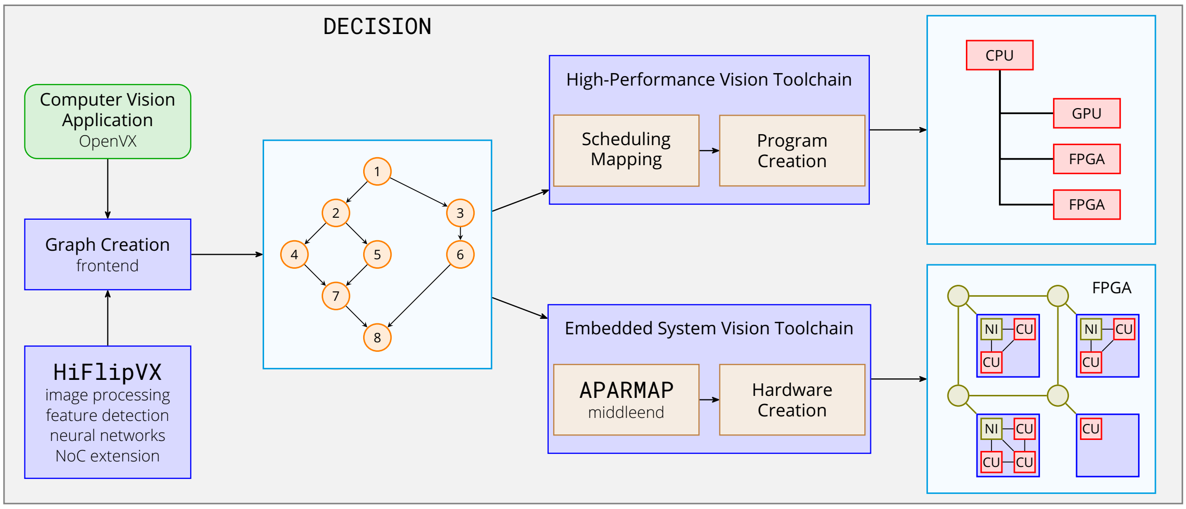


Figure 1.1: `DECISION` framework including `HiFlipVX` object detection library and `APARMAP` application distribution algorithm. Maps computer vision application to heterogeneous HPC architecture (top), or creates an application specific NoC-based architecture (bottom). NI (Network Interface), CU (Compute Unit)

- `HiFlipVX`: An open-source, portable, highly parameterizable, resource and performance optimized HLS (High-Level Synthesis)-based FPGA library containing 66 computer vision functions for implementing object detection algorithms.
- `DECISION`: A modular framework for FPGA-based embedded and high-performance systems that enables efficient programming with an OpenVX-based frontend, automatically builds application and hardware, and integrates `HiFlipVX`.
- `APARMAP`: A scalable application distribution algorithm for partition-based and mesh-like topologies that uses load balancing techniques and heuristics in a multithreaded grid-based algorithm to generate an application-specific hardware architecture.

The first contribution is **`HiFlipVX`**, an open-source HLS-based FPGA library for computer vision, optimized for performance and resource efficiency [12]. It is based on the OpenVX standard, but extends it for feature extraction, which is challenging due to unknown element size at design time [13]. Most functions provide additional parameters, such as multiple SIMD (Single Instruction Multiple Data) widths, data types, or kernel sizes, to increase throughput and reduce latency. The various compile-time parameters make it highly parameterizable, thus offering a wide range of possibilities for an optimized design and extensive DSE (Design Space Exploration). It contains 66 computer vision functions including a neural network extension to cover several fields in one library [14]. All functions are streaming capable, which is a key feature to achieve maximum performance by increasing parallelism and reducing off-chip memory access. Using C++ and HLS with OpenVX simplifies cross-platform development between different architectures, vendors, and toolchains. `HiFlipVX` does not require external or vendor libraries, which eases integration into existing projects, usage of different devices and portability to other vendors, as shown for Intel FPGAs [15, 16].

In addition to the OpenVX standard, the library extracted various functions from different algorithms and converted them into generic and reusable functions. These include the FAST (Features from Accelerated Segment Test) corner detector [17], Canny edge detector [18], ORB (Oriented FAST and Rotated BRIEF) feature detector [19], AKAZE (Accelerated KAZE) feature detector [20], and MobileNets neural network [21]. This thesis also compares different combinations of feature extraction algorithms to identify the most promising ones and implement an improved algorithm from them [22, 23]. It implements these algorithms named ORB, AKAZE and FREAK (Fast Retina Keypoint) [24] in an optimized VHDL (Very High-Speed Integrated Circuit Hardware Description Language) design for embedded systems [25, 26], and to improve and evaluate the library. The improved algorithm is based on AKAZE and FREAK and outperforms the other combinations in terms of repeatability.

The second contribution is **`DECISION`**, a framework for the efficient programming of FPGA-based heterogeneous systems. It consists of two toolchains composed of different modules and models that serve as interfaces. Both integrate `HiFlipVX` and use a common OpenVX-based frontend to implement computer vision applications, build the application model, and check their validity. The advantage of this abstraction is that, regardless of the target platform, the user does not have to learn any new concepts or input languages, or deal with the underlying hardware architecture. OpenVX makes it possible to integrate libraries from various vendors or developers. Therefore, this thesis examines the integration of different OpenCL (Open Computing Language)-based libraries for GPU and CPU architectures [27]. Since libraries cannot provide everything, it needs methods to create and integrate custom accelerators. Therefore, this thesis explores the generation of user-defined OpenCL-based kernels using source-to-source compilers [28]. Implementing custom kernels is not trivial and can differ greatly between different architectures. Therefore, this thesis describes the difference for CPU, GPU, and FPGA devices to implement own vision functions in OpenCL [9].

The first toolchain targets x86-based HPC (High-Performance Computing) systems, which can consist of CPUs, GPUs, and FPGAs [27]. It integrates a heterogeneous scheduling algorithm that distributes the application graph across the different architectures, taking the streaming capability of the FPGA into account. It generates synthesis results for FPGAs and profiles functions and devices for others, to annotate the graph with information about resource consumption, computation time, and communication costs. The toolchain creates a runtime optimized program, which manages synchronization, memory transfers and data coherency at design time. The OpenCL-based runtime system excels by its high parallelism and a low overhead. The second toolchain creates an application specific and adaptive NoC (Network-on-Chip)-based architecture. The streaming-optimized architecture enables the reusability of vision functions by multiple applications to improve the resource efficiency while maintaining a high performance. The toolchain automatizes the creation of the hardware design and specialized components, such as NIs (Network Interfaces), DMA (Direct Memory Access) controllers, and a MA (Manager). This MA stores the flow of multiple applications to execute and orchestrate them in an adaptive runtime system.

The third contribution is `APARMAP`, an application distribution algorithm for partition-based and mesh-like FPGA topologies [29, 30]. The proposed algorithm is integrated into the `DECISION` framework, but keeps all models and methods as general as possible to cover a wide range of use cases and to be vendor independent. This thesis chose a combination of synthesis results and analytical models based on the `HiFlipVX` library as one possible way to calculate the model data. The application model is a dataflow graph consisting of tasks and transactions. First, the algorithm maps and schedules these tasks to a flexible set of physical nodes that form a node graph. The platform model enables heterogeneous architectures with irregular structure that provide high flexibility. This architecture uses a NoC as one possible instantiation for inter- and intra-chip communication. The NoC connects the NIs to PRRs (Partial Reconfigurable Regions), which can be entire FPGAs or regions, using routers, to be applicable for SoCs (Systems-on-Chip) or large FPGA clusters.

The main part is the clustering and placement of the node graph into the platform model in a multithreaded approach. It processes both parts in one to not lose optimality. First, the algorithm maps the node graph to the platform graph within a two-dimensional Euclidean vector space. It uses load balancing techniques to find reasonable solutions within a predictable and scalable amount of time. Second, the algorithm optimizes the solutions using various heuristics. SA (Simulated Annealing) allows solutions to get worse up to a certain value, which decreases from time to time. TS (Tabu Search) uses a multilevel history to prevent the algorithm from calculating the same solutions. The constraints and objectives are the FPGA resource utilization, NoC bandwidth consumption, NoC hop count, and execution time of the proposed algorithm. The algorithm divides the solution space into a grid to prevent threads from calculating the same solution.

1.3 Thesis Outline

The next chapter will discuss the SoA (State-of-the-Art) and provide the necessary background knowledge that will help in understanding this thesis. Its topics are object detection, FPGA-based heterogeneous systems, application distribution, programming methods, and toolchains. Chapter 3 will describe the `HiFlipVX` library, the implemented object detection algorithms, and the improved feature extraction algorithm. Chapter 4 will discuss the `DECISION` framework with the modules and models of its two toolchains targeting embedded and high-performance systems. Chapter 5 will describe the `APARMAP` application distribution algorithm that uses load balancing techniques and heuristics. The last chapter concludes this thesis and provides an outlook for future research.

2 Background

This chapter provides the background information necessary to understand the following chapters based on the current SoA. For an efficient programming of parallel and heterogeneous FPGA-based systems, this thesis identified five different research topics. The first part discusses the different types of object detection algorithms. The second part explores different types of architectures and their associated compute nodes, memory architectures, communication infrastructures, and topologies. As heterogeneous systems and object detection algorithms become more complex, it requires efficient algorithms for a near-optimal application distribution. There is a need for programming languages, models and libraries that are both expressive and simple so that the user can program applications that are as performant as possible. Finally, it needs a comprehensive toolchain that acts as a link between the mentioned parts and provides the user with an automated toolflow and an easy-to-use interface.

Object detection is a subfield of computer vision and image processing. Many applications areas need to detect objects, such as medical x-ray imaging, advanced driver assistance or UAVs [3, 1]. Section 2.1 investigates the topic of object detection. This thesis divides the topic into three different techniques, namely (1) pattern recognition, (2) traditional machine learning, and (3) deep learning. Deep learning is based on neural networks that are trained to detect objects. One advantage of deep learning is that it does not require complex feature engineering. On the other hand, pattern recognition has the advantage that it does not need training data. Pattern recognition is based on the extraction of features or points of interest from images. Traditional machine learning algorithms also extract features and then use them in a trainable classifier. The focus of this work is on the improvement and acceleration of feature extraction algorithms, which are part of pattern recognition and traditional machine learning algorithms. Many applications rely on these algorithms, e.g., when little or no training data is available, as in SLAM (Simultaneous Localization and Mapping) [31].

Section 2.2 explores heterogeneous systems consisting of different parallel computing architectures. It divides the field of computer architectures into five areas: (1) computing (2) memory (3) communication (4) topology and (5) I/O (Input/Output). The computing area includes all PEs (Processing Elements) of the system. This thesis considers PEs as the smallest units that perform computations. Depending on the constellation of nodes, a system may contain different communication types for off-chip/on-chip or inter/intra communication. Thereby, the chosen topology strongly depends on the selected communication types. The focus of this work is on the use of heterogeneous systems containing FPGAs. In this context, it investigates various systems from the embedded and HPC areas. One difficulty with these two areas is the terminology. Based on the research results, this paper extends the OpenCL terminology for the computing field so that it is suitable for both areas.

One challenge in designing or deploying FPGA-based systems is the distribution of the application(s), which strongly depends on the communication model and the topology of the overall system. As a result of the increasing number of nodes in heterogeneous and distributed systems, it needs a good algorithm to place communicating nodes close to each other. Due to the high complexity of application distribution algorithms, heuristics, metaheuristics, or even neural networks are used to solve these problems in a reasonable time. Section 2.3 investigates the different steps needed to create an application-specific FPGA-based system from one or more application(s). Based on the SoA, this thesis has identified six areas within this process: (1) partitioning (2) tuning (3) mapping (4) scheduling (5) clustering and (6) placement. This process is a very complex problem, and no one has yet addressed all these areas in one work. Therefore, this thesis examines the problem for different FPGA-based systems from different perspectives. These perspectives include (1) mapping and scheduling in NoC-based MPSoCs (Multiprocessor Systems-on-Chip), (2) design space exploration for neural networks, (3) clustering and placement in FPGA-based CAD (Computer-Aided Design) tools, and (4) tuning, clustering, and placement for PRRs.

Section 2.4 focuses on the programming of heterogeneous parallel computer architectures. There are several approaches that are highly expressive on the one hand and simplify the programmability on the other hand. This thesis divides these approaches into two categories, namely domain-specific and general-purpose. GPLs (General-Purpose Languages) such as OpenCL or CUDA (Compute Unified Device Architecture) often contain explicit constructs to program parallel architectures. These can be directive-based approaches such as OpenACC or OpenMP (Open Multi-Processing), which simplify programming when using GPLs. By adding a domain, such as computer vision, the programmer can use both DSLs and libraries to improve efficiency and performance. Depending on the architecture or use case, different programming methods are useful. This thesis uses several C/C++-based approaches such as (1) OpenCL, which is a GPLs that supports the widest range of architectures, (2) HLS, which is a directive-based approach to program FPGAs and (3) OpenVX, which includes a domain-specific library for computer vision. OpenVX is an open, royalty-free standard for cross-platform acceleration of computer vision applications [32].

Section 2.5 investigates the SoA for the various modules of the vision framework developed in this thesis. The first part focuses on source-to-source compilers, also called transpilers, and introduces the needed compiler topics. This thesis examines transpilers to create user-generated and accelerated functions that the toolchain can easily integrate. The second part focuses on HLS-based tools and discusses their benefits and realization by researchers and vendors. This thesis uses HLS-based toolchains to implement a C++-based object detection library for FPGAs. The third part focuses on FPGA-based tools that implement part of the OpenVX standard. This thesis uses OpenVX as a common frontend for the toolchain to program both embedded and HPC systems. The last part focuses on OpenCL-capable toolchains. On the one hand, this thesis uses the OpenCL device code to program non-FPGA devices. On the other hand, it uses the OpenCL host code in a runtime system as a low-level API (Application Programming Interface) for x86-based systems.

2.1 Object Detection

This section will look at the object detection topic, which is a subfield of computer vision and image processing, from different perspectives. Many application areas need to detect objects, such as autonomous driving, hand gesture recognition, medical x-ray imaging, advanced driver assistance or UAVs [1, 2, 3, 4, 5]. Many implementations rely on DSPs (Digital Signal Processors), GPUs, FPGAs, or ASICs (Application-Specific Integrated Circuits), since the field of computer vision often involves very computationally intensive operations. Various publications have shown the good performance and energy efficiency of FPGAs compared to GPUs and CPUs for image processing and computer vision related tasks [33, 7, 8, 10]. Depending on the application and its streaming capability the performance can be lower, higher, or similar to GPUs. However, when it comes to energy efficiency FPGAs can even outperform GPUs by a factor of 10 and more [9].

As shown in Figure 2.1, this work divides the object detection topic into three techniques. Deep learning is based on neural networks that are trained to detect objects. One advantage of deep learning is that it does not require complex feature engineering. With a good dataset for training, deep learning algorithms can achieve a significantly better detection rate than the other techniques.

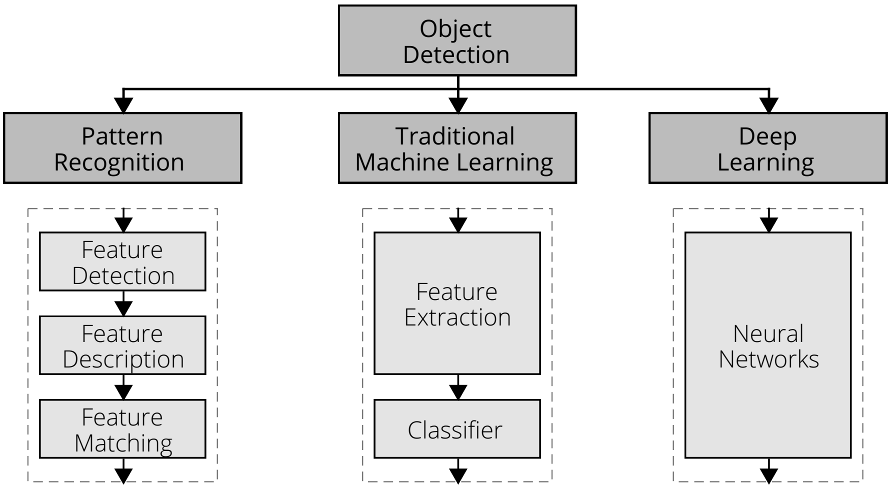


Figure 2.1: Three different object detection techniques in the computer vision field.

The advantage of pattern recognition is that it does not need training data. The technique is based on extracting features or points of interest from images. These features can be, for example, corners, edges, or blobs. It is important that the algorithm detects features regardless of their viewpoint, rotation, size, or image quality. A descriptor needs to describe these features so that they can be compared with each other.

Deep learning is a subfield of machine learning. However, traditional machine learning algorithms first detected and described features before they can classify them. Therefore, they often use the same algorithms as in pattern recognition to extract features. Due to the mix of feature extraction and training of a classifier, traditional machine learning is suitable for use cases where there is not enough data to train a neural network. In addition, deep learning algorithms can be computationally intensive and consume a lot of resources in comparison to other algorithms.

However, the focus of this thesis is on feature extraction. Many applications rely on these algorithms, e.g., when little or no training data is available, as in SLAM [31]. The following sections present the SoA in the field of pattern recognition and discuss device-specific implementations.

2.1.1 Pattern Recognition

Pattern recognition algorithms can still outperform machine learning algorithms when, for example, the required data is unavailable or insufficient. One important step in many pattern recognition algorithms is the extraction of features. However, feature extraction also plays an important role in classical machine learning algorithms. Features are points of interest extracted from images to describe their content. A feature can be a corner, an edge, or a region in an image. To detect the same features in different images, it is very important to achieve high repeatability and distinctiveness against different image transformations, such as viewing angle, blur, or noise. To achieve high repeatability, an algorithm must detect features at different locations invariant to rotation, scale and changes in brightness or contrast.

The feature extraction process contains two steps, namely feature detection and feature description. Numerous computer vision tasks, such as object detection, object tracking and SLAM, need to extract features. Based on the current state of research, this thesis focuses on the following three algorithms:

- AKAZE [20] detection: Due to the nonlinear scale-space in this algorithm, it outperforms other algorithms in terms of repeatability.
- ORB [19] detection: It outperforms other algorithms in terms of computational speed and achieves good results in terms of repeatability.
- FREAK [24] description: The algorithm is resource efficient, not as computationally intensive, and when combined with AKAZE or ORB, it achieves better repeatability results than other descriptors.

2.1.2 Feature Detection, Description and Matching

Features Detection: One of the first steps in many computer vision applications such as object recognition, image registration and motion tracking, is feature detection [34]. Image features can be divided into two main categories, namely global and local features [35]. Global features usually describe an entire image, for example, using a multidimensional feature vector. Among other things, this vector contains information regarding resolution, colors, textures, and shapes. Local features, on the other hand, describe individual points of interest in an image separately. They are called features, points of interests or keypoints. As shown in Figure 2.2, one can distinguish between three different types:

- **Corner**: FAST [36, 17, 37] and Harris [38]
- **Edge**: Canny [18] and Sobel
- **Blob**: DoH (Determinant of the Hessian) and DoG (Difference of the Gaussian)

Scale-Space: Multiscale feature detection algorithms build a scale-space to detect features independent of their scale. This scale-space represents the same image in different sizes by resizing the image as shown in the middle of Figure 2.2. How this scale-space is created may vary among different feature detection algorithms.

Feature Descriptor: A feature descriptor describes the individual features to create a unique pattern so that they can be compared. It describes a feature based on its neighboring pixels

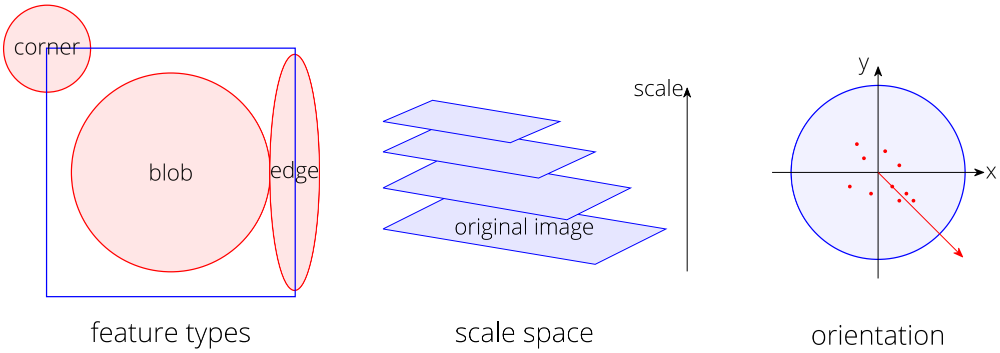


Figure 2.2: Three different feature types (left). Scale-space needed for scale invariance (middle). Orientation needed for rotation invariance (right).

within a certain range, which depends on the scale size of the feature. Mostly, it selects and compares the surrounding pixels according to a certain pattern. An important property of a good feature description algorithm is that the described features are invariant to rotation. It can achieve this by calculating the orientation of the surrounding pixels, as shown on the right side of in Figure 2.2. Feature descriptors usually store their results in a floating-point or binary representation.

- **Floating-point**: like in SIFT (Scale Invariant Feature Transform) [39] or SURF (Speeded Up Robust Features) [40]
- **Binary**: like in BRIEF (Binary Robust Independent Elementary Features) [41, 42], FREAK [24] or BRISK (Binary Robust Invariant Scalable Keypoints) [43]

Binary Descriptor: A binary feature descriptor is a feature vector consisting only of a sequence of ones and zeros. It describes each feature by a bit vector. One advantage over floating-point descriptors is that it can use the Hamming distance instead of the Euclidean distance for feature matching. Many algorithms use them to improve the computational speed of the feature description and the subsequent algorithms. They work with intensity comparisons between the pixels surrounding the feature using sampling patterns. For example, the descriptors BRIEF and rBRIEF (rotated BRIEF) use random sample points around the selected features. The BRISK descriptor uses concentric circles around the feature and then performs intensity comparisons. The FREAK descriptor is based on the retinal scanning pattern to perform intensity comparisons.

Properties: Feature extraction algorithms can differ in several properties regarding their ability to detect and describe features. Mikolajczyk et al. [44] categorize these properties as follows:

- **Robustness** describes the ability of detecting the same feature locations regardless of scale, rotation, shifting, or problems caused by the image quality, such as compression or noise.
- **Repeatability** is the ability of recognizing the same feature under different photometric and geometric transformations.
- **Accuracy** is the ability of a feature detection algorithm to precisely locate the pixel position of image features. This property is particularly important for image comparison tasks.

- **Generality** describes how well features can be detected in different types of applications.
- **Efficiency** is the ability to detect features in images in a reasonable time. This property is especially important for real-time tasks.
- **Quantity** describes the ability of a feature detection algorithm to detect all or most of the features in an image.

Feature Matching: Many pattern recognition algorithms are based on the feature detection and description process. Simple object detection algorithms compare features of two different images to find the same object in both images. They compare the descriptors of two or more images to identify similar features. They use distance metrics to determine the similarity or distance between two feature descriptors. The most used distance metrics are the Euclidean distance for floating-point descriptors and the Hamming distance for binary descriptors. They use algorithms like k-NN (k-Nearest Neighbors) to eliminate outliers. An object tracking algorithm compares consecutive images of a video with each other. Algorithms such as SLAM [31] use well-known feature extraction algorithms, like ORB. SLAM refers to a method of robotics in which a mobile robot must simultaneously create a map of its environment and estimate its location in space within that map.

2.1.3 Multiscale Feature Detection and Description Algorithms

Several algorithms exist that detect and describe image features in a multiscale environment. Two of the first and most known multiscale feature detection and description algorithms are SIFT [39] and SURF [40]. SIFT has achieved remarkable success in many computer vision applications and SURF is a computational efficient alternative. Both approaches and many related algorithms build on the Gaussian scale-space and use its derivatives as smoothing kernels for scale-space analysis. This approach has some important drawbacks since Gaussian blurring does not preserve object boundaries and smooths both detail and noise to the same degree at all scale levels. SIFT identifies features using the DoG. Its descriptor is represented by histograms of image gradients calculated at each image point around the detected feature. SURF uses integral images to accelerate the convolution, and its detector is based on the Hessian matrix. Its descriptor is like SIFT but uses Haar wavelets. While SURF has its advantages within its computational speed, SIFT outperforms SURF in cases of image transformations like rotation and scaling.

Several feature detection and description algorithms, such as ORB, BRISK, KAZE, AKAZE and FREAK have been proposed to improve the computation time and/or repeatability of SIFT and SURF. Alahi et al. [24] proposed FREAK, which is a binary descriptor inspired by the human retina. It describes features by efficiently comparing image intensities across a retinal sampling pattern. Leutenegger et al. [43] proposed BRISK, which uses the FAST detector. They use a binary descriptor computed from intensity comparisons obtained by selectively sampling each feature in its neighborhood [45].

Rublee at al. [19] proposed ORB, which combines the FAST corner detector [36, 17, 37] and the BRIEF feature descriptor [41, 42]. They improve the algorithms by adding a pyramid scheme to achieve scale invariance and the intensity centroid function to achieve rotation invariance. Different works show that ORB performs better than SIFT, SURF, BRISK and AKAZE in terms of computation time. FAST is a high-speed corner detector that detects corners, by comparing 16 pixels that lie on a circle around a feature. BRIEF is a binary descriptor that is based on pairwise intensity comparisons of image patches [45].

Alcantarilla et al. [46] proposed KAZE, to improve the repeatability of feature extraction by creating a nonlinear scale-space. They show that KAZE achieves a better repeatability than other algorithms, such as ORB, BRISK, SIFT and SURF. Alcantarilla et al. [20] proposed AKAZE, to close the performance gap between KAZE and algorithms such as ORB and BRISK by using FED (Fast Explicit Diffusion), since the creation of the nonlinear scale-space is a computationally intensive process.

Only a few works have dealt with the comparison of the different algorithms and the testing of new combinations. D. Dwarakanath et al. [45] compared various combinations of different feature detectors, like SIFT, SURF, BRISK, ORB, KAZE and AKAZE, and different descriptors, like BRIEF and FREAK, for stereo vision of 3D applications. The best results in terms of repeatability they achieved with the combinations of AKAZE-BRIEF and of KAZE-FREAK. Unfortunately, they did not achieve good repeatability results for the combination of AKAZE-FREAK. However, the KAZE algorithm in its original form performed best, but is also the slowest in terms of computational speed. Song et al. [47] use the FREAK algorithm in a solar image matching software. They combined the SIFT, SURF and AKAZE detectors with FREAK and evaluated different configurations to find the most suitable for their use case. The AKAZE-FREAK combination provided the best results for their application.

2.1.4 ORB

The ORB [19] algorithm relies on the oFAST (oriented FAST) corner detector [36, 17, 37] and the rBRIEF feature descriptor [41, 42]. Both offer good performance with low computational costs. They augment the FAST detector with a pyramid scheme to create a multiscale algorithm and achieve scale invariance. They add the Intensity Centroid function to the FAST to compute the dominant orientation of a pixel and achieve rotation invariance [48]. They add a Harris corner detector [38] to reject edges and provide a reasonable score. The detector passes the orientation values to the rBRIEF descriptor in addition to the features. BRIEF is a descriptor that uses simple binary tests between pixels in a smoothed image patch, to create binary strings, to reduce its construction and matching time.

The pyramid scheme consists of multiple levels, to detect features at different scales to achieve scale invariance. Each level represents a scaled image of the original image. However, this also means that the algorithm performs the subsequent functions on each image in the scale-space to detect features. There are various methods for resizing an image. For example, there is the nearest neighbor method, bilinear interpolation, and area interpolation. The nearest neighbor method is computationally faster but loses more image information because it takes the closest pixel of the original image to create the pixel of the resized image. The interpolation methods try to include the image information of the neighboring pixels to reduce this information loss.

The FAST corner detector uses a Bresenham circle of radius three, as shown in Figure 2.3. A segment test considers the 16 pixels on the Bresenham circle and the center pixel p (pixel under test). The detector classifies p as a corner if there are n contiguous pixels on the Bresenham circle, which satisfy one of the two conditions. It was observed that for $n = 9$ contiguous pixels the best results were obtained.

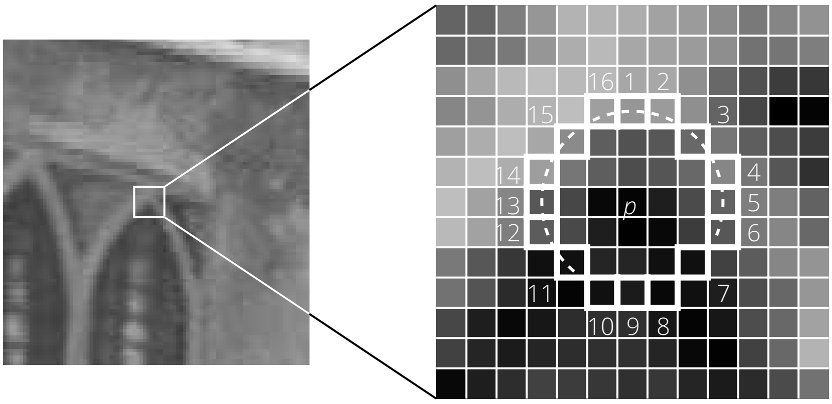


Figure 2.3: The 16 contiguous pixels of the Bresenham circle used in the FAST detector to detect corners [17].

- All pixels are brighter than the intensity of pixel p plus a threshold t $(I_p + t)$.
- All pixels are darker than the intensity of pixel p minus a threshold t $(Ip - t)$.

One problem that the FAST detector faces is that it detects multiple features adjacent to one another, which means that it detects redundant corners. To solve this problem, it applies NMS (Non-Maximum Suppression) on the detected features. This method suppresses a corner if it does not have the local maximum value. Therefore, it searches in a squared window around the pixel if all other pixel values are below the value of the observed pixel. To apply NMS, there must be a score or response function that describes the strength of a corner (V).

$$v = max\left(\sum_{x\in S_{bright}} \left|I_{p\to x} - I_p\right| - t, \sum_{x\in S_{dark}} \left|I_p - I_{p\to x}\right| - t\right) \tag{2.1}$$

$$S_{bright} = \left\{x \,|\, I_{p\to x} \geq I_p + t\right\}, \quad S_{dark} = \left\{x \,|\, I_{p\to x} \leq I_p - t\right\} \tag{2.2}$$

The intensity centroid function assumes that the brightness centroid of a feature is offset from its center. Thus, when considering a field around a feature, where the feature is the center, a vector that starts at the center and points to the centroid is used to determine the orientation. Defining the moments as m_{pq}, the centroid can be determined as C and the orientation of the patch as θ, where $atan2$ is the quadrant-aware version of $arctan$.

$$\theta = atan2(m_{01}, m_{10}) \tag{2.3}$$

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \tag{2.4}$$

2.1.5 (A)KAZE

Alcantarilla et al. [46] proposed KAZE, a multiscale 2D feature detection and description algorithm. KAZE describes 2D features in a nonlinear scale-space by means of nonlinear diffusion filtering. It is the first algorithm to use the nonlinear diffusion in multiscale feature detection. The advantage is that nonlinear diffusion filtering is adaptive to the local image structure, as shown in Figure 2.4. This reduces noise and keeps image boundaries in higher scale levels, unlike the Gaussian scale-space (linear diffusion), which evenly smooths all structures in an image. The algorithm shows that it is possible to obtain multiscale features that achieve much higher repeatability and distinctiveness than other algorithms, by using nonlinear diffusion filtering. The main drawback of KAZE is the computational intensity needed to build the nonlinear scale-space.

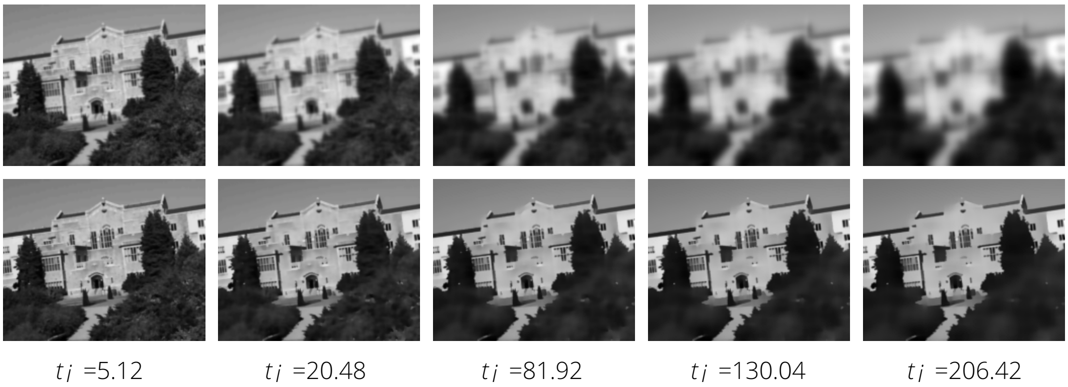


Figure 2.4: Comparison between the Gaussian and nonlinear diffusion scale-space for several evolution times t_i. First row: Gaussian scale-space (linear diffusion). Second row: nonlinear diffusion scale-space [46].

Alcantarilla et al. [20] also proposed AKAZE, which is based on KAZE. It aims at reducing the computation time, while preserving the high repeatability of KAZE. AKAZE uses FED to accelerate the creation of the nonlinear scale-space. In addition, FED is simpler to implement and more accurate. Furthermore, they introduced a binary descriptor called M-LDB (Modified-Local Difference Binary). It is rotation and scale invariant and reduces computation time and memory requirements compared to M-SURF (Modified SURF) used in KAZE. The implementation of the AKAZE algorithm contains four parts: (1) computing the contrast factor, (2) building the nonlinear scale-space, (3) detecting features and (4) describing features.

The contrast factor (k) is important for building the nonlinear scale space. It first smooths the image with a Gaussian filter. It then calculates the maximum absolute gradient values (h_{max}) of the smoothed image pixels. It then fills a histogram of 300 *bins* with the gradient values. It then loops over the histogram to find its index i at which the 70 percentile of the gradient histogram is achieved.

$$k = \frac{h_{max} \cdot i}{300} \tag{2.5}$$

The scale-space is built by numerically solving the partial differential equation iteratively using FED with varying time steps τ_i.

$$\frac{\partial L}{\partial t} = div(g(|\nabla L_\sigma|) \cdot \nabla L) \tag{2.6}$$

$$L^{j+1} = (I + \tau_j A(L^j))L^j \tag{2.7}$$

$$g(|\nabla_\sigma|) = \frac{1}{1 + \frac{|\nabla L_\sigma|^2}{k^2}} = \frac{k^2}{k^2 + |\nabla L_\sigma|^2} \tag{2.8}$$

$g(|\nabla L_\sigma|)$ is the conductivity function (or flow function) and $A(L)$ is the matrix notation of this function. The conductivity function depends on the contrast factor k. This thesis adjusted the equation, to prevent from using two division operations. For each pixel, computing the conductivity function requires one division, two multiplication and two addition operations. Additionally, each FED step needs five multiplication and twelve add/subtract operations for each pixel. The conductivity function and the high amount of FED steps is what makes AKAZE so computationally intensive.

The scale-space of AKAZE is a pyramidal framework. It consists of octaves and each octave consists of sublevels. The amount of both octaves and sublevels is four in the baseline configuration. For each new octave it quarters the image size.

After creating the nonlinear scale space, AKAZE calculates the DoH blob detector for each image in scale space. The scale size of the DoH increases with each sublevel. That is also why AKAZE downscales the images at each octave. A pixel of the DoH image is a feature if its value exceeds a threshold T and is maximum in a 3×3 window within the DoH image. A further stage filters the resulting features by comparing all features with each other. It suppresses a feature if its value is not maximum in a $\sigma \times \sigma$ window of the same, the upper and the lower scale-space level. At the end, the algorithm calculates a SR (Subpixel Refinement) for each feature.

2.1.6 FREAK

Alahi et al. [24] presented the FREAK descriptor, which creates a binary string through intensity comparisons. It uses the human eye as a model, where intensity comparisons are based on the pattern of the retina. FREAK does not use all pixels of the neighborhood, but maps values from areas of the neighborhood according to a certain pattern.

The retina has four areas that perform different tasks. It is noticeable that the density decreases exponentially towards the outside of the retina. The FREAK creates the sampling pattern according to the distribution of each area. The inner part of the retina shows many small circles, while the outer part shows large circles. These circles correspond to receptive fields in the eye, as shown in Figure 2.5. As a result, it weights points that are close to the center more heavily than those located further away. The radii of the circles increase exponentially towards the outside, while the weights of the sampling points decrease exponentially.

The algorithm calculates an integral image for the fast calculation of the area of these circles. Furthermore, it simplifies the circle with radius r by placing a rectangle around the circle. The value of a pixel in the integral image ($I_{x,y}$) represents the sum of all pixel values of the input image whose x and y coordinate is smaller or equal. The algorithm can calculate a square area in the input image, by using the values of the four corner pixels in the integral image.

$$\text{intensity} = \frac{I_{x+r} - I_{x-r} + I_{y+r} - I_{y-r}}{2 \cdot r^2} \tag{2.9}$$

It compares two different sample points for each intensity comparisons. Having 43 sample points results in 903 different intensity comparisons. Furthermore, it only compares sample points whose receptive fields overlap. The descriptor consists of a total of 512 of these pairs. Increasing this number did not show any noticeable improvement. The descriptor results in a 512 bit wide binary vector describing the feature. It assigns 128 bit to each of the four areas

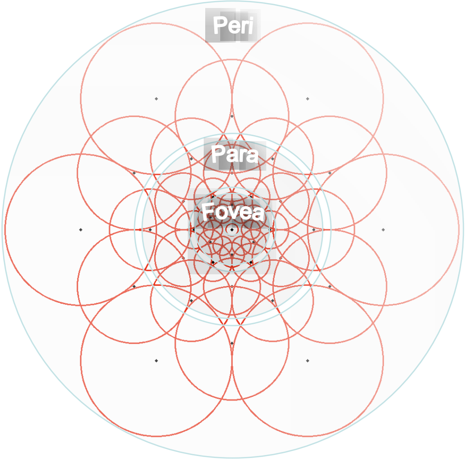


Figure 2.5: Illustration of the FREAK sampling pattern similar to the distribution of retinal ganglion cells with their corresponding receptive fields. Each circle represents a receptive field [24].

of the retina. The importance of these partial vectors decreases starting from the innermost region of the retina.

For the descriptor to become invariant to rotation, it forms the gradients between the sample points. It uses only pairs that have a particularly large distance to each other and can be mapped to each other due to their symmetry. The descriptor then takes the average of these gradients to determine its main orientation. It uses 45 pairs to calculate the orientation. Due to the large receptive fields in the outermost region, the errors in the orientation calculation are allowed to be larger than in BRISK. Due to this approach, the memory required to calculate the orientation is 1/5 of BRISK.

2.1.7 FPGA Implementations

Detecting and describing features on high-resolution images at high frame rates requires an optimized implementation. Depending on the required computational power, it needs either a multicore CPU, a GPU or an FPGA. The following will look at the various implementations from the SoA. It will mainly focus on AKAZE, ORB and FREAK, as this thesis also implements them. Table 2.1 shows five different FPGA implementations that focus on these three feature extraction algorithms.

Two different research groups investigated the implementation of the ORB algorithm. Lee et al. [54, 49] proposed an FPGA implementation of the ORB algorithm. Fularz et al. [50] proposed an FPGA implementation of the FAST corners and the BRIEF descriptor, which are part of the ORB. In the second work, the authors omit the implementation of the orientation, which is necessary to achieve rotation invariance. Both implementations did not create a scale-space, so the implementations are not scale invariant in comparison to the original algorithm. The creation of a scale-space would multiply the resource consumption of the detector. The second work uses a higher resolution, thereby only achieves a lower frame rate. However, they calculate 3.24 times more pixels per second than in the other work. Unfortunately, there is no reliable information about the repeatability in both papers, which makes a more precise evaluation of the design difficult.

Table 2.1: Different FPGA implementations for feature detection and description.

	ORB Lee [49]	FAST & BRIEF Fularz [50]	AKAZE Jiang [51]	FREAK Bello [52]	SURF & FREAK Zhao [53]
Resolution	640 × 480	1920 × 1080	1920 × 1080		800 × 600
FPGA type	Artix7	Artix7	Virtex5	Virtex5	Kintex7
Frequency [MHz]	100	100	200	108	122
fps	100	48	111	15	60
FF	6411	9543	*0.95M gates*	1975	60 044
LUT	31 677	4118		6706	147 190
DSP				14	139
BRAM [36 kbit]	31	31	*2.12* Mbit	4	289

The main problem of the AKAZE algorithm is the computation intensiveness. Jian et al. [51] presented an FPGA implementation of the AKAZE algorithm that achieves real-time performance. However, the design is not that flexible to be integrated in other systems. They made several optimizations to the algorithm to improve performance and reduce resource utilization. They achieve a 111 fps (frames per second) for an image resolution of 1920 × 1080. The good performance comes with a small loss of repeatability compared to the original implementation. A missing part in their work is the calculation of the contrast factor, which is essential for the construction of the nonlinear scale-space. In addition, the hardware testing methodology in this paper is very vague.

Only few implementations of the FREAK algorithm exist and none of them implements the complete algorithm. Bello et al. [52] implemented the FREAK on an FPGA with approximations to the original algorithm to accomplish their hardware requirements. For example, they do not implement the integral image which is needed as input by the FREAK algorithm. They achieve 14 fps for 1000 features running at 108 MHz on an XC7Z020. Zhao et al. [53] presented an FPGA-based system for traffic sign detection. In their implementation, they use the SURF detector together with the FREAK descriptor. The resource utilization of their implementation is relatively high, although it includes the SURF algorithm. They reach 60 fps at 122 MHz for an image resolution of 800 × 600. They did not specify the number of processed features, although it is decisive for the computational speed of the descriptor. Both FREAK implementations do not show repeatability results needed to evaluate the accuracy of the implementation. Both implementations also do not integrate the calculation of the orientation needed for rotation invariance.

An important part of the FREAK algorithm is the calculation of the integral image. However, there are also other algorithms that benefit from calculating the integral image, such as the Viola and Jones [55] and the AdaBoost [56] algorithms. Svab et al. [57] implemented the integral image for the SURF algorithm. Chakrasali et al. [58] have taken a similar approach for their implementation. Both show a simple and efficient architecture of the integral image that computes one pixel after each other. Unfortunately, they do not show the performance or resource utilization of the integral image function.

2.2 Heterogeneous Systems

In the past decade, there has been a significant growth towards parallel computing architectures and heterogeneous systems. Primary reasons for this growth are the decreasing transistor size and power wall. Parallel architectures can achieve high performance with an improved energy efficiency, while running at a slower clock speed. In the past, there have been several optimization strategies to increase parallelism and throughput of these architectures. Among them are ILP (Instruction-Level Parallelism), DLP (Data-Level Parallelism), TLP (Task-Level Parallelism), out-of-order execution, branch prediction and specialized instruction sets. The combination of those strategies with leads to a variety of architectures, with different strengths, weaknesses and uses. The architectures have different strengths and require different strategies to exploit their full potential and parallelism. Most known examples are CPUs, GPUs, ASICs and FPGAs. By combining different architectures in a heterogeneous system, one can benefit from the relevant advantages, compensate their disadvantages, and increase efficiency. Heterogeneous systems are becoming more and more important due to the increasing performance and energy requirements of more complex applications.

Before discussing the topic of application distribution, this section addresses some characteristics and distinctions of computer architectures. For this reason, this thesis divides the field of computer architectures into the following five main areas, namely computing, memory, communication, topology, and I/O. The smallest unit observed in the computing area is a PE. Wanhammer [59] describes a PE as follows: "PEs usually perform simple, memory-less mappings of input values to a single output value". When using the term PE in this thesis, it includes the internal memory and control circuits, as such granularity and complexity is sufficient.

Memory architectures can be split into two groups: (1) distributed and (2) shared memory. On the one hand, PEs can access the shared memory symmetrically and with the same access time (UMA (Uniform Memory Access)). On the other hand, PEs can access the shared memory asymmetrically, resulting in different access times (NUMA (Non-Uniform Memory Access)). In distributed memory systems, each PE has its own memory and the PEs exchange data via messages. The field of supercomputing often follows a hybrid approach, where a distributed memory system interconnects several shared memory architectures [60].

The communication infrastructure also plays an important role in heterogeneous parallel computing architectures. The choice of the communication infrastructure, for example a bus, depends on the scalability and number of physical nodes in the system. Communication requirements between the different nodes can be met by using a NoC as a scalable solution to shared buses [61]. NoCs have many advantages when it comes to connecting large architectures and enabling dynamic communication. Further benefits are their modularity, flexibility, reusability and reprogrammability [62]. NoCs are also valuable for data-intensive applications such as: VOPD (Video Object Plane Decoder), MWD (Multi Window Display) and DMC (Depth Map Computation) [63]. Numerous FPGA-based systems use NoCs, such as in RAMPSoC [64], RAR-NoC [65], Hoplite [66] or RingNet [67].

In the embedded system domain, heterogeneous MPSoCs have proven to be a good solution to meet the increasing performance requirements of more complex applications. They can provide higher performance, energy efficiency and lower costs compared to homogeneous MPSoCs [68]. They can have different types of physical nodes, like PUs (Processing Units), ACs (Accelerators) or I/Os, interconnected via a communication infrastructure. Figure 2.6

shows an example MPSoC containing multiple physical nodes, which are interconnected by a NoC. In the HPC domain, heterogeneous FPGA-based clusters are proving to be an ideal solution due to their performance, flexibility, and energy efficiency.

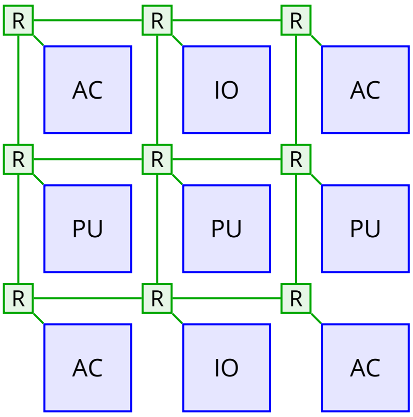


Figure 2.6: Heterogeneous MPSoC using a NoC as communication infrastructure, including Router (R), I/O (Input/Output), PU (Processing Unit) and AC (Accelerator).

Rethinagiri et al. [69] show the potential of heterogeneous platforms containing CPUs, GPUs, and FPGAs for both HPC and embedded systems. Furthermore, they show the energy efficiency of different platform combinations. Applications have been hand optimized using C/C++, CUDA and VHDL. However, there is also a version that shows the potential of using OpenCL as a common language for all platforms. They used OmpSs [70] to program the XILINX FPGA using OpenCL. Unfortunately, they did not optimize the OpenCL model and only achieve 1/8 of the performance of the hand optimized implementation.

The next subsection will introduce FPGAs and their benefits over other compute devices. The following subsection will present different FPGA-based systems from the embedded system and HPC domains. Thereby, this thesis investigated different computer architecture areas, with focus on computing, communication, and topologies. The end of this section will summarize the gained knowledge, refer to this thesis and present a common terminology for the embedded system and HPC domains.

2.2.1 FPGA-based Systems

FPGAs enable rapid prototyping, fast emulation, and exploration of new architectures without the overhead of ASIC production [61]. They offer reconfigurability, a higher degree of flexibility, and lower development costs than ASICs, for both embedded and HPC systems. Recent developments have brought FPGAs to the point of closing the gap to GPUs, in terms of raw performance, while being more energy efficient [8]. For example, XILINX's Alveo U250 FPGA offers 1728k LUTs (Lookup Tables), 3456k FFs (Flip-Flops), 12 288 DSPs and 54 MB of on-chip memory, achieving up to 33.3 INT8 TOPS (Terra Operations per Second) of raw performance. To further increase energy efficiency and flexibility, DPR (Dynamic Partial Reconfiguration) can be used to reconfigure selected regions of an FPGA at runtime while the remaining regions continue to operate [71]. The reconfiguration time of a region is faster than that of the entire FPGA, since it mainly depends on the byte size of the bitstream [71]. A bitstream is a generated file that is needed to configure the FPGA.

In recent decades, power consumption has become a dominant factor in the development of computer architectures, making the energy efficiency of the various compute devices increasingly important [6]. Many researchers have shown that FPGA-based systems can achieve comparable compute performance to high-end GPUs and higher energy efficiency than CPUs and GPUs [7, 8]. While ASICs are still more efficient than FPGAs, newer FPGA technologies are diminishing the gap as they also incorporate more dedicated DSPs and on-chip BRAM (Block Random-Access Memory) [7]. Due to the enormous computational capacity needed in various application fields, researcher have introduced several approaches that address this problem by combining multiple FPGAs in one large system [72, 73, 74, 75]. One challenge of such systems is the scalability, as they need to distribute multiple compute devices over a large infrastructure [76].

An increasingly common solution is the use of clusters consisting of efficient compute devices interconnected by high-performance networks [77, 76, 6]. The slowdown of Moore's Law combined with the growing size of data center infrastructure makes specialized compute devices extremely valuable [76]. Therefore, using an FPGA-based cluster is a simple and effective technique to achieve high computational performance thanks to massive parallel processing [78]. FPGA-based clusters are widely used for the internet of things, digital signal processing, and prototyping [79]. Recently, data centers with clusters that contain multiple FPGAs or clusters that directly connect multiple FPGAs have become more common [80, 81, 79]. In this context, widely used environments such as Hadoop [73] and Spark [80, 79] are used for distributed computing.

FPGAs provide a high degree of reconfigurability and flexibility, which makes them very suitable for hardware emulation and prototyping. Due to advances in semiconductor technology and manufacturing, they can emulate very large systems. However, large FPGA systems, such as the CHIPit from Synopsys, can be very costly. To reduce these costs, a cluster of off-the-shelf FPGA boards [82], such as proposed by Mentens et al. [83], can be used as an emulation platform. Examples such as the Zedwulf of Moorthy et al. [74] and the ZCluster by Lin et al. [73], demonstrate the growing need for such low-cost FPGA clusters. Both approaches combine several low-cost commercial FPGA boards, such as the ZedBoard [82] containing an ARM-FPGA-based SoC. Using an FPGA cluster does not only reduce the costs of an emulation platform, but also increases the maximum number of available resources. Another advantage is expandability, as new FPGAs can be added without having to replace the entire system when running out of resources.

2.2.2 Communication and Topology

Off-chip communication, between different components of an FPGA-based cluster, increases latency and limits data throughput. To design high-performance clusters, a scalable topology and an efficient communication infrastructure with low overhead is needed [78]. One challenge is the interconnection between the different compute devices. They can be connected either directly or indirectly, as shown in Figure 2.7. In a direct network, each node in the network is both an FPGA and a switch. In an indirect network, the nodes are either a switch or an FPGA.

A large portion of existing FPGA clusters consist of multiple CPU-based compute nodes, where one or more FPGAs are connected to the rest of a compute node (e.g. the CPU) via PCI-e (Peripheral Component Interconnect Express) bus. In many cases, the communication and

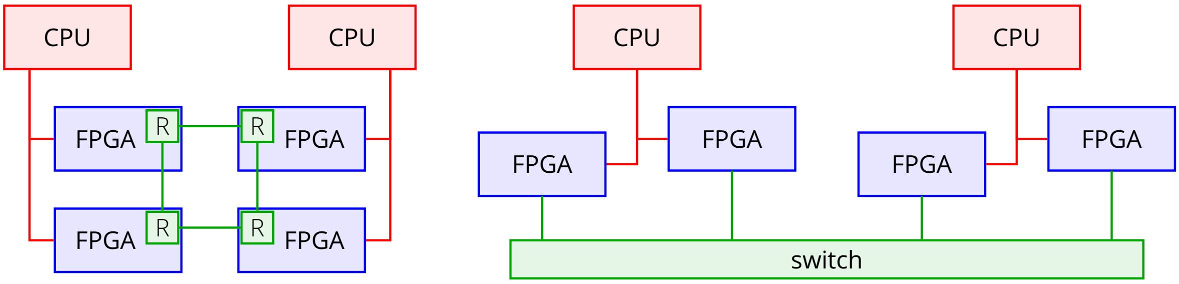


Figure 2.7: FPGA-based cluster with inter-FPGA connection using a direct (left) or indirect (right) network. R = Router

control between different compute nodes in a cluster is done by the CPUs via Ethernet [77, 6]. The detour of sending data via the CPU often results in a bottleneck and increases latency. Therefore, many clusters additionally connect all FPGAs in the system via an Ethernet switch, such as in the Microsoft Catapult system [76]. Indirect inter-FPGA communication without the detour of sending data via the CPU enables a lower latency and higher bandwidth [84], as shown in Figure 2.7 on the right.

However, the use of switches can limit the scalability of the overall system. Therefore, tightly coupled FPGA clusters contain a dedicated network in which the FPGAs are directly connected to each other [78], as shown in Figure 2.7 on the left. In this type of connection, each FPGA requires its own router. The communication between two FPGAs is done via two or more routers. The maximum number of routers on the communication path between two FPGAs depends on the size and topology of the cluster.

In many systems with a direct connection between FPGAs, they are interconnected in a ring or line topology, which is sufficient for medium sized clusters [85, 86, 79, 87]. In the various publications, Gbit transceivers have shown to be among the best performing and most efficient physical link types between FPGAs. Many clusters suffer from poor scalability and become inefficient when it comes to building a large cluster. Therefore, mesh and torus topologies provide a significantly improved scalability of the overall system and can connect hundreds of compute nodes within a cluster [78].

A common problem in large distributed systems is that different protocols cannot communicate directly with each other. Wu et al. [88] presented a flexible FPGA-based inter-node communication interface that can transmit different bus protocols, such as PCI-e or Ethernet, simultaneously. Other researchers rely on less generic approaches for the communication interface. Some are specifically designed for FPGAs and can therefore be implemented more resource efficient. Markettos et al. [75] describe building a reliable FPGA cluster containing hundreds of low-cost commodity FPGA boards interconnected using BlueLink, a lightweight modular interconnection library.

One alternative is the use of wireless networks [89]. These allow direct communication between different compute nodes in a cluster without routing through other FPGAs or routers. However, large data centers require very large bandwidths, which is why they rely on hardwired connections. Knodel et al. [72, 90] presented a scalable cluster infrastructure consisting of multiple hybrid CPU/FPGA compute nodes. They employ a high-bandwidth inter-FPGA link using multiple Gbit transceivers (100 Gbit) in addition to a global inter compute node link via Ethernet (40 Gbit). External connections can always add additional points of failure. Therefore, Fox et al. [91] proposed an error detection and correction method.

Which topology suits which communication type strongly depends on the application and its constraints. These constraints can be cost, size, flexibility, scalability, robustness, or power consumption. Therefore, a widely used approach is to develop a custom FPGA cluster based on an application. Buscemi et al. [92] presented a design for a digital wireless channel emulator based on a cluster of 64 FPGAs. Kono et al. [93] presented a tightly coupled FPGA cluster for Lattice Boltzmann computation.

Heterogeneous computing architectures consisting of different compute devices often have a problem with a direct communication between them if they are from different vendors. Various vendors offer driver support for direct communication with their compute devices over PCI-e, for example using DirectGMA from AMD or GPUdirect from NVIDIA. Communication between compute devices from different vendors often has to be done via the host CPU due to incompatible drivers. It is obvious that this is inefficient, leads to an increased latency and further limits the bandwidth. Communication between two compute devices is either done via a shared PCI-e bus or a vendor specific direct physical connection, e.g., NVSwitch and NVLINK from NVIDIA. On the FPGA side, a customized IP (Intellectual Property)-core including drivers enable a direct connection to a GPU. Table 2.2 compares two different approaches in which direct communication between GPUs and FPGAs is enabled via PCI-e bus. Among other things, the table compares the resource consumption of the PCI-e IP-cores and their achieved bandwidth. The method from Bittner et al., where the GPU is master, achieves better performance, but requires more resources.

Table 2.2: Two approaches for direct GPU-FPGA communication over PCI-e.

	Thoma 2013 [94]	Bittner 2012 [95]
GPU	NVIDIA 8400 GS	NVIDIA 580 GTX
FPGA	XILINX ML506	XILINX ML605
DMA master	FPGA	GPU
GPU driver	Gdev & Nouveau	NVIDIA
Bandwidth (FPGA-to-GPU) [$\mathrm{MB\,s^{-1}}$]	203	500
Bandwidth (GPU-to-FPGA) [$\mathrm{MB\,s^{-1}}$]	189	1800
FPGA LUT	6234	7839
FPGA FF	4090	6965
FPGA BRAM	8	29

2.2.3 FPGA-based Clusters

This subsection presents five modern FPGA-based clusters shown in Table 2.3. It shows the types of CPUs and FPGAs each compute node in a cluster uses. It shows how the inter-node communication between CPU and CPU or FPGA and FPGA takes place. The communication between CPU and FPGA within a compute node is usually done on-chip or via PCI-e.

Bai et al. [77] proposed a low-cost cluster using 48 XILINX Zynq-7020 ARM-FPGAs-based SoCs. They use the ARM for node-to-node communication via MPI (Message Passing Interface), while using the FPGA for more complex computations. They connect the compute nodes

Table 2.3: Comparison of five FPGA-based cluster. ETH (Ethernet)

	Bai 2017 [77]	Hernandez 2018 [6]	Caulfield 2016 [76]	Takano 2019 [79]	Ueno 2019 [78]
CPU	1x ARM Cortex-A9	1x ARM Cortex-A9	2x Intel Haswell	1x Intel Core-i7 4770	1x Intel Xeon 5122
FPGA	1x XILINX Artix-7	1x Intel Cyclone-V	1x Intel Stratix-V	1x XILINX Kintex-7	4x Intel Arria-10
CPU-CPU	48-port Gbit ETH	2x4-port Fast ETH	40 Gbit QSFP+ (shared)	ETH HUB 1000 BASE-T	16-port 100 Gbit InfiniBand
CPU-FPGA	yes	yes	PCI-e Gen3x8	PCI-e Gen2x4	PCI-e Gen3x8
FPGA-FPGA	no	no	40 Gbit QSFP+ ring (shared)	4 Gbit coaxial ring	40 Gbit QSFP+ torus
Compute node	48 SoC	4 SoC	-	4	8

via Gbit Ethernet switch to form a local area network. They evaluated the cluster with an asymmetric encryption algorithm, called RSA (RivestShamirAdleman).

Hernandez et al. [6] proposed a low-cost cluster of Cyclone-V FPGAs, embedded in DE1-SoC boards, and connected to an ARM dual-core. The use two fast Ethernet routers to connect the ARM CPUs with each other. They developed an OS based on a Debian 8, which runs on the ARM CPU, for a fast network communication and to provide OpenCL support. Several benchmarks have been parallelized for the cluster and a workstation, to compare execution time and energy consumption. The workstation has two Intel Xeon E5-2695 v3, which have 14 cores and 28 threads each. The results show the efficiency of the cluster, which reduces energy consumption by up to 83 % while maintaining a similar performance to the workstation. In addition, the cluster is more than five times cheaper than the workstation, which is an advantage of using low-cost FPGAs.

Caulfield et al. [76] describe an FPGA-based acceleration architecture for data centers that is both scalable and flexible. By using FPGAs as I/O, between the server network card and between the local switch, the FPGA can serve as a network device and as a local compute device. The FPGAs can communicate directly with each other over the network without going through a CPU. They developed a reliable inter-FPGA communication protocol that achieves comparable latency to the previous SoA while scaling up to hundred thousand nodes. The evaluated the architecture in multiple scenarios: to accelerate the Bing web search, as a local network acceleration engine, and as a remote web search acceleration service.

Takano et al. [79] presented a cluster consisting of four compute nodes connected via Ethernet through a hub. Each compute node has an Intel Core-i7 CPU and a XILINX Kintex KC705 FPGA connected via PCI-e. In addition, the FPGAs have a bidirectional connection to each other via coaxial cables (4 Gbit) in a ring topology. Each FPGA has an internal bus that includes a router for FPGA-to-FPGA communication, a PCI-e DMA for CPU-to-FPGA communication, a bus DMA, a connection to the DDR (Double Data Rate) memory, control registers, a hardware ICAP (Internal Configuration Access Port) [96] and the user modules. The hardware ICAP, which was not used in their work, is capable of performing DPR for the user modules. To distribute the application, they used the Spark system [80], which they extended for FPGAs using Python. They implemented a JPEG encoder using HLS for the FPGA to evaluate the system. The overall execution time was shortest for a CPU-to-FPGA ratio of 9 to 1.

Ueno et al. [78] presented a tightly coupled FPGA cluster. The host system consists of eight (Intel Xeon 5122 x 2) CPUs connected via a 16-port 100 Gbit fat tree network switch using InfiniBand cables. For their host-server network, they used the work of Sheng et al. [97], who presented a communication infrastructure that supports both online and offline routing. Each host is connected to up to four FPGAs via Gen3x8 PCI-e. In addition, the FPGAs are directly connected to each other in a 2D torus topology, which makes the system very scalable. Each FPGA has a common hardware-based system for efficient inter-FPGA communication. The system consists of a router, a flow controller [98], a serial transceiver, and a remote DMA controller. The remote DMA controller reads and writes from and to local memory(s) on any FPGA controlled by the servers using MPI. The router is connected to four bidirectional communication ports, adds header information to the message, and uses a X-Y routing algorithm. The measured payload bandwidth for a single FPGA-to-FPGA link is 32.2 $\text{Gbit}\,\text{s}^{-1}$. Whereas the FPGA-to-host link reaches 7.88 $\text{GB}\,\text{s}^{-1}$ and the host-to-host link reaches 12.5 $\text{GB}\,\text{s}^{-1}$. The results show that the FPGA network is superior to the host network in terms of communication delay between any individual nodes. On the other hand, the host network is more effective for collective communication where performance is highly dependent on the bandwidth. When only the FPGA or the host network is used, an input to the destination server becomes a bottleneck because there is no alternative route.

2.2.4 Summary

This part summarizes the knowledge gained from the SoA and relates it to this thesis. It introduces a common terminology that combines the embedded system and HPC domains to bridge the gap between them. This thesis focuses on the computing and communication areas, where topology also has a major impact. It uses a distributed (or hybrid) memory model, without going too much into detail about the memory architecture. Thereby, it will use dataflow models and treat nodes that use a shared memory model as single unit. The reduction in complexity has a significant impact on the application distribution process.

The system represents the top level, which comprises the entire architecture including memory, computing, communication, topology, and I/Os. Thereby, a distinction can be made between heterogeneous and homogeneous systems. The concepts and methods developed in this thesis focus on heterogeneous FPGA-based systems. Within this context, it mainly considers MPSoCs in the embedded system domain and compute clusters in the HPC domain.

Figure 2.8 shows a common computing model for the two domains, which is derived from the OpenCL terminology. A later section will discuss the OpenCL API in more detail. A compute cluster consists of 1 to n compute nodes. These can range from a classical PC (Personal Computer) in the HPC domain to an energy efficient SoC in the embedded system domain.

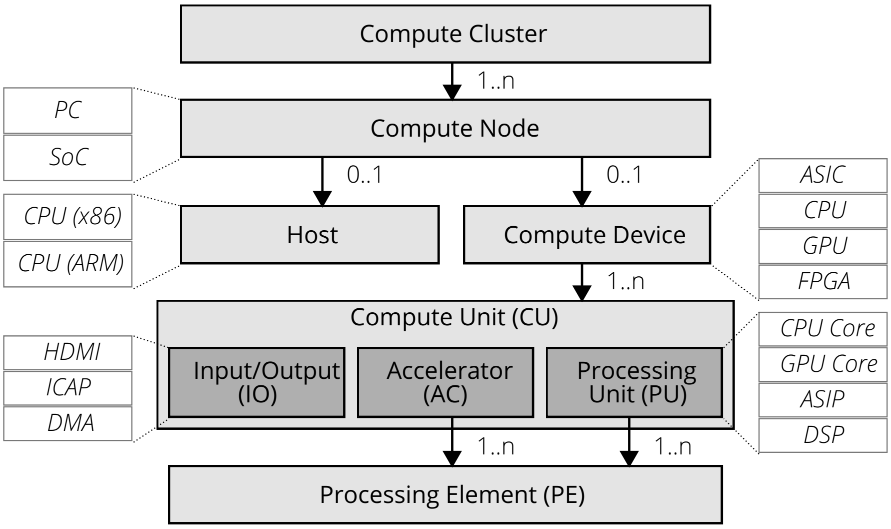


Figure 2.8: A combined computing model for embedded systems and HPC computing using the OpenCL terminology.

A compute node consists of 0 to 1 hosts and 0 to n compute devices. The host, which is not always necessary in decentralized systems, is typically an ARM CPU in embedded or an x86 CPU in HPC systems. In the embedded domain, host and compute devices are usually on one SoC, which connects them via AMBA (Advanced Microcontroller Bus Architecture) interconnect on ARM-based systems. In the HPC domain, a compute node connects host and various compute devices via PCI-e bus. However, direct communication between compute devices from different vendors via PCI-e requires special drivers and IP-cores.

Loosely coupled compute clusters usually connect the various hosts via Gbit Ethernet or InfiniBand through a switch. Tightly coupled compute clusters additionally connect all compute devices in the cluster with each other. For example, via Gbit transceivers on XILINX FPGAs or via NVLINK on NVIDIA GPUs. The topology chosen for a tightly coupled interconnection network is mostly a ring. Depending on the size and scalability, line, mesh, and torus topologies can also be used. Which topology and which communication type is used strongly depends on the application and its constraints. These constraints can be cost, size, flexibility, scalability, robustness, or power consumption.

A compute device can be a GPU, an FPGA, a multicore CPU or an ASIC. An embedded system could have an integrated MPSoC implemented on an FPGA or ASIC compute device. Examples of ARM-based SoCs are the Tegra from NVIDIA [99] or the Zynq from XILINX [100].

A compute device consists of 1 to n CUs (Compute Units). This thesis distinguishes between three types of CUs, which are I/Os, ACs and PUs. The difference between ACs and PUs is that ACs cannot act independently. In a multicore CPU, the CU would be a single core, referred as PU. On an MPSoC, the CU can be an AC, I/O or PU. On an FPGA-based MPSoC, this I/O would be the IP-core needed for interfacing to the physical I/O. The different CUs can be directly connected to each other. For example, via ring bus in CPUs or NoC in MPSoCs. For GPUs, the connection between CUs within the same compute device usually does not exist.

The smallest units in the computing area considered in this thesis are the PEs. A CU can have 1 to n PEs. Multiple PEs within a CU can be processed in SIMD manner, which enables data-level parallelism.

Using a direct network topology is a useful approach for both off-chip communication between compute devices and hosts, and on-chip communication between CUs, if a scalable system needs to be set up using a NoC. In a direct network topology, each node has its own router. The off-chip communication can also be wireless. This allows a direct communication between different compute devices without routing via other FPGAs or routers. However, many systems prefer wired connections due to their reliability and available bandwidth.

2.3 Application Distribution

One challenge in the design or deployment of FPGA-based systems is the distribution of application(s), which strongly depends on the communication model and the topology of the overall system. A good distribution of an application reduces the influence of communication overhead between the different compute devices or CUs. However, most of FPGA-based systems leave this task to the developer. While tools and libraries such as Simulink and SystemC are suitable for system-level modeling and simulation, automatic mapping to heterogeneous and distributed systems has proven to be a very difficult task [101]. Due to the increasing number of nodes in these systems, it needs a good algorithm to place communicating nodes close to each other [63].

The following subsections deal with the different steps that are necessary to create an application specific FPGA-based system. Since this is a very complex problem and no one has dealt with all steps, this thesis examines the problem for different types of FPGA-based systems and from different perspectives. Due to the high complexity, application distribution algorithms use heuristics, metaheuristics, or even neural networks to solve these problems in a reasonable amount of time. Therefore, the first subsection discusses metaheuristics.

The second subsection deals with the mapping and scheduling problem in NoC-based MPSoCs and heterogeneous systems. It examines the classical mapping problem, which is also relevant in non-FPGA-based systems. However, the architectures in these systems are mostly fixed and do not use the full potential of an FPGA. The third subsection examines DSE in FPGA-based clusters for neural networks. Neural networks are a computationally intensive use case, and DSE is part of all steps in the application distribution process. However, by being tied to the use case, both the toolchain and the resulting system lose generality.

The fourth section examines the clustering and placement problem for CLBs (Configurable Logic Blocks) in CAD tools. The fifth subsection examines the tuning, clustering, and placement problem for PRRs. Both subsections deal with mesh-like structures and therefore have similarities. The last subsection summarizes the collected knowledge from the different perspectives and systems and related them to this thesis. Based on this, it provides a generic definition for the different application distribution steps, including dimensionality, constraints, and objectives.

2.3.1 Metaheuristics

Many real-world optimization problems involve different complexities such as non-convexity, nonlinearities, discontinuities, and large dimensionality, making many mathematically provable algorithms ineffective or inapplicable [102]. There are no known mathematical models to find an optimal solution to all these problems in a limited computation time [102]. An example is the solution of the QAP (Quadratic Assignment Problem), where even for medium sized problems of > 20 it is not possible to compute all possible combinations to find an optimal solution. The first part of this subsection discusses the QAP as it describes the placement problem in FPGA CAD tools. The following parts discuss the characteristics and classification of metaheuristics. The last part describes four different metaheuristics used in the SoA or in this thesis.

Quadratic Assignment Problem

The QAP addresses the problem of allocating a set of facilities to a set of locations, with the cost being a function of the distance and flow between the facilities, plus costs associated with a facility being placed at a certain location. The objective is to assign each facility to a location such that the total cost is minimized. The problem can be described with three $n \times n$ input matrices ($F = (f_{i,j})$, $D = (d_{k,l})$, $B = (b_{i,k})$). The first matrix describes the flow between facility i and facility j. The second matrix describes the distance between location k and location l. The third matrix describes the costs of placing facility i at location k. The QAP is NP-hard and even finding an approximate solution within some constant factor from the optimal solution cannot be done in polynomial time unless P=NP. [103]

Characteristics

In many optimization problems it is not necessary to find the best solution. In these cases, it is sufficient to find the best possible solution in the available time. Therefore, algorithms using heuristics are applied to optimization problems where it is not possible to find a solution in a reasonable amount of time. Many heuristics are adapted to the problem that must be solved and often use greedy-based algorithms. They (1) look for an approximate solution, (2) do not particularly need a mathematical convergence proof and (3) do not explore each possible solution in search space [102].

Metaheuristics, on the other hand, are problem-independent and can therefore better search the solution space to avoid getting stuck in a local optimum. While they do not guarantee finding an exact optimal solution, they can lead to a near-optimal solution in a computationally efficient manner [102]. The solution space describes the set of all possible solutions. Metaheuristics are often used to solve NP-hard problems. They are popular for solving combinatorial optimization problems because few classical methods can manage the type of variables they involve. Examples include knapsack problems, bin-packing, network design, traveling salesman, vehicle routing, facility location and scheduling [102]. Metaheuristics inherit the heuristic properties to solve search and optimization problems [102]. They (1) seek to find a near-optimal solution, (2) usually do not have a rigorous proof of convergence to the optimal solution and (3) usually are computationally faster than an exhaustive search [102].

To do this, they operate on a representation or encoding of a solution that can be easily manipulated. The algorithm then performs a series of operations to iteratively converge to the best possible solution. To this end, they evaluate potential solutions and perform a series of operations on them to find a better solution in solution space. For this purpose, the objectives of the optimization problem must be clearly defined. Metaheuristics are inherently an iterative process.

- They start with an initial solution that is randomly generated in most cases.
- They need a search function to find new solutions in solution space.
- They need an evaluation function that rates solutions based on the objectives and constraints.
- They need a termination criterion that is also based on the objectives and constraints.

When multiple objectives are involved, there is usually no single solution that represents the optimum for all objectives. Rather, a variety of optimal solutions are possible, which represent a compromise between the different objectives. These solutions are known as the Pareto-optimal solutions. For multi-objective optimization problems with conflicting objectives where there are multiple optimal solutions, population-based metaheuristics are often preferred so that the entire Pareto-optimal front can be represented simultaneously. [102]

Good metaheuristics balance between exploitation and exploration to search for new solutions. Exploration uses a global search behavior to avoid getting stuck in a local optimum. Exploitation uses a local search behavior that can find the nearest optimum.

Classification

Many metaheuristics are of a stochastic nature, mimicking a natural, physical, or biological principle that resembles a search or optimization process. Figure 2.9 shows a common taxonomy for metaheuristics and lists a few of the most popular algorithms as examples. Single-solution metaheuristics, on the one hand, start with an initial solution, which they iteratively modify. Population-based metaheuristics, on the other hand, start with more than one initial solution. In each iteration, they modify multiple solutions, and some of them make it to the next iteration. Using multiple initial solutions allows single-solution metaheuristics to be transformed into population-based ones. Population-based metaheuristics can be further divided into two categories. Evolutionary algorithms mimic various aspects of evolution in nature, such as survival of the fittest, reproduction and genetic mutation. Swarm intelligence algorithms mimic the group behavior or interactions of living organisms, such as ants, bees, birds or fish, and non-living things, such as water drops or river systems. [102]

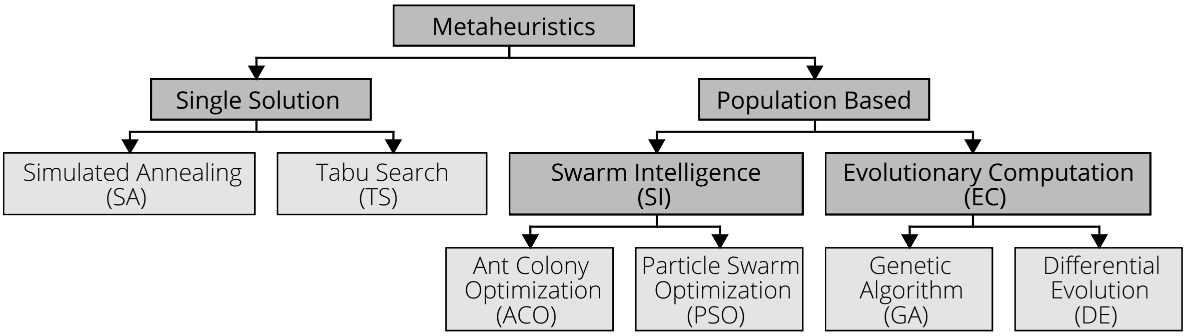


Figure 2.9: Classification of metaheuristics including the most known examples.

However, there are also other classifications that can be made. Deterministic methods follow a trajectory from an initial solution. Whereas stochastic methods allow probabilistic jumps from a current solution to the next. Greedy methods tend to search in the neighborhood of the current solution and move on to a better solution when it is found. Whereas non-greedy methods either wait for some iterations before updating a solution or have a mechanism to backtrack from a local optimum. Memory-based methods maintain a record of past solutions and their trajectories. Some metaheuristics allow only a limited number of moves from the current solution within a single neighborhood (e.g. SA or TS). Whereas others use operators and parameters to allow for multiple neighborhoods (e.g. PSO (Particle Swarm Optimization)). Unlike static objective functions, dynamic objective functions can be updated depending on the current search requirements. Most metaheuristics are population-based, stochastic, non-greedy and use a static objective function. [102]

Algorithms

The SA process like to the cooling process of molten metals by a structured annealing process [102]. At a high temperature, the atoms in the molten metal can move freely against each other. The movement of the atoms is restricted when the temperature is reduced. The atoms begin to arrange themselves and eventually form crystals with the lowest possible energy. This process is simulated with the Metropolis algorithm [104]. In its original form, it is a metaheuristic based on single solutions. It allows that new solutions can also become worse. However, the value for deterioration also decreases from time to time. This allows the simulated physical system to break out of local optima. The decrease of the temperature also leads to a termination of the algorithm.

The TS algorithm searches the entire neighborhood from a correct solution to find a new solution [105]. A neighboring solution is a solution that can be reached with an elementary step. The algorithm chooses the best solution from all neighboring solutions, even if it is worse than the current one. A problem that arises from this approach is that it can easily lead to cycles. For example, when reaching a local optimum, the search would always return back to it. The solution of the algorithm is to save already visited solutions and to forbid to visit them again. Therefore, it stores intermediate solutions in a tabu list. Mostly it does not store the whole solution but features or the modifications that led to this solution. However, the tabu-lists can also lead to the inability to find solutions that have not yet been visited [105]. The aspiration criterion can override the taboo criterion. This allows the algorithm to revisit a previously visited solution. The tabu-list, also named as memory or history can have different types of memories, such as STM (Short-Term Memory), MTM (Medium-Term Memory) or LTM (Long-Term Memory). The definition and way of using the different memories differs in the literature. TS is efficient for solving combinatorial optimization problems with discrete search spaces and vehicle routing [102].

GAs (Genetic Algorithms) are among the most popular metaheuristic techniques [102]. The solutions are represented by binary bit strings that are iteratively modified through recombination and random bit flips. This approach is similar to the evolution of genes in nature, which also undergo natural selection, genetic crossovers, and mutations. The algorithm begins with an initial population that it randomly generates [106]. From this initial population, it performs a series of genetic operations such as selection, recombination, and mutation to generate a new population in an iterative manner. The first step of this iterative process is to evaluate the individuals using an objective function that calculates the fitness value. Then, the algorithm selects the individuals with the highest fitness value for reproduction. It generates new individuals from the selected individuals through various operations, such as genetic crossover and mutation.

In the ACO (Ant Colony Optimization), ants often follow each other along a specific path between their nest and a food source [102]. When ants walk, they leave a trail of a chemical substance called pheromone. This pheromone has the property of disappearing over time. If there is no further deposition of pheromone on the same pheromone trail, the ants cannot follow each other. In the beginning, when the ants do not have a specific food source, they are wandering almost randomly in search for such a source. The stronger the pheromone trail is, the higher the number of ants that follow it. When a group of ants finds a food source, repeatedly carrying small amounts of food back to the nest, it increases the pheromone content of that trail, attracting more and more ants to follow it. For example, if two different ants can reach a food source via two pheromone trails of different lengths, the pheromone content on the shorter trail will be increasingly strengthened. The fact that pheromones strengthen shorter trails allows the entire ant colony to detect a possible minimal trail. Therefore, traveling salesman problem commonly uses the ACO algorithm.

2.3.2 Mapping and Scheduling on Many-Core Systems

Singh et al. [107] presented a survey for mapping applications to many-core and multicore systems. As shown in Figure 2.10, they divide mapping algorithms into two groups (design time and runtime). They further divide these into two subgroups (heterogeneous and homogeneous systems). While runtime mapping methods can manage dynamic workload scenarios, their computational overhead must be kept low. Whereas design time methods can produce better quality results if the application parameters are known. The mapping algorithms can have different objectives: (1) performance (2) resource usage (3) energy consumption (4) temperature distribution and (5) system reliability. In addition, they can include other tasks such as adding synchronization, communication between tasks, memory management, or ensuring the correct functionality [107]. They require a scheduler to ensure the execution order of tasks and to provide synchronization between them.

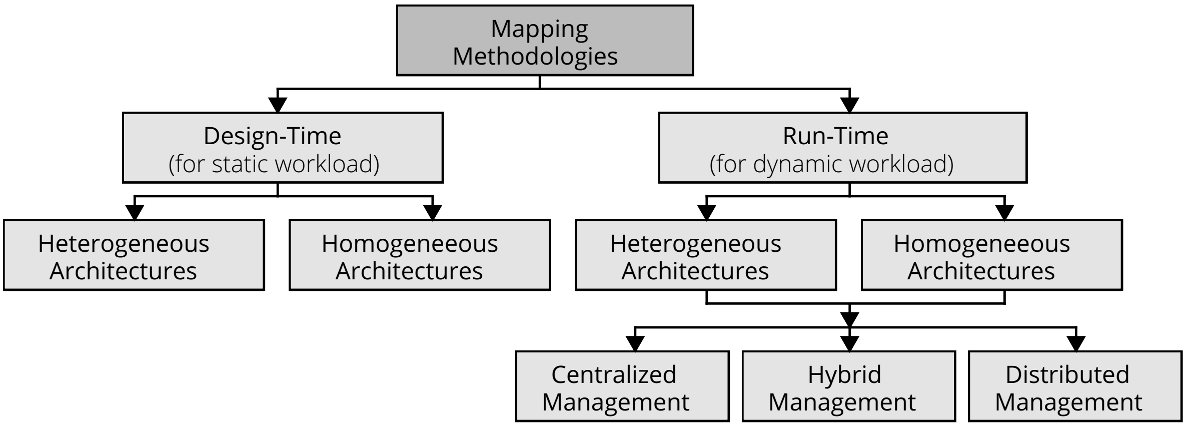

Mapping Methodologies
Design-Time (for static workload)
Run-Time (for dynamic workload)
Heterogeneous Architectures
Homogeneeous Architectures
Heterogeneous Architectures
Homogeneeous Architectures
Centralized Management
Hybrid Management
Distributed Management

Figure 2.10: A taxonomy of mapping methodologies [107].

Mapping and Scheduling of Task Graphs on Heterogeneous Systems

A topic of this thesis is the scheduling and mapping problem of DAGs (Directed Acyclic Graphs) on heterogeneous FPGA-based systems at design time. The considered problem is known to be NP-hard [108] and has a strong impact on system performance. Which is why many researchers use heuristics. One interesting heuristic involves the duplication of tasks. They schedule some tasks redundantly to reduce the overhead of inter-process communication. They usually differ only in the selection strategy of the tasks that are duplicated.

Goens et al. [109] present a runtime system for static mapping of multiple applications on heterogeneous architectures. However, they do not cover DAG-based applications and cases targeting multiple devices. Grew et al. [110] distribute OpenCL programs to multiple devices based on the set and properties of operations used using a machine learning model. The model can be easily ported to different applications and devices. Unfortunately, this approach only considers applications written in a singular kernel. This makes it useless for scheduling DAG-based programs that consist of multiple kernels. Van Craeynest et al. [111] observe the performance impact of tasks on heterogeneous devices, in particular cycles per instruction and memory/instruction level parallelism. The approach uses runtime estimates of tasks on a given core or device to predict the performance of other cores or devices for the same tasks.

In a comparative study, Canon et al. [112] compare several task mapping and scheduling algorithms in terms of their robustness and performance [112]. One heuristic that stood out is the HEFT (Heterogeneous Earliest Finish Time) [113] algorithm. The algorithm aims at an efficient and static scheduling of tasks of an application, which can be represented as a DAG, on the available resources of a heterogeneous system. It annotates the nodes (tasks) with the estimated computation time and the edges (data dependencies) show the estimated communication time between the tasks. The goal is to map tasks to processors and order their executions to meet task precedence requirements and achieve a minimum total computation time.

Mapping and Scheduling on NoC-based MPSoCs

For NoC-based MPSoCs, mapping is the step where individual system processes can be optimally assigned to NoC entry points (routers), to reduce traffic and energy consumption in the network [114]. One advantage of NoC-based MPSoCs is the easier way of mapping multiple applications to it. Another advantage of NoC-based MPSoCs is that applications can easily be mapped at runtime [115, 61]. Using FPGAs-based techniques such as DPR, not only virtual tasks can be mapped at runtime, but also physical nodes (CU) can be placed at runtime.

Many researchers aim in improving the design or mapping process of NoC-based MPSoCs using rapid prototyping techniques and DSE [116, 101, 117, 118]. DSE refers to the investigation of design alternatives to the implementation. Zhu et al. [118] present a platform-involved design flow for multi-application mapping onto tiled NoCs, by integrating various single-application heuristics. Mand et al. [116] map artificial neural networks to a NoC-based MPSoC using a scalable and extensible methodology for rapid prototyping of complex applications. They create the MPSoC with a NoC generation tool that describes the system using an XML (Extensible Markup Language) configuration file. The purpose of the experiments was to find the best architecture to emulate artificial neural networks on FPGAs. One conclusion for their system was that on-chip memory is a main bottleneck. Robino et al. [101] describe a system-level design methodology that starts from a Simulink model and generates a NoC-based MPSoC. The hardware and software description files are generated using rapid prototyping, and the generated MPSoC gives the same results as in simulation. They use the HeartBeat model, an application implementation concept for NoC-based MPSoCs that uses the synchronous computation model [119]. Their design flow reduces the design time to create NoC-based MPSoCs and enables fast DSE. Sultan et al. [117] proposed a development environment for rapid prototyping on NoC-based MPSoCs to improve DSE. They convert a high-level dataflow model into multiple executable C codes that are executed on an emulation platform and collect runtime performance traces. By using the proposed method, developers can quickly design NoC-based platforms and accurately evaluate mapping algorithms.

For many researchers, the main objective for mapping in NoC-based MPSoCs is the communication infrastructure. For example, they aim to reduce the number of hops in NoCs to reduce the energy consumption of the communication infrastructure. The number of hops is defined as the number of routers that a message must pass between two communicating tasks. Ost et al. [68] present runtime task mapping on CUs in heterogeneous NoC-based MPSoCs using predefined constraints. They use a uniform model-based approach to separate application, mapping and the platform. They use static and dynamic mapping heuristics like LEC-DN [120]. LEC-DN uses two cost functions for the number of hops and the communication volume between tasks. Bayar et al. [63] use simple switches instead of routers to set communication paths between nodes on NoC-based architectures. They do task mapping using PFMAP and communication routing according to the application. PFMAP is inspired by a systematic resampling algorithm for particle filters [121]. They generate the topology and reconfigurable switches based on the routing information and load (start) the tasks. Pang et al. [122] give a good overview of task mapping algorithms. They explore different mapping algorithms for different shaped mesh topologies for FPGA-based NoCs. The analyzed results are the applications execution time, energy consumption and mapping time. Reddy et al. [123] proposed an energy efficient mapping algorithm for NoCs and show a good comparison to other algorithms in terms of energy efficiency.

2.3.3 Design Space Exploration for Neural Networks on FPGA-based Cluster

A very computationally intensive application area is machine learning. Therefore, many optimized FPGA implementations exist in the area of machine learning [84, 85, 124, 87]. As the complexity of machine learning algorithms, such as ResNet [125], increases, the resources of a single FPGA may not be sufficient to efficiently implement these complex algorithms. Therefore, clustering multiple FPGAs is necessary to manage Big Data applications. However, the growing size and complexity of neural networks, coupled with communication and off-chip memory bottlenecks, make it increasingly difficult for multi-FPGA designs to achieve high resource utilization and performance, especially for the backward propagation [126, 127].

Depending on the application and its distribution on the compute cluster, different communication scenarios can occur [84]. One case would be the transfer of data from one CU to multiple other CUs (`scatter`). Another case would be that intermediate results generated by one CU need to be transferred to another CU (`point-to-point`). The last case would be that partial results need to be merged and written back to the application software (`gather`). Therefore, when distributing CUs across different FPGA-based compute devices, the bandwidth and latency between them must be considered.

Zhang et al. [87] proposed an implementation of the AlexNet CNN (Convolutional Neural Network) [128] on a pipeline of six FPGAs connected in a ring topology using Aurora serial interconnects. They highlight the key advantages of an inter-FPGA network with low overhead and high bandwidth. They reduced the complexity of the solution space to $\binom{N-1}{K-1}$ for N layers and K FPGAs, due to their restricted topology and application area. They propose a polynomial-time DSE solution using dynamic programming. The algorithm reduces the problem of DSE from multiple FPGAs to that of a single FPGA. The problem of a single FPGA was addressed in a previous work, in which they identified all possible solutions in the design space using a roofline model, under the constraints of resources and memory bandwidth [124].

Geng et al. [126, 127] present FPDeep, a framework for CNNs. Unlike other work, they focus not only on the forward pass, but also on the backpropagation (training). For acceleration they use a pure FPGA cluster, which is arranged in a line topology. They use different strategies for the distribution of the neural network over the FPGA cluster and an efficient resource utilization. FPDeep uses a deep pipeline to map the CNN to multiple FPGAs. It provides fine-grained inter- and intra-layer partitioning to improve workload distribution between FPGAs. It uses fine-grained pipelining to minimize the activations while waiting for back-propagation. It uses weight load balancing to balance weight allocation between FPGAs to reduce the required number of compute devices. Experimental results show a good scalability with a large number of FPGAs, while the bandwidth between them remains the limiting factor. Using six transceivers per FPGA, they can achieve linearity for up to 83 FPGAs. Using XILINX VC709 FPGAs they achieve an 8.8 times better energy efficiency than with a Titan X GPU system.

Owaida et al. [84, 85] explored and developed techniques to map resource-intensive machine learning applications, namely inference over decision tree ensembles, on an FPGA cluster. Inference is a resource-intensive operation and behaves differently depending on size and data dimensionality. For large scales it becomes compute-bound, while with increasing data dimensionality it becomes memory- and network-bound. The proposed cluster compute consists of 20 Microsoft Catapult CPU/FPGA compute nodes [76]. The CPUs are interconnected via a $10\,\mathrm{Gbit\,s^{-1}}$ network switch. In addition, the FPGAs are interconnected with each other in a ring topology. They developed a light-weighted inter-FPGA communication protocol and routing layer to simplify communication between the different FPGAs. They investigated different strategies for data partitioning and the distribution of the machine learning application, thereby maximizing the performance. They investigated how to efficiently distribute applications across the mentioned compute cluster. They demonstrate the importance of decoupling I/O from the computational part in the algorithm and developing reusable hardware and software I/O management components to productively map complex operations on an FPGA cluster. Their empirical evaluation demonstrates the efficient use of resources in an FPGA cluster while achieving linear scalability. For large tree ensembles the system becomes computation bound, while with increasing data dimensionality it becomes memory and network bound.

2.3.4 Clustering and Placement for Configurable Logic Blocks

Many different researchers looked at partitioning, clustering, placement, and routing problems for CAD tools to efficiently map a netlist to the resources (LUTs, FFs, etc.) of a single FPGA [129, 130, 131, 132, 133, 134]. Efficient algorithms in CAD tools are important to reduce the performance differences between FPGAs and ASICs [131]. The CAD toolflow can be broadly divided into five distinct steps, namely synthesis, technology mapping, clustering, placement, and routing, as shown in Figure 2.11 [129]. Different objectives have been addressed in the various publications to reduce performance, energy consumption or area (resource utilization). For example, they attempt to reduce the critical path to increase performance. Bouzaziz et al. [129] use clustering methods to improve power consumption, area, critical path delay and energy. Using a first-choice algorithm, they can achieve 17 %, 11 %, 13 %, and 24 % improvement for the mentioned four objectives compared to using the T-VPack algorithm [135]. For CAD tools, clustering is the process of partitioning a mapped netlist of logic blocks into a netlist of clusters mapped directly into logic elements on the FPGA architecture [129]. Other objectives focus on the reduction of the execution time needed in CAD tools for computationally intensive processes, like clustering, placement, and routing.

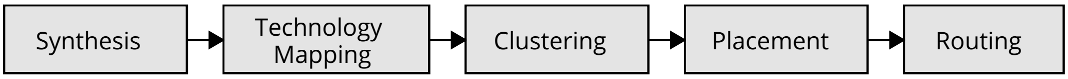


Figure 2.11: The flow of FPGA-based CAD tools.

Many researchers describe the placement process as a QAP in which each node of a task graph is mapped to a node in a resource graph [133]. In addition, the distance (hop count) should be considered in the resource graph and the communication data in the task graph. Due to the complexity, it is difficult to find an optimal solution for this type of problem in a short time. Therefore, heuristics or metaheuristics can be used. Several metaheuristics have been explored for the placement problem in reconfigurable systems, such as GA, TS, ACO, and SA [132, 133, 134]. An observed advantage of SA was that it finds good results in a reasonable amount of time [134].

Mohtavipour et al. [132] use quadratic subregions to divide a task graph into different independent parts. Then, each part is placed into a predefined quadratic region in a resource graph. To reduce the computational complexity of the placement problem, there is a preprocessing to simplify the graph by using link elimination.

Mohtavipour et al. [133] also proposed a distributed task graph placement algorithm to analytically reduce the intensive computations of the problem. They introduce a distance model for the resource graph to remove heavy weighted connections from search space. For this purpose, they analyze forbidden regions in the placement matrix. They split the task graph into several parts (clusters) to place each part into a lighter weighted region. They were able to reduce the number of hops between CLBs by 9.3 % to 27.25 % compared to a uniform and normal distribution. They evaluated their approach on four different real-world applications that consume between 32 and 77 CLBs, which is a rather small size compared to the total amount of CLBs in modern FPGAs, which can reach over one million.

Mohtavipour et al. [134] presented another approach that includes the placement and clustering process. They use graph convolutional networks to partition the task graph into the least dependent parts. They compute the final placement using inter-cluster and intra-cluster optimizations. They perform the intra-cluster placement by using a conventional SA algorithm and the inter-cluster optimization by estimating the average distance. They evaluated their approach using three applications that contain 1550 to 1955 CLBs. They could improve the communication costs between 17 % and 26 % in comparison to a uniform and normal distribution.

All three publications by Mohtavipour et al. [132, 133, 134] use clustering and placement methods to map a task graph to a resource graph containing CLBs with the goal of decreasing the hop count. In contrast to this thesis, the first two publications use the partitioning term for the clustering process. All three publications target a rather small CLB size (32 - 1955) compared to the available resources of a modern FPGAs. Separating inter-cluster and intra-cluster optimizations reduces complexity, but carries some loss of optimality for the final solution. It would have been interesting to see some timing measurements of the proposed methods to also evaluate complexity and scalability.

2.3.5 Tuning, Clustering and Placement for Partial Reconfigurable Regions

The use of FPGAs in the HPC domain, such as cloud computing and data centers, creates opportunities for novel mapping and resource allocation strategies to increase system throughput [136]. This affects, for example, strategies involving spatial partitioning of resources, since vendor tools often treat the FPGA as a sea of gates [137]. Two different ways of how to place multiple applications on an FPGA are temporal (sequential) or spatial (parallel) [136]. While the former provides a high throughput, PRRs can be used to run multiple applications simultaneously, but also in a time-shifted manner [137]. For spatial partitioning of resources between independent applications, frameworks that use PRRs support partitioning of FPGA resources into fixed rectangular regions [136]. These regions are often homogeneous and allow a set of offline mapped applications to be placed on tile-based FPGAs [137].

Placing heterogeneous and independent applications with variable requirements, such as memory, compute power, resources, or bandwidth, onto PRRs can result in lower compute density and thus low utilization of FPGA resources by up to 40 % [137]. Without an intelligent placement strategy, spatial partitioning of the FPGAs using PRRs can yield equal or worse computational performance than sequential execution [137]. This is partly due to the asymmetric spatial distribution of the various on-chip resources of an FPGA [137]. Together with the placement constraints of static logic, for example for I/Os, and suboptimal placement of heterogeneous applications to homogeneous PRRs, resource utilization can be very inefficient [136]. The problem of inefficient use of PRRs has been addressed by some researchers using intelligent DSE and clustering methods [137].

Minhas et al. [137] developed a methodology using DSE, machine learning based ridge regression, and clustering of heterogeneous independent tasks to select an optimized combination of tasks for resource sharing on FPGAs. They first run DSE to generate multiple hardware bitstreams for each task that provides speedup according to resources and throughput. For DSE they use task-specific parameters such as the block size or the number of rows, in the HLS tool. They then use DSE along with machine learning based ridge regression models to evaluate the relationship between on-chip and off-chip resources and FPGA throughput. They use this to split all tasks into smaller clusters so that the tasks in a cluster share the FPGA resources at a single point in time while maximizing overall resource utilization.

They evaluate their design on a Stratix-V FPGA using the OpenCL SDK (Software Development Kit) from Intel. They implemented eleven independent HPC applications, such as graph analysis, linear algebra, media streaming, and data mining, using HLS. Each application consists of one task. Resource sharing is limited to a cluster of two tasks. Both tasks are merged into one bitstream. For DSE, they create between four and nine different bitstreams per task. For random clustering, they achieve 2.4 times higher system throughput and 1.9 times better energy efficiency with static partitioning compared to using PRRs. In their proposed clustering method, system throughput increases by 1.2 times using PRRs, which is due to a more efficient use of the off-chip memory bandwidth. For static partitioning, the throughput increases by 1.4 times, which is due to the optimization of on-chip and off-chip resources.

The complexity of the clustering methods in the application examples is very low, since only a maximum of two applications per cluster were evaluated for both methods. Furthermore, the work is limited to independent applications with only one task per application. However, combining the applications into one bitstream limits the use of PRRs. Creating multiple bitstreams for each application to get estimates is a very time-consuming process compared to using synthesis results, since it can greatly increase with the number and complexity of applications.

In the past, various DPR techniques have been proposed for PRRs on NoC-based SoCs, which mainly include multiple CUs and efficiently utilize a mesh topology [138]. The development of scalable NoC-based systems also brings other challenges, such as efficient mapping and placement, which are NP-hard problems [114, 138]. The traditional NoC mapping problem is a QAP, defined as mapping a set of tasks to locations on a NoC, where each location contains a CU connected to a router [138]. The objective is mainly to reduce network traffic and thereby energy consumption [138]. Free placement of CUs within the NoC, adds an additional complexity. In this case, mapping and placement cannot be completely separated, because the changing physical constraints must be considered in the mapping process [138].

Rad et al. [139] presented a placement method based using neural networks. First, they use the Node2vec embedding algorithm, dimension reduction, and rotate and scale an initial mapping from the task graph to the resource graph. Second, optimization of dilation (hop count) and maximum capacity utilization was performed using the stochastic gradient descent method and a loss function. Main goal of their work was to reduce the dilation (hop count). It would also be interesting to see computation time of their proposed methods in terms of scalability to an increasing number of FPGAs and CUs.

A few works have addressed the placement problem for PRRs using DPR on NoC-based architectures [114, 138]. In this context, DPR has the potential to lower the cost of digital circuits, increase flexibility, and reduce space requirements [114]. However, DPR increases the complexity of the task mapping and CU placement problem by adding a new layer.

A few researchers have tried to solve the placement problem for NoC-based reconfigurable systems, using GA [140, 114] and TS [138, 141] metaheuristics. They have adapted the DyNoC architecture proposed by Bobda et al. [142] as a base model. It includes a NoC-based mesh topology that is flexible, simple, and based on traditional routers, as shown in Figure 2.12. A core (CU) is always connected to the router in its upper right corner. For larger cores, all routers within the occupied area are disabled. The heterogeneous size of the core area leads to an irregular structure. Additionally, not each router is connected to a core. They simplify temporal partitioning, by quantizing time into scenarios. The evaluated applications, contain three to five scenarios in which the reconfigurable cores can be placed. The smallest scenario places six static and seven dynamic cores to 7×7 allocation slots and three scenarios. The main objectives in both works are the communication bandwidth and the number of hops between routers.

Filho et al. [140, 114] presented a GAs-based solution for the placement of hardware cores to NoC-based partially reconfigurable systems in the design phase. Three GA crossings and two population diversification methods were applied and compared to analyze the best solution. The best GA solution is compared with the best solution obtained with a semi-exhaustive algorithm [140]. The goal is to place the cores in such a way that the total number of hops in

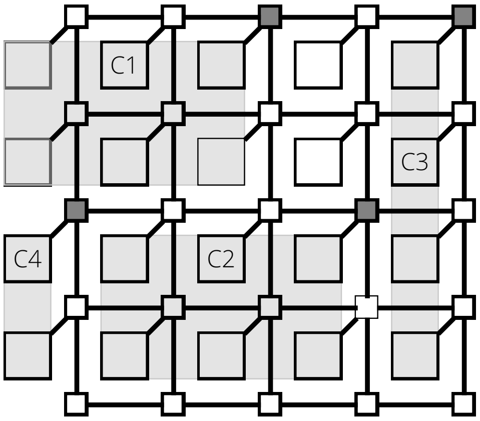


Figure 2.12: Placement example of the DyNoC [142] architecture containing four cores (C1 - C4) placed to 20 allocation slots.

the NoC is minimized for each configuration scenario. The measured execution times were 48, 73 and 113 seconds for applications with 13, 16 and 26 cores on an Intel Core-i7-3770 CPU. The different solutions had an average penalty of 5.4 % compared to the best solution found by the semi-exhaustive algorithm.

Novaes et al. [138, 141] explore TS-based algorithms for the placement problem in NoC-based systems containing PRRs. They compare four combinations of TS algorithms. They use the Robust and Adaptive TS algorithms as base and combine them with the Navigation and Force Inversion heuristics. The robust TS adds a STM called taboo list and the aspiration criteria. The adaptive TS adds backtracking and an adaptive radius to avoid getting stuck in a local minimum. The Navigation heuristic adds a restart mechanism to explore new regions from the search area. The Force Inversion heuristic, which was proposed by the authors, inverts the solution after a random solution is selected, to explore new regions and avoiding local minima. The Force Inversion-Adaptive TS algorithm provides the best results and an average penalty of 2.0 % compared to the best solution found by a semi-exhaustive algorithm. Its execution time is between 1.3 and 6.9 seconds.

The size of their cores is quite coarse-grained due to their tiled architecture, which can lead to high fragmentation. In comparison, this thesis allows more heterogeneous size of these tiles and the placement of multiple cores to one partition, which can reduce the fragmentation problem [29, 14]. Their mapping process is quite simple, as each task is mapped to exactly one core. In comparison, this thesis allows multiple tasks to run sequentially on one core, which is also the purpose of a PU. Their time slots or scenarios for DPR are kept fixed and quite coarse-granular. Therefore, resource consumption is not an objective for them. In comparison, to this thesis their placement process is very time consuming. Compared to this thesis, their placement process is very time-consuming. This thesis does not start with a random mapping and increases the exploration using a multithreaded grid-based approach. Additionally, the SA algorithm increases exploration while reducing memory requirements. Furthermore, this thesis has a more complex memory structure with STM, MTM and LTM elements.

2.3.6 Summary

The previous subsections presented various concepts for application distribution on FPGA-based systems based on the SoA. The systems range from a NoC-based compute device to a FPGA-based compute cluster. Based on the preliminary work, this thesis identified six subareas for the distribution of applications on FPGA-based systems at design time. Based on these subareas, it should be possible to produce an application-specific architecture starting from its application code. In this context, the last two subareas, namely clustering and placement, refer exclusively to reconfigurable hardware.

- **Partitioning** is the process of dividing the original application into tasks. Dependent tasks span the application or task graph that represents the dataflow or control flow of the application. These tasks can be performed on physical nodes, such as compute devices or CUs. In addition, spatial partitioning of FPGA resources into PRRs can be done to span a resource graph.
- **Tuning** is the process that optimizes the application at functional level. This involves improvements to the task graph that also affect dataflow by using task duplication, loop fusing, and other optimizations. This can also be the tuning of task parameters to achieve higher throughput, and a more optimal use of resources and bandwidth. In addition, an initial selection of CUs that can be placed on PRRs must be done.
- **Mapping** is the step in the design flow where application tasks are assigned to physical nodes, such as compute devices or CUs. These nodes can be connected to the entry points of an interconnection network. For example, in a NoC-based MPSoC, these entry points are the routers.
- **Scheduling** determines when to execute which task on which physical nodes. The scheduler is responsible for maintaining the schedule and synchronization between tasks. The schedule can be monitored in a centralized or decentralized way.
- **Clustering** is a method that combines multiple CUs to place them into one PRR to achieve an efficient resource utilization if there are more CUs than regions. This PRR can be either statically or dynamically reconfigurable. It can represent a complete FPGA in a cluster or a region of an FPGA.
- **Placement** is the step that allocates resources and places CUs, including their associated tasks, inside of PRRs. A set of CUs must be placed in these regions so that they do not overlap and do not exceed space boundaries [114].

The various subareas shown in Figure 2.13 do not have to be performed in sequential order. The design process can perform them repetitively, interleaved or in parallel to each other to create an optimized application specific system. However, computing all these subareas in parallel would make the design process extremely complex. The various subareas span multiple dimensions in the design space, which include physical, temporal, and functional partitioning of the original applications to a physical architecture.

- The **functional** dimension partitions the application into task graphs and tunes it within the design space.
- The **temporal** dimension assigns tasks to physical nodes and creates the schedule to improve performance while taking communication overhead into account.

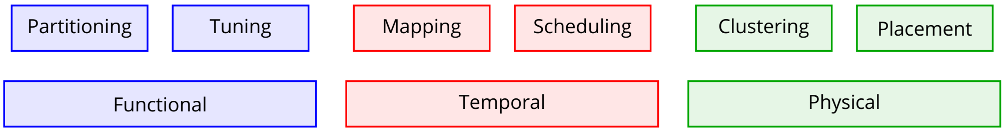


Figure 2.13: Subareas for the application distribution process on FPGA-based systems including their main dimensional categorization within the design space.

- The **physical** dimension clusters and places CUs to PRRs considering the available resources. In addition, a resource graph needs to be created and the physical nodes need to be selected.

However, the different subareas cannot be assigned to only one of these dimension types. For example, DPR adds a temporal dimension to the clustering and placement problems, since different physical nodes are assigned to the same resources even though they exist at different points in time. This leads to different configurations in time which will have different communication profiles. Creating the resource graph containing PRRs and selecting CUs to be placed in these regions also adds a physical dimension to the partitioning and tuning subareas. Additionally, the different subareas have an impact on each other. For example, a different physical clustering and placement will affect the required synchronization of the schedule. In addition, partitioning and tuning algorithms should consider communication costs and physical constraints, such as resource consumption.

The different subareas can have different objectives or constraints, which they address using different approaches. For example, performance optimization in the tuning process is a tradeoff between whether the application becomes memory-bound or compute-bound. Reducing the hop count is a common method for mapping and placement problems to reduce energy consumption. Reducing the critical path in clustering and placement processes is a valid way to improve application performance. Various metaheuristics such as GA, TS, ACO, and SA or machine learning techniques can be used to improve the performance and complexity of the toolchain. The efficient use of DPR techniques can improve resource utilization. Furthermore, temperature distribution and system reliability are also valid objectives, besides tool performance, application performance, resource consumption and energy consumption. Based on the application distribution, the final system often requires mechanisms for synchronization, error detection, communication, and memory management.

This thesis considers all six subareas within the application distribution process [29, 30]. The focus is the process of clustering CUs and placing them to PRRs in mesh-like topologies. Preoptimization using clustering before placement holds the problem of losing information and therefore optimality, thus ending in a local optimum. Therefore, both processes are merged, while keeping the complexity of the algorithm low. The proposed algorithm focuses on design time methods for heterogeneous systems. However, the final system, created from the application distribution results, also includes a minimal runtime system including synchronization mechanisms, memory management, and a communication protocol. The combination of load balancing techniques and metaheuristics is a unique feature of this thesis. By using a more generic definition of the application and platform model, this thesis can cover a much wider range of different architectures and topologies than most of the related work.

Table 2.4 compares the various publications concerning application distribution from different areas. Since the number of comparable studies to this thesis is limited, it also considers related research areas. The comparison can roughly be divided into the following areas:

- subareas of the application distribution process
- layout and size of the task graph(s)
- support for PRRs or multiple FPGAs
- topology and architecture of the compute cluster
- heuristics and methods for application distribution
- objectives, constraints and multithreading capability of the toolchain

Table 2.4: Comparison of different application distribution algorithms that target FPGAs. CLB (Configurable Logic Block), GCN (Graph Convolutional Network)

	Filho 2015 [114]	Novaes 2019 [138]	Minhas 2019 [137]	Mohtavip. 2020 [134]	Mohtavip. 2020 [133]	Geng 2018 [126, 127]	Proposed 2022 [29, 30]
Partitioning	✗	✗	✗	✗	✗	✓	✓
Tuning	✗	✗	✓	✗	✗	✓	✓
Mapping	✗	✗	✗	✗	✗	✗	(✓)
Scheduling	✗	✗	✗	✗	✗	✗	✓
Clustering	✗	✗	✓	✓	✓	(✓)	✓
Placement	✓	✓	✗	✓	✓	(✓)	✓
Dependent tasks	✓	✓	✗	✓	✓	✓	✓
Multiple tasks	✗	✗	✗	✗	✗	✓	✓
Multiple apps	✗	✗	✓	✗	✗	✗	✓
Application	9 synth 13-26 core	9 synth 13-26 core	11 apps	3 apps 1752 CLB	4 apps 32-77 CLB	AlexNet VGG16/19	13 synth 1 app 24-271nodes
PRR/DPR	✓	✓	✓	✗	✗	✗	✓
Multi FPGA	✗	✗	✗	✗	✗	✓	✓
Topology	mesh, NoC	mesh, NoC	-	mesh	mesh	line	mesh, NoC
Node type	core	core	IP-core	CLB	CLB	IP-core	AC, PU, I/O
Methods	genetic algorithms	tabu search	ridge regression	GCN, simulated annealing	distance model, forbidden region	weight balancing, pipelining	load balancing, TS,SA
Objectives	hop count, tool time	hop count, tool time	throughput, energy	hop count	hop count	performance, energy	resources, hop count, bandwidth
Constraints	-	-	resources	-	-	resources	resources bandwidth
Multithreading	✗	✗	✗	✗	✗	✗	✓

Most of the works that have dealt with application distribution on MPSoCs have predominantly dealt with the mapping and scheduling problem, which is needed in dynamic runtime systems [107]. However, they have a fixed architecture at design time and contain only PUs that are statically connected to entry points. In contrast, the works that looked at the clustering and placement problem did not deal with the mapping and scheduling problem [114, 126, 127, 137, 133, 134, 138]. This is mainly because only ACs or CLBs and no

PUs were used in these systems. Even though none of the works in Table 2.4 addressed the mapping/scheduling problem, some of these works still required temporal quantization due to the use of PRRs [114, 137, 138]. However, this quantization is very coarse-grained.

For many clustering and placement problems, the goal is to map a task graph to a resource graph. This is true for different domains, such as CLBs in CAD tools or PRRs in single or multiple FPGAs. Filho et al. [114] and Novaes et al. [138] looked at the placement problem for PRRs on the DyNoC architecture, using different heuristics. Minhas et al. [137] looked at clustering and tuning problems for independent applications on a single FPGA. Mohtavipour et al. [133, 134] worked on dealing with the placement problem of CAD tools and how they can be improved using clustering methods. Although CAD tools are not the target of this thesis, there are similarities to them in the placement and clustering process due to the regular structure of FPGA resources and mesh topology. This implies that the tools developed in this thesis can also be used to solve the clustering and placement problem of CAD tools.

Geng et al. [126, 127] and Owaida et al. [84, 85] focused on the partitioning and tuning problems for neural networks on FPGA clusters. The information from the use case allows for an easier partitioning and better fine-tuning at a functional level. However, by focusing on the application, the presented algorithms lose generality. No special scheduling or mapping is needed because their FPGA designs are only based on ACs. Since they are working on FPGA clusters, there is a simple clustering and placement process integrated in their partitioning algorithm. Like the work of Owaida et al., this thesis also introduces a lightly weighted communication protocol. The proposed architecture of this thesis focuses on the communication between different types of CUs and the avoidance of deadlocks.

2.4 Programming Methods

Several concepts emerged to overcome the difficulties of an efficient programming of parallel architectures and heterogeneous systems. This section will discuss how these architectures and systems can be programmed. There are different approaches, which on the one hand have a high expressiveness, and on the other hand simplify the programmability. These approaches fall into two categories, domain-specific and general purpose. This section will concentrate predominantly on C++-based approaches, since they are the most widespread and can address a wide variety of parallel architectures.

One approach is to use a GPL. They often contain explicit constructs to program parallel structures. Furthermore, these types of languages provide additional constructs needed to copy data or to synchronize between different nodes, such as compute devices or CUs. The great expressiveness of these languages allows a wide range of applications and programming codes to achieve high performance. However, with languages like CUDA and OpenCL many new paradigms must be learned, and the required program code is greatly increased.

There are two approaches to simplify programming using GPLs. They allow to achieve useful results faster, but do not always reach the same maximum performance. Directive-based languages use directives to parallelize code and offload it to compute devices. Single-source approaches, such as SYCL, aim at eliminating the separation of host and compute device code without using directives.

Another possibility is to add the information of the domain, such as computer vision. There are two approaches used to program parallel heterogeneous architectures that incorporate domain information. One approach is to use a library specifically designed for the application field. The more generic approach is to use a DSL, which is based on a standard language such as C/C++.

2.4.1 General Purpose Approaches

Different programming paradigms exist to ease the complexity of programming parallel architectures within heterogeneous systems. However, using a new programming paradigm and dealing with different types of architectures adds additional challenges. Developers need to detect potential program parts that would benefit from running on a compute device, like a GPU or FPGA. The choice depends on the computation and communication time of the implemented algorithm since compute devices do not always share the memory with the host CPU.

Although there are alternative approaches, C/C++ based languages are most common for programming applications on parallel architectures within heterogeneous systems. Most of the alternative approaches, like python, use common programming languages, like CUDA or OpenCL, as an intermediate language. CUDA [143] and OpenCL [144] made it possible to use GPUs for general-purpose computing. They allow the use of GPUs beyond the task of computing image contents and make it easier to execute scientific calculations on them. Table 2.5 gives an overview of different C/C++-based general purpose programming approaches to program x86-based parallel computing architectures for a set of vendors. The different languages in the table will be explained in the following.

Table 2.5: Existing C/C++ based parallel programming approaches for different vendors and compute devices including the papers that extend those approaches.

Device	Vendor	CUDA	OpenCL	SYCL	OpenMP	OpenACC
CPU	Intel	Stratton [145]	✓	✓	✓	✓
	AMD	Stratton [145]	✓	✓	✓	✓
GPU	Intel	Harvey [146]	✓	✓	✓	✗
	AMD	Harvey [146]	✓	✓	✓	✓
	NVIDIA	✓	✓	✓	✓	✓
FPGA	Intel	Mavroidis [147]	✓	✓	Knaust [148]	Lee [149]
	XILINX	Papakons. [150]	✓	✓	Sommer [151]	✗

CUDA

One of the first approaches to program GPUs scientifically was CUDA [143]. It is a parallel computing platform and programming model to program NVIDIA GPUs and was released in 2006. Developers can program their code in C, C++, Fortran, Python and MATLAB. Tasks can be offloaded using kernels called from the host program. The API defines several functions for memory movement and allocation, and for querying device and platform information. As shown in Table 2.5, there are several approaches from different researchers to extend CUDA for other types of architectures [146, 147, 150, 145]. Most methods use transpilers to convert the CUDA code to a different language to target other platforms.

Stratton et al. [145] present the MCUDA framework, which enables efficient execution of CUDA programs on shared memory multicore CPUs. Their framework consists of a set of compiler transformations and a runtime system for parallel execution. They can demonstrate that their framework approaches the performance that can be achieved with manually parallelized and optimized C code. Harvey et al. [146] present Swan, a transpiler that translates CUDA to OpenCL code. Using OpenCL also other devices, like GPUs from AMD or Intel, can be programmed. For NVIDIA GPUs, CUDA remains the better programming paradigm, since it has been developed for these devices and can therefore outperform OpenCL on NVIDIA GPUs.

There are also approaches to program XILINX [150] and Intel [147] (formerly Altera) FPGAs using CUDA. Mavroidis et al. [147] present FASTCUDA, which is an open-source FPGA accelerator and hardware/software co-design toolchain for CUDA kernels. To program kernels on FPGAs they transform the CUDA code to SystemC, as an intermediate HLS language. The software code is executed on an embedded multicore CPU. The generated software also contains the host code that controls the device kernel code. Papakonstantinou et al. [150] present FCUDA, to efficiently map coarse-grained and fine-grained parallelism onto an FPGA. They use a transpilers that transforms CUDA thread blocks into parallel C code. They use the AutoPilot [152] HLS tool as backend to program the FPGA.

OpenCL

OpenCL [153, 154, 144] is an open-standard and API for general-purpose parallel programming of heterogeneous systems. It is a very promising programming model, due to its rich instruction set and support of TLP and DLP, as well as heterogeneous platforms. Developers can implement applications for all kinds of architectures, like CPUs, GPUs, DSPs and FPGAs. Nearly all vendors (AMD, NVIDIA, Intel, XILINX, ARM, and IBM) provide an implementation for their different devices. This eases the programmability of heterogeneous systems since the developer does not need to learn multiple languages and can access different devices from one program code. The concept of OpenCL can be divided into execution model, platform model and memory model [153].

The OpenCL **Platform Model** is shown in Figure 2.14. It consists of one host (e.g. CPU) and one or more compute devices (e.g. CPU, GPU or FPGA). Each compute device is composed of one or more CUs, such as a CPU or GPU core. Each CU is further divided into one or more PEs. The computations on a device take place on the PEs. On most compute devices, the PEs of a CU are executed in SIMD manner.

The OpenCL **Execution Model** splits an application into host and kernel code. The host code initializes and manages the OpenCL specific contexts and moves data between host and devices. It can execute kernels on multiple devices from different vendors at the same time. The kernel code is portable to any compatible device, but not in terms of performance. A kernel instance concurrently executes work items on a device over a virtual grid, called NDRange, with one to three dimensions. Work-items are grouped into work-groups. The size

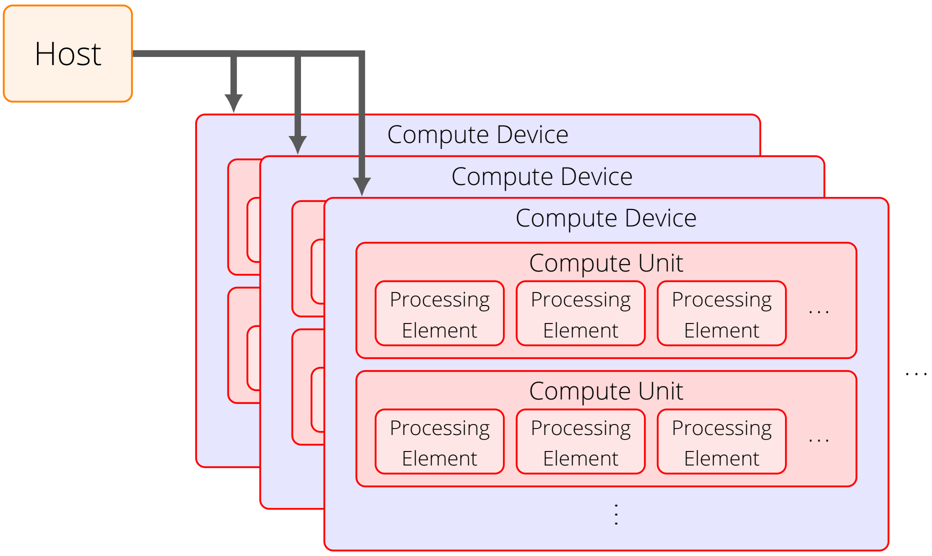


Figure 2.14: The OpenCL platform model [153].

of these work-groups can either be set manually by the user or generated automatically by the OpenCL runtime system. The advantage of the automatic approach is that a near-optimal size does not have to be identified by the user for each single device. If no particular size is required by the kernel code, this approach often achieves a near-optimal result.

The OpenCL **Context** is bound to a specific host driver installation and stores the:

- Platform ID: represents the vendor installation on the host.
- Devices IDs: physical devices supported by the platform.
- Kernel objects: function instances, which are executed on devices.
- Program objects: store the source and executable files of the kernels.
- Event objects: are used for synchronization between kernels and profiling.
- Command Queue objects: execute commands for devices.

Each command queue is linked to an OpenCL device. The host can place three different types of commands into a command queue:

- commands to enqueue a kernel for device execution.
- commands to transfer data between the different host and device memories.
- commands for synchronization.

The OpenCL **Memory Model** is shown in Figure 2.15. The host memory is defined outside of the OpenCL context and is available by the host. Memory objects can move between host and OpenCL memory space through functions. The device memory is split into private, local, global, and constant memory. The global and constant memory can be cached.

- The private memory is visible to its work-item that is executed on a PE. Work-items cannot see the private memory of other work-items.

- The local memory is visible to its work-group that is executed on a CU. Work-items within a work-group can share data through the local memory.
- The global memory of a compute device is visible to all work-items in all work-groups of the corresponding compute device.
- The constant memory is a region of the global memory that remains constant during the execution of a kernel.

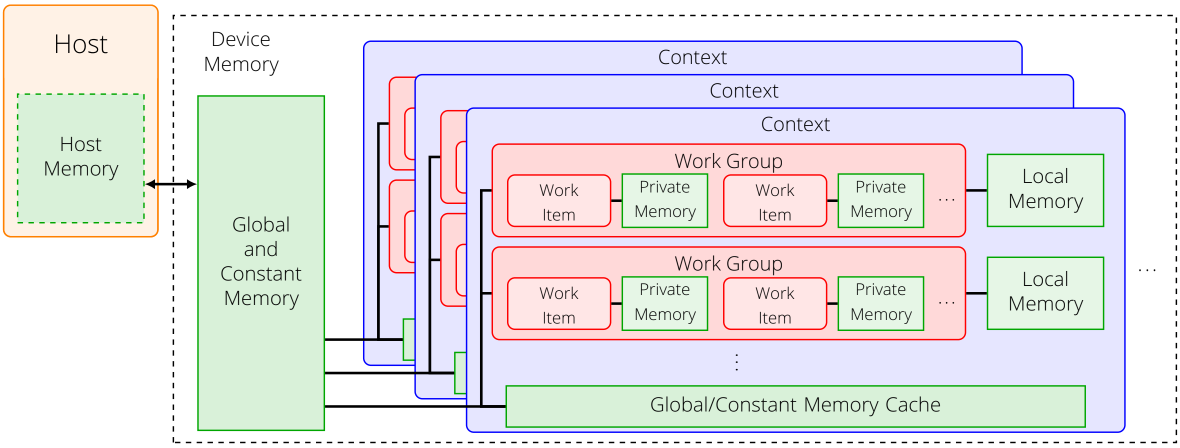


Figure 2.15: The OpenCL memory model [153].

OpenMP and OpenACC (Directive-Based)

A further general-purpose based approach to program parallel architectures is to use directives. In this approach, the programmer adds directives to the code, such as C/C++, as a hint for the compiler. The compiler can transform the source code based on the directives or ignore them. One advantage of this type of programming is that the program is still usable without the directives. Another advantage of directive-based languages in comparison to OpenCL or CUDA is the reduction of the needed host code or boilerplate code. However, directive-based languages often use OpenCL or CUDA as an intermediate language for GPU and FPGA targets [155, 149, 148]. The directive-based approach eliminates the need to package accelerator code into separate functions, explicitly manage data transfers, and optimize device memory usage. Reducing boilerplate code also comes with some drawbacks. For example, the level of control required when distributing multiple tasks across multiple nodes and optimizing data transfers and synchronization.

One of the most known directive-based languages is OpenMP [156]. It is an API designed for shared memory programming on CPUs in C/C++ or Fortran. Directives are used to tell the compiler, which code areas should be executed in parallel on a compute device, like a CPU or a GPU. OpenMP implements a typical fork-join model, where a primary thread forks a number of subthreads and divides the tasks among them, as shown in Figure 2.16. Parallel execution is managed by the OpenMP runtime system in the background. Later versions of OpenMP introduced constructs that focus on different aspects of parallelism, such as DLP or TLP. While memory management is mostly covered by OpenMP, only a basic task scheduling exists and finding an efficient schedule is left to the developer.

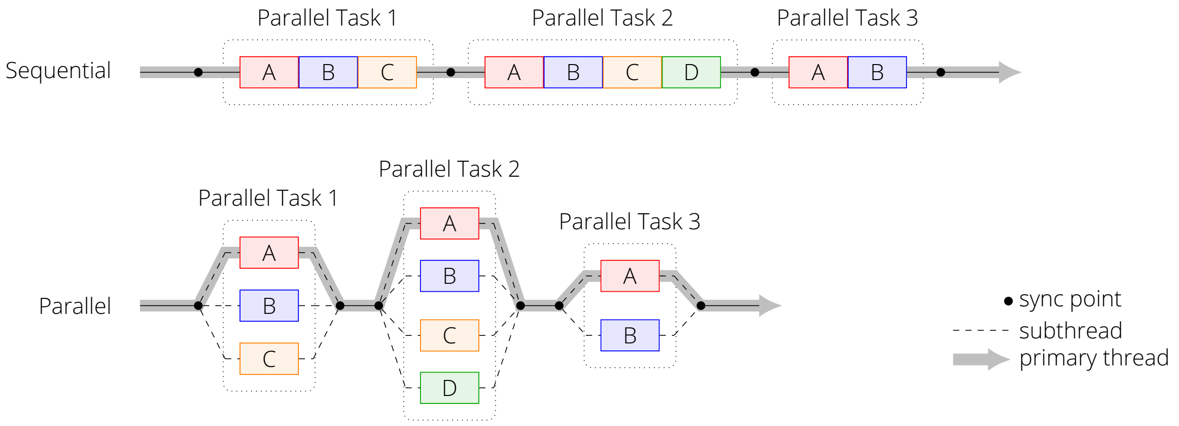


Figure 2.16: The OpenMP fork-join model.

In contrast, OpenACC (Open Acceleration) [157] is a directive-based language designed for compute devices, like GPUs. OpenACC uses directives, as hints for the compiler, to show which part of the application should be executed on a compute device. It consists of a collection of compiler directives, library routines and environment variables. It provides three different types of instructions, which are for computation, data management and synchronization. Additional information can be provided to the compiler regarding data handling, work distribution and control flow.

Over the years, both languages have evolved and learned from each other. More and more constructs and available platforms have been added as shown in Table 2.5. The progress also brought the maximum achievable performance closer to languages like OpenCL and CUDA. Due to the original intention of both languages, they still have some fundamental differences as shown in Table 2.6. OpenMP aims to extend known concepts from multicore programming, while OpenACC was developed for GPU users [158]. Both OpenMP and OpenACC have been extended by various researchers for FPGAs using HLS methods.

Table 2.6: Comparison of directive-based languages.

OpenMP	OpenACC
Consistent, predictable behavior between implementations	Quality of implementation will greatly affect performance
Users can parallelize non-parallel code and protect data races explicitly	Users must restructure their code to be parallel and free of data races
Substantially different architectures require different directives	High-level parallel directives can be applied to different architectures

Lee et al. [155] presented OpenARC, an open-source framework that implements the OpenACC model. They take standard OpenACC C code and use source-to-source transformations to target heterogeneous devices, like NVIDIA GPUs using CUDA [155]. They have extended OpenARC to perform source-to-source translations and optimizations of an input OpenACC program into OpenCL code [149]. This is further compiled into an FPGA program using the Intel OpenCL compiler as backend.

Many researchers have addressed the task of developing OpenMP for FPGAs. Mayer et al. [159] give a good overview of many different approaches. Sommer et al. [151] presented an OpenMP device offloading model for XILINX FPGAs integrated into the existing LLVM (Low Level Virtual Machine) offloading infrastructure. The programmer can generate a complete FPGA design, including memory and host connectivity, from an input source code. Knaust et al. [148] presented a HLS approach that uses OpenMP to target Intel FPGAs using Intel's OpenCL SDK.

There are many other directive-based languages besides OpenMP and OpenACC. For example, Duran et al. [160] presented OmpSs, a programming model based on OpenMP and StarSs [161] for the execution on CPUs and GPUs. Bosch et al. [162] extended OmpSs to program heterogeneous systems, like GPUs and FPGAs using CUDA, OpenCL and the HLS toolchain from XILINX.

SYCL (Single Source)

SYCL is a royalty-free, cross-platform abstraction layer specified by the Khronos Group, to overcome the issue of the source code separation in OpenCL [163]. It enables code for heterogeneous systems to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file. SYCL builds on the features of C++11, with additional support for C++14 and C++17, enabling parallel STL (Standard Template Library) programs and using lambda expressions to integrate kernels as anonymous functions. It is mainly implemented as a C++ template-based library and is interoperable with already existing OpenCL code. This allows the developer the reuse of existing OpenCL kernels. Therefore, it is also possible to interact with Vulkan and OpenGL (Open Graphics Language) without transformation overhead. Since SYCL is an abstraction layer on top of OpenCL, it inherits many further features:

- a broad hardware support
- a set of data types representing multidimensional vectors and images
- device partitioning into subdevices
- shared virtual memory
- pipes (FIFO (First-in-First-out) buffers)
- support of SPIR-V as an intermediate representation

SYCL adds the idea of data accessors, representing the data access of kernels. It detects data dependencies to build the processing graph. The kernel scheduling is based on OpenCL queues, but only supports out-of-order queues. SYCL is managing the processing order and corresponding data movements by itself. Explicit buffer transfer and event handling, as in OpenCL, is therefore not needed. This prevents from unnecessary copying of data and allows the runtime to optimize data movements. All SYCL data types and objects are inherited from OpenCL. To maintain interoperability, the developer should be able to access the underlying OpenCL object.

SYCL as an abstraction layer on top of existing languages like OpenCL comes with some overhead. Silva et al. [164] compare SYCL with OpenCL and OpenMP using two example applications. In average the SYCL implementation needed a 2.18 times higher execution time and 0.52 times more memory in comparison to OpenCL and OpenMP as shown in Table 2.7. The 27stencil problem [164] is an algorithm used by PDE (Partial Differential Equation) solvers, which update a multidimensional grid in both time and space. The CMP (Common MidPoint) problem [164] is an optimization method for seismic data to reduce noise. As the publication was still in an earlier state of SYCL, the results should be better by now.

Table 2.7: Relative execution time, memory usage, kernel size and API calls of SYCL in comparison to OpenCL and OpenMP [164].

	27Stencil		CMP	
	OpenCL	OpenMP	OpenCL	OpenMP
Execution time	2.35	2.22	2.77	1.38
Memory usage	0.39	1.16	0.20	0.32
Kernel size	0.73	1.19	0.85	0.46
API calls	0.45	4.50	0.75	25.00

Since SYCL is a specification of an API, the set of provided features highly depends on the implementation. Depending on the implementation and target device, languages other than OpenCL, such as CUDA or OpenMP, are also used internally. Most libraries available on the market use OpenMP as the basis for parallelization on CPUs and as a fallback solution. Figure 2.17 gives an overview of the most advanced implementations that realized part of the SYCL specification [165]. It also shows the supported devices and used programming models.

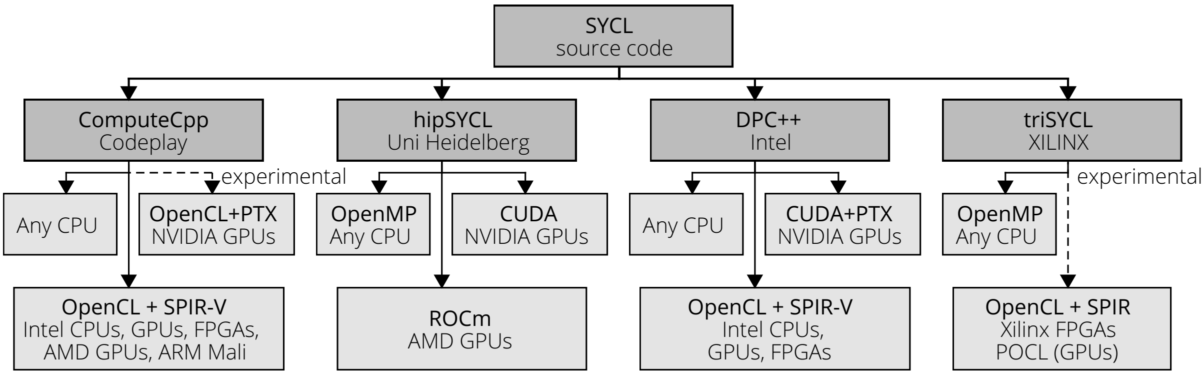


Figure 2.17: Available implementations of the SYCL specification, their supported devices and used programming models [165].

ComputeCpp from Codeplay offers a professional version including support and a free community version for research and evaluation. In terms of usability, this library provides the best user experience thanks to precompiled packages for Windows and Linux distributions and a reasonable user documentation.

HipSYCL is not based on OpenCL, but on OpenMP, AMD's ROCm and NVIDIA's CUDA, which has advantages and disadvantages. On the one hand, vendor-specific frameworks are more optimized in terms of performance than OpenCL [166]. On the other hand, the missing of OpenCL leads to incompatibility with all other OpenCL devices.

In contrast to the previously mentioned projects, **Intel** is working on an implementation of SYCL for the LLVM compiler framework. There are ongoing efforts to incorporate this work into the upstream build of LLVM. This will improve the usability of SYCL device compilers and the overall development process.

TriSYCL is an open-source implementation of SYCL, formerly started by AMD and now led by XILINX. It is used as a testbed to provide feedback to the Khronos group and to influence the standardization development of SYCL. The team behind triSYCL is working to merge their work with the Intel project mentioned earlier. triSYCL also provides a header-only library, which simplifies its integration. Besides these four projects, there is a long list of further projects that implement and support parts of the SYCL standard [167].

2.4.2 Domain Specific Approaches

When developing an application that is in a specific domain, it is advantageous to use a DSL or library. The developer needs to deal less with the implementation details and can focus more on the design of the algorithm compared to a general-purpose language. Both approaches are good to further abstract the hardware from the application developer. The border between both approaches is seamless. On the one hand, a DSL can provide greater flexibility. On the other hand, a library can provide further abstraction. Therefore, a combination of the two approaches is advantageous.

2.4.2.1 Libraries

In addition to the realization of SYCL-based libraries, there are numerous other generic C++ libraries which rely on parallel programming paradigms such as OpenCL, CUDA or OpenMP. By their nature, most libraries are limited to a specific domain due to the large number of possible applications. Denis et al. [168] present a comparison of modern C++ libraries providing high-level interfaces for programming multicore and many-core architectures. The comparison focuses on the solution of ODEs (Ordinary Differential Equations) for libraries such as Thrust, MTL4, VexCL or ViennaCL. One conclusion of this study was that CUDA and OpenCL perform equally well on large problems, while OpenCL has a higher overhead on smaller problems. In addition, they show that modern high-level libraries make it possible to effectively use the computational resources of GPUs or CPUs without much knowledge of the underlying hardware. Another conclusion was that due to the similar performance, the differences in the programming interfaces of the libraries are more relevant for the choice of the library. A few of these libraries will be described in the following paragraphs.

Thrust [169] is a C++-based parallel library to implement high-performance applications with minimal programming effort for STLs. Its high-level interfaces significantly increase developers' productivity while enabling performance portability between GPUs and CPUs. Internally, it uses CUDA to program GPUs.

The MTL (Matrix Template Library) [170] is a linear algebra library providing an intuitive interface by creating a DSL embedded in C++. The library aims at maximizing performance, achievable through high-level languages and the use of compile-time transformations. Different versions exist: (1) an open-source edition supporting CPUs, (2) a supercomputing edition providing MPI and (3) a CUDA version supporting NVIDIA GPUs.

VexCL [171] is a vector expression template library for OpenCL and CUDA, which has been developed for general purpose GPU development using C++. It aims at reducing the amount of boilerplate code needed and supports computation on multiple devices and platforms. It provides a convenient and intuitive notation for vector arithmetic, reduction, sparse matrix-vector products, and more.

ViennaCL [172] is a free open-source linear algebra library for computations on multicore and many-core architectures such as CPUs, GPUs, or microcontrollers. The developer only needs to interface with a single library to use different parallel architectures. The library is written in C++ and uses CUDA, OpenCL and OpenMP as backend. Despite the hardware abstraction, target-specific optimizations can still be applied. It also achieves a better overall performance than the respective vendor libraries, while preserving the ability to apply target-specific optimizations. Although it does not provide a solution for task scheduling or data consistency, it enables portable performance for GPUs through a device database coupled with a kernel code generator.

2.4.2.2 Computer Vision Libraries

The number and complexity of image processing and computer vision applications is growing continuously [173, 3, 174]. Libraries are a good solution to reduce algorithm details from the developer. The use of optimized libraries for specific devices, like GPUs or FPGAs, further abstracts away the hardware specific optimizations.

Frameworks such as TensorFlow [175] or Caffe [176] have been developed for the field of deep learning. The developer uses them via scripting languages, which fully abstract away the implementation details of the underlying hardware. They are mainly optimized for GPU programming. Shi et al. [177] show a good comparison between different deep learning frameworks, such as Caffe, CNTK, TensorFlow and Torch.

To ease the development process of computer vision applications, standards, and libraries such as OpenVX [32] and OpenCV [178] have been proposed and are widely used. OpenCV is an open-source computer vision software library, which is built to provide a common infrastructure for computer vision applications. It is one of the most known and used libraries in the field of computer vision and image processing. Parts of the OpenCV library have been optimized to use GPUs and multicore CPUs in an efficient way. The algorithm developer can turn optimizations, like OpenCL, OpenMP or CUDA, using specific flags.

Since version 3.0, OpenCV supports OpenCL device acceleration through the T-API. A wide range of functions are seamlessly accelerated through fine-tuned image processing functions. Figure 2.18 shows the acceleration achieved by OpenCL on a Radeon HD7790 GPU and an integrated GPU of an AMD A10-7850k (Kaveri) CPU compared to a C++ implementation on the same CPU [179].

OpenVX [32] is an open, royalty-free standard for cross-platform acceleration of computer vision applications. Because of its graph-based approach and memory management it is much more than just a library. Various vendors implemented compliant libraries or frameworks, such as MIVisionX (former AMDOVX) from AMD or VisionWorks from NVIDIA, due to its well-defined interfaces. Therefore, it is predestined to create computer vision applications for heterogeneous systems.

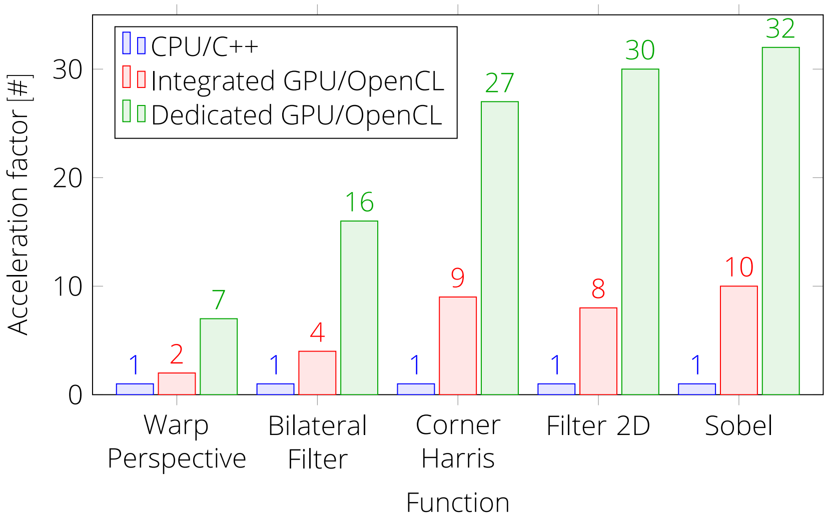


Figure 2.18: The OpenCL acceleration of OpenCV, adapted from [179].

Table 2.8 shows a list of OpenVX compliant or similar libraries [179]. The libraries or tools are usually highly optimized for their own architectures, so combining them requires some effort. For example NVIDIA for their GPUs and SoCs, Cadence for their Tensilica Vision DSPs, Imagination for their PowerVR GPUs, or Texas Instruments for their automotive SoCs.

Table 2.8: OpenVX or similar libraries with their conformance version and acceleration.

Vendor library	OpenVX conformance	Acceleration method
AMDOVX	1.0.1	OpenCL and optimized x86 code
Cadence	1.1	Device driver
Imagination	1.1	Device driver
Texas Instruments	1.1	Device driver
NVIDIA VisionWorks	1.0	CUDA
OpenVINO	1.1	OpenCL and optimized x86 code
CLIJ2	✗ (similar functions)	OpenCL
OpenCV	✗ (similar functions)	OpenCL

OpenVINO, Intel's Open Visual Inference & Neural Network Optimization Toolkit, enables the development of applications and solutions that emulate human vision. The toolkit is based on CNNs and is capable of extending workloads to Intel's CPUs, GPUs, VPUs (Vision Processing Units) and FPGAs while maximizing application performance.

The **CLIJ2** library [180] implements many image processing functions as OpenCL kernels. It was originally developed as a module for the ImageJ image processing program, but its kernels can also be used via bindings for C, Java, and Python. Unfortunately, the language bindings are not yet fully usable, as they are still in a prototype phase.

AMDOVX is a highly optimized implementation of the OpenVX specification. The library enables fast prototyping as well as fast execution on a variety of architectures, ranging from small embedded x86 CPUs to large discrete workstation GPUs. Its source code is available on

GitHub and functions are accelerated using OpenCL kernels. The library consists of several components:

- An OpenVX library that provides a standards-conformant implementation of the specification. It includes a graph optimizer that examines the entire processing pipeline and removes, replaces, and merges functions.
- The RunVX command line tool for executing OpenVX graphs. It defines its own DSL, which provides a simple and intuitive syntax for describing the various data buffers, nodes, and dependencies in a graph.
- The RunCL command line tool for creating, running, and debugging OpenCL programs.

2.4.2.3 FPGA-based Computer Vision Libraries

The benefits of FPGAs to process computer vision applications in comparison to other architectures, like CPUs and GPUs, has been shown by different researchers [7, 8, 10]. Qasaimeh et al. [10] compare the energy efficiency of CPU, GPU, and FPGA implementations for vision kernels on embedded platforms. They evaluated embedded vision applications on different devices (ARM Cortex-A57 CPU, NVIDIA Jetson-TX2 GPU and XILINX ZCU102 FPGA) using the vendor optimized libraries (OpenCV, VisionWorks and xfOpenCV). The GPU achieved an energy/frame reduction ratio of 1.1 times to 3.2 times for simple kernels in comparison to the other devices. The FPGA outperforms the other devices with an energy/frame reduction ratio of 1.2 times to 22.3 times for more complicated kernels and complete vision pipelines. In conclusion, it was observed that FPGAs perform increasingly better as the pipeline complexity of the computer vision application grows.

Several libraries have been proposed which reduce the complexity of developing computer vision and image processing applications on FPGAs. For example, XILINX released their own FPGA-oriented OpenCV implementation called xfOpenCV [181], which has already been used in several systems [174]. xfOpenCV also includes many computer vision functions, which are part of the OpenVX specification. xfOpenCV can be integrated into the different HLS toolchains from XILINX.

Özkan et al. [182] proposed a highly efficient and parametrizable C++-based library for image processing applications. It targets HLS to produce optimized algorithms for FPGAs. The motivation behind their work is the implementation of image processing applications that can be expressed as DFGs (Data Flow Graphs). They also provide designers multiple Pareto-optimal architectures for the same library instances to tailor their implementation.

Vasiljevic et al. [183] show the potential of streaming data between hardware components on FPGAs. They presented an OpenCL library of preoptimized stream memory components targeting FPGAs using SDAccel. A video watermarking and a matrix multiplication are used to evaluate the library, by comparing a naive with an optimized implementation. Unfortunately, the results are not compared to alternative architectures, like CPUs or GPUs.

2.4.2.4 FPGA-based Computer Vision DSLs

The use of a DSL increases the compiler's ability to generate efficient code for a certain application field. There are numerous DSLs that reduce the complexity of developing computer vision applications, such as Halide [184, 185], HIPA^{CC} [186, 187, 188], PolyMage [189, 190], Darkroom [191] or Rigel [192]. Table 2.9 gives a comparison of four known image processing and computer vision DSLs and their FPGA extensions.

Table 2.9: Comparison of FPGA-based computer vision domain specific libraries.

	Halide	HIPA^{CC}	PolyMage	Darkroom
Publication	Ragan-Kelley [185]	Membarth [186, 188]	Mullapudi [190]	Hegarty [191]
FPGA Extension	Pu [184]	Özkan [187]	Chugh [189]	Hegarty [191]
Input	embedded in C++	embedded in C++	embedded in Python	embedded in Terra
Compiler	own IR lowered to LLVM-IR	Clang (LLVM)	Polyhedral compiler	Terra, Genesis2
Output	OpenCL, CUDA, OpenGL	OpenCL, CUDA, C/C++	C++	System Verilog using Synopsis and Vivado
Architecture	CPU (x86, ARM), GPU, FPGA	CPU, GPU, FPGA for x86, ARM	CPU (Intel), FPGA (XILINX)	CPU (x86), FPGA, ASIC
License	Apache 2.0	BSD-2, BSD-3	Apache 2.0	MIT

Ragan-Kelley et al. [185] presented a high-performance programming language and compiler, called **Halide**. It has been designed to ease the development of implementing image processing applications. It supports various CPU and GPU architectures, and OSs. Halide is embedded into the C++ language and uses its own IR (Intermediate Representation), which is lowered to LLVM-IR. It uses OpenCL, CUDA and OpenGL to program the different computing architectures. It works on data parallel pipelines based on simple interval analysis and discovers high quality schedules. It is not built on the polyhedral model, which is used in many other DSLs. Pu et al. [184] extended Halide to allow users to specify which parts of their applications should become a hardware accelerator. They provide a compiler that uses this code to automatically generate the accelerator along with the additional code needed for the user application to access that hardware. They use the HLS tools from XILINX to synthesize their accelerators. The evaluation shows that on a XILINX Zynq FPGA, they achieve a 3.5 times higher performance and a 12 times lower energy consumption than on a K1 192-core GPU from NVIDIA.

Membarth et al. [186] proposed a framework and transpiler for automatic code generation of image processing algorithms, called **HIPA^{CC}**. They show that domain knowledge can be captured within a language and that this knowledge allows to generate tailored implementations for a given target architecture. HIPA^{CC} is embedded into the C++ language and can even run normal C++ code. It provides a runtime system and uses OpenCL, CUDA and C/C+ to target x86 and ARM-based CPU and GPU architectures. Its compiler is based on the Clang frontend of LLVM. Reiche et al. [188] extended the HIPA^{CC} DSL for FPGAs. They proposed a code generation technique for C-based HLS from a high-level DSL description for XILINX FPGAs using the vendor toolchain, called Vivado HLS. Their approach includes FPGA-specific memory architectures for handling point and local operators, numerous high-level transformations, and automatic testbench generation. Their evaluation shows higher execution time but lower power consumption on a XILINX Zynq-7045 FPGA compared to an NVIDIA Tesla K20 GPU. Özkan et al. [187] extended this approach to generate highly optimized OpenCL code for Intel FPGAs. They designed a compiler backend that supports arbitrary bit-width operations.

Mullapudi et al. [190] presented **PolyMage**, a DSL and compiler. It automatically generates high-performance implementations of image processing pipelines by using complex fusion, tiling, and storage optimizations. It is embedded into the Python language and works on a Polyhedral compiler that generates C++ code to program multicore CPUs. Experimental results on a CPU show that the achieved performance is up to 1.81 times higher than by a manual tuned Halide implementation. Chugh et al. [189] extended the PolyMage work for FPGAs using the XILINX HLS toolchain. They have developed an approach to map image processing pipelines expressed in the PolyMage DSL to efficient parallel FPGA designs. It first starts exploiting the pipeline structure from a directed graph of image processing stages. It then replicates the CUs to exploit DLP until either memory bandwidth or FPGA resources are exhausted. The evaluation shows a speedup of 1.05 times, 1.06 times, 1.85 times or 15.6 times for different image processing applications using a XILINX Virtex-7 690T FPGA in comparison to an optimized implementation for an Intel Xeon E5-2680 16 core CPU.

Hegarty et al. [191] created an image processing DSL and compiler embedded into the Terra language to target CPUs, called **Darkroom**. They use the Genesis2 tool to generate System Verilog to target FPGAs and ASICs using the toolchains from Synopsis and XILINX. They work on a DAG and compile programs directly into line buffered pipelines. In comparison to Halide, they are less expressive.

The various DSLs in Table 2.9 have many similarities, but also some differences. All approaches have developed a transpiler to convert the input language to another one. This allows the use of the vendor-specific toolchains to program different architectures. The DSL constructs were built into existing programming languages, so no completely new language was developed. Consequently, existing programs can be reused. Halide and HIPA^{CC} can program most devices and are embedded in to the C++ language, which opens many more possibilities. Whereby HIPA^{CC} seems to achieve the better performance between the two DSLs. PolyMage shows very good performance and works with the polyhedron model, which opens many optimization possibilities. On the other hand, Darkroom is the only DSL that allows to program ASICs.

2.5 Toolchains

This thesis proposes a framework for application distribution of object detection applications in heterogeneous FPGA-based systems. In addition to the topics, application distribution and object detection presented in the previous sections, this toolchain comprises several topics. Four of these topics will be examined in this section based on the current SoA.

The first part looks at transpilers, their advantages and disadvantages, and their realization in research. Transpilers usually transform source code from one language to source code of the same or another language. They are used in many languages, such as DSLs, and toolchains, such as for HLS. This thesis uses transpilers to build user-generated functions for GPUs and FPGAs to be integrated more nicely into the rest of the framework.

The second part discusses HLS tools, their benefits and their realizations by researchers and vendors. Using HLS has several advantages, such as an easier and faster way of testing functional correctness, good code portability, and a much shorter design cycle compared to RTL (Register Transfer Level)-based languages. This thesis uses HLS-based toolchains to implement a C++-based object detection library for FPGAs. Furthermore, HLS-based toolchains from XILINX are integrated into the framework of this thesis.

The third part looks at FPGA-based OpenVX tools. OpenVX [32] is an open, royalty-free standard for cross-platform acceleration of computer vision applications. Because of its graph-based approach and memory management, it is much more than just a library. This thesis uses OpenVX as a common frontend of the framework to program both embedded and HPC systems.

The last part looks at existing OpenCL-capable toolchains in research. As mentioned earlier, OpenCL allows programming a wider range of different parallel architectures compared to other languages. On the one hand, OpenCL kernels are used in this thesis to program non-FPGA devices. On the other hand, this thesis uses the OpenCL host code as a low-level API in its runtime system to orchestrate the work in x86-based systems.

2.5.1 Source-to-Source Compilers (Transpilers)

Compiler optimizations can also be used to offload program code to a compute device without the influence of the developer. By using different optimization steps and adjusting the schedule, an optimized code for parallel architectures is created. A frequently used approach in this context are transpilers. These convert the input source code, for example, into a general-purpose language that is used as IR to leverage vendor toolchains. As shown in Section 2.4.2.4, DSLs use transpilers to address parallel architectures by transforming their code into OpenCL, CUDA, C++ or GPLs. This section will provide an overview of transpilers that target parallel and heterogeneous architectures. It will individually elaborate on the Polly tool and the PPCG (Polyhedral Parallel Code Generator) tool. Before that, there is a brief introduction to some essential concepts about LLVM and the polyhedral model.

Transpilers usually transform source code from one language to source code of the same or another language. Unlike normal compilers, which usually generate machine code from an input language. Transpilers have the following advantages.

- **Human readable code**: Transpilers can make machine code readable again for the developer. This allows the developer, for example, to make own changes and optimizations to the source code. Such changes would be difficult on a machine code or IR, such as LLVM-IR, MLIR [193] or SPIR (Standard Portable Intermediate Representation) [194].
- **New language features**: Using a transpiler the developer can add new features to a language. This approach is often used within DSLs, like Halide [185] or HIPA^{CC} [188].

- **Downwards compatibility**: Transpilers can also enable downwards compatibility of languages. For example, C++03 could be generated from C++20 if the target platform does not support it and the programmer does not want to lose the useful features of C++20.
- **Reuse of existing tools**: Transpilers allow an easy integration and usage of existing toolchains. This means that no new backend needs to be programmed for each target architecture, for example if a new language is developed. The integration of the vendor tools also allows to use their upgrades for new architectures without changing the backend. Various HLS tools generate, for example, VHDL or Verilog to reuse the existing toolchain of the vendor. Also, other directive-based languages like OpenACC use intermediate languages like OpenCL to address different architectures more easily.

There are, of course, some challenges in the development of a transpiler. Whether or not certain aspects of the transpiler toolchain have tolerable side effects largely depends on the desired use case. The detour via the IR increases the optimization possibilities but makes it more difficult to get back to human-readable code. Generating human-readable source code from an IR is a challenging task. Especially when naming variables since information can be lost on the way to the IR or disappear due to transformations and additional code. Therefore, it is important to pass through as much debug information as possible from the original code.

LLVM

The LLVM framework is a modular open-source compiler written in C++11 and divided into three sections (frontend, middleend and backend). The frontend parses the desired language, such as C, syntactically and semantically and translates it into the intermediate language (LLVM-IR). Using a common IR has the advantage that optimizations and transformations of the program code can be performed independently of the input language. Thus, techniques such as loop detection, dependency analysis, or transformations need to be performed only once and not for each input-output combination. The transformations and the analyses are cascaded in LLVM using passes. Each pass is programmed as a separate module and can access data from other passes. The LLVM framework ensures that these passes are executed in the correct order. In the backend, LLVM-IR is converted to the desired machine language, such as x86, x64, ARM or NVIDIA PTX.

Polyhedral Model

The polyhedral model [195] attempts to model memory accesses to arrays in nested loops as a function of the iteration variables. A program is divided into *Statements* that are executed depending on the iteration. Listing 2.1 shows a function with two *Statements* ($S0[i,j]$, $S1[i,j,k]$). They are denoted as integer tuples ($n[i_0, \ldots, i_{d-1}]$). n denotes the name and d the number of parameters. Together, the two are called a tuple space. Using the ISL (Integer Set Library) [196], calculations are performed on *Statements*. The ISL includes a solver for integer linear optimization and can be used for calculations in set theory. A program area that is writable with polyhedral modeling is a SCoP (Static Control Part), which includes the following parts:

- Fixed or parametrized loop bounds: $i < 8$; $i < n$; $\xcancel{i < A[i,j]}$

- Affine accesses: $a + \sum_{i=1}^{n}(b_i \cdot X_i) \quad \rightarrow \quad A[i \cdot 6]; \quad A[i+j]; \quad \xcancel{A[i \cdot a]}$
- Single-entry-single-exit
- Side-effect free calls

```
void matmul (float A[m][o], float B[o][n], float C[m][n], int m, int n, int o)
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
        C[i][j] = 0; // S0
      for (int k=0; k<o; ++k)
        C[i][j] = C[i][j] + A[i][k] * B[k][j]; // S1
```

Listing 2.1: Simple matrix multiplication code including two statements (S0 and S1).

Figure 2.19 shows the four main steps of the polyhedral optimization.

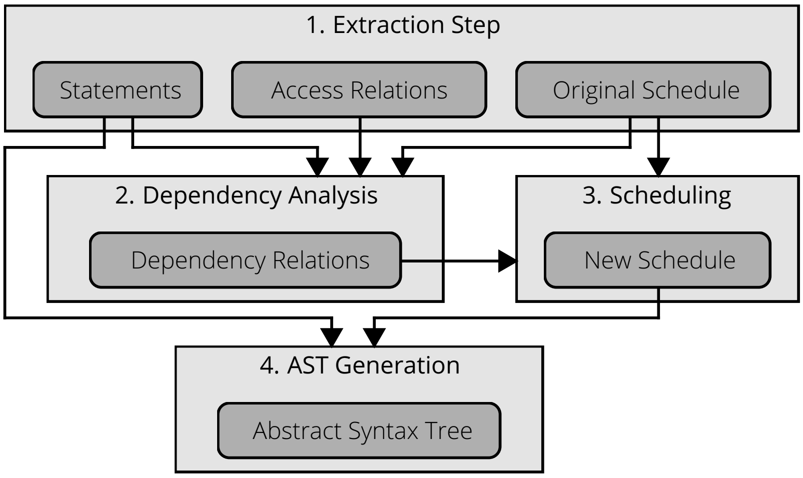


Figure 2.19: The four main steps of the polyhedral optimization.

1. The **Extraction** step searches for areas in the input code that would be suitable for the polyhedral modeling. From these areas the *Statements* of the *Instance Set*, which is the set of all dynamic execution instances, is formed. Subsequently, the memory *Access Relations* are created for the *Statements*. Depending on the *Statements*, the *Original Schedule* is created from the program code. The results of the *Extraction* step for Listing 2.1 looks as follows:
    - Statements: $S0$ and $S1$
    - Write-access relations: $S1[i,j,k] \rightarrow C[i,j]; \quad S0[i,j] \rightarrow C[i,j]$
    - Read-access relations: $S1[i,j,k] \rightarrow C[i,j]; \quad S1[i,j,k] \rightarrow A[i,k]; \quad S1[i,j,k] \rightarrow B[k,j]$
    - Original schedule: $S0[i,j] \rightarrow [i,j,0,0]; \quad S1[i,j,k] \rightarrow [i,j,1,k]$
2. The **Dependency Analyzes** step forms the *Dependency Relations*, which are binary relations between the *Statements* of the *Instance Set*. For this, the *Instance Set*, the *Access Relations*, and the *Original Schedule* are used. A relation indicates whether one element must be executed in front of the other: read-after-write (true dependence), write-after-read (anti dependence) or write-after-write (output-dependence).

3. The **Scheduling** step creates a *New Schedule* using the *Dependency Relations* and *Original Schedule*. The schedule can be recreated with the *Dependency Relations*, or the *Original Schedule* can be modified by incremental changes. The target hardware architecture is also important for the schedule.
4. The **AST Generation** step converts the *Statements* of the *Instance Set* into an AST (Abstract Syntax Tree) with the help of the *New Schedule*. The code for a high-level language such as C or a compiler intermediate language can be generated using the AST.

Polyhedral Tools

Polyhedral modeling has become the basis for automatic program optimization and parallelization [197]. LooPo (Loop Parallelizer) by Griebl et al. [198] is one of the first practical tools. PLUTO (Polyhedral Parallelizer and Locality Optimizer) by Bondhugula et. al. [199] performs automatic parallelization and optimization of program code. PLUTO is a transpiler for generating OpenMP code for CPUs with an UMA architecture. The program code is optimized to maximize parallelism and data locality by using loop tiling to optimize cache utilization. It uses LooPo as a scanner and parser. For linear optimization it uses PIPLib (Parametric Integer linear Programming) of Paul Feautrier [200]. To generate source code from the polyhedral model, Cedric Bastoul's CLooG (Chunky Loop Generator) [201] is used. A newer version of PLUTO uses the PET (Polyhedral Extraction Tool) of Verdoolaege et al. [202]. PET uses Clang as frontend instead of LooPo.

PoCC (Polyhedral Compiler Collection) combines many of the presented tools to form a compiler collection. With Polly for LLVM [203] and GRAPHITE for GCC (GNU C Compiler) [204], polyhedral optimization has also been introduced into traditional compilers. Polyhedral optimization requires integer linear programming. For Polly and other projects, this is realized in the ISL of Verdoolaege [196]. It includes an integer linear programming solver, offers modeling of sets and relations and can express piecewise quasi-affine expressions. A quasi-affine expression is a function that maps from a named integer tuple with a given tuple space to a rational number.

The C-to-CUDA compiler by Baskaran et al. [205], mainly based on PLUTO and CLooG, was the first to address GPUs using CUDA. The challenge with GPUs is the different processing and memory architecture and the fact that data often needs to be copied explicitly. A more advanced version is PPCG from Verdoolaege et al. [206], which is described in a later subsection.

Polly Tool

Polly [203] is a high-level loop and data-locality optimizer for the LLVM framework. It works on the LLVM-IR and tries to detect loops and transforms them to parallelize a program using OpenMP, SIMD or GPU code. Polly uses the polyhedral model and the ISL library for polyhedral optimization. The structure of the Polly tool is shown in Figure 2.20.

First, the *Polly Canonicalize* pass canonizes and normalizes the LLVM-IR code, so that the code is in a standard form and the subsequent steps do not need any special treatment. The detected loops are reshaped to only use one iteration variable per loop. The loops consist of a single-entry single-exit region, and they conform to the loop-closed SSA (Single

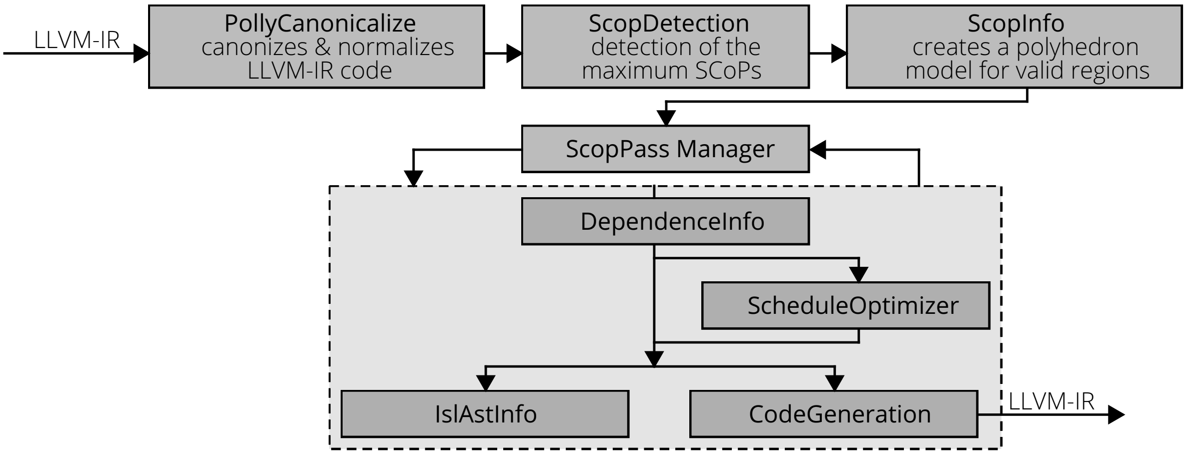


Figure 2.20: The structure of the Polly tool.

Static Assignment) form. This form checks for each loop whether a variable written inside the loop is also read outside of the loop. Then the *Scop Detection* pass checks whether parts of the program code can be converted into a SCoP. It checks if a region only has valid loops, statements, control flow graphs and affine memory accesses. A valid loop is a loop that does not allow jumps out of the current region since it can only have one entry point and one exit point. Then the *Scop Info* pass creates a polyhedral model for valid regions from *Scop Detection*. First, the *Statements* of the *Instance Set* of the current region are generated. Then, for each load, store and PHI instruction, the memory *Access Relations* are formed. PHI statements assign a value to a variable depending on from which basic block it branches to the current block. Finally, the schedule for the current SCoP is generated from the LLVM-IR code. The *Scop Pass Manager* runs various SCoP passes in sequence that perform calculations and optimizations on the SCoP and ensures the order of the SCoP passes. At the end, the *Code Generation* pass converts the *New Schedule* into LLVM-IR by generating an ISL-AST, which is then converted into LLVM-IR code.

Polyhedral Parallel Code Generator (PPCG) Tool

PPCG [206], shown in Figure 2.21, is a transpiler based on polyhedral optimization. It uses affine transformations to extract data parallelism and generate code. The focus is on multicore compute devices with strict memory hierarchies, like GPUs. PPCG is divided into several phases. PET is used to extract a polyhedral model from C code which is based on the C99 standard. The *Statements* and memory *Access Relations* are created in ISL format. Unlike Polly, PET works on the AST of Clang, which is a frontend for the LLVM framework. Like in Polly, a *Dependency Analysis* is performed using the ISL. The *Dependency Relations* are used to create a GPU optimized *Schedule*. In the *Code Generation* step, it is converted using the ISL to create OpenCL, OpenMP or CUDA code.

A multilevel tiling strategy has been implemented, which respects the hierarchical memory architecture of GPUs. It is possible to select work-group and work-item sizes regardless of the size of the array. It therefore decouples the multilevel parallelization from the optimization of the data locality. Non-fully nested loops can be split into multiple kernels. PPCG creates the host code, which allocates the memory on the compute device, organizes the data transfer, and defines the correct order of kernels. Using the *Live Range Reordering* [207] feature, PPCG

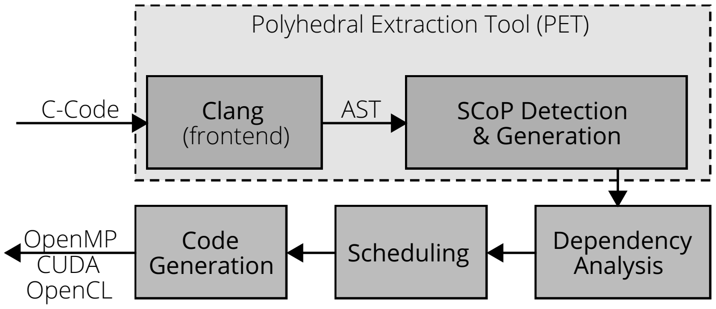


Figure 2.21: The structure of the PPCG (Polyhedral Parallel Code Generator) tool.

has a function to detect and resolve data dependencies of local variables when they are only used within the SCoP. There is a write-after-read data dependency to ensure for the next loop pass that the variable is successfully read first before it can be written again. If the computation with local variables is distributed over several arithmetic units, a separate private memory can be created for each unit to resolve the write-after-read dependency.

2.5.2 High-Level Synthesis (HLS) Tools

FPGAs are used for various purposes. Classically, they are programmed with HDLs (Hardware Description Languages) such as VHDL or Verilog. However, implementing applications for digital circuits is very time consuming and most software developers are not familiar with such programming paradigms. When it comes to the development of computer vision or image processing algorithms, such detailed knowledge is not always required. Moreover, algorithm developers often do not have much knowledge about FPGAs. Therefore, programming methods are necessary, which provide a further abstraction. A good step towards faster development are languages like SystemC [208] or Chisel [209]. However, the developer still needs to have a good understanding of the hardware being developed.

Developing FPGA designs with HLS has several advantages, such as an easier and faster way of testing functional correctness, good code portability, and a much shorter design cycle compared to RTL-based languages. As a result, several vendors have developed tools to program FPGAs with a high-level programming language such as C/C++ or OpenCL. Intel, for example, offers its OpenCL SDK [210, 211] for FPGA devices. XILINX also offers various HLS toolchains, such as Vivado HLS [212], which is included in SDSoC [213] for embedded systems or in SDAccel [214] for HPC systems. OpenCL is a very promising programming model for a wide range of different parallel architectures, due to its structure and rich set of functions and built-in constructs. This also improves code portability between different FPGA vendors. Another advantage of FPGAs that can be realized in a user-friendly way by OpenCL is DPR, as runtime programming of devices is part of the OpenCL standard. In return, C++ has advantages over OpenCL (version 2.0 or older) in the implementation of parameterizable libraries, for example through templates.

Hardware designs for image processing and computer vision can be developed much faster than with an HDL, with the drawback of a slightly increased resource utilization. However, due to the high progress in the VLSI (Very Large-Scale Integration) technology, an increasing amount of logic blocks are contained within one FPGA. Therefore, the impact of the slightly increased resource consumption by using HLS is becoming less. When it comes to the implementation of hardware architectures such as a PU, HLS languages are not necessarily the best choice, as the developer must consider more details and cover more corner cases. Regardless of the use case and the chosen language, the HLS developer needs experience with FPGA designs when writing efficient code. However, this is also the case with other platforms, for example GPUs.

Vendor Tools

Vivado HLS [212] is provided by XILINX for the development of hardware accelerators using a high-level language such as C/C++, SystemC or OpenCL. The tool uses a transpiler to convert the source code into an RTL implementation that can be synthesized for XILINX FPGAs. The developer must optimize and improve the source code using directives (`pragmas`) to obtain efficient code for the FPGA.

In XILINX's classic toolflow, developers of computer vision algorithms would need three tools to program a heterogeneous system consisting of an ARM CPU and an FPGA when using HLS. First, the developer would need to implement the computer vision ACs. The individual ACs can be implemented either in VHDL/Verilog using Vivado or in C/C++/OpenCL using Vivado HLS. An IP-core is then generated from the AC and integrated into the final system design. The system (block) design is created using Vivado. The block design simplifies the reuse of IP-cores and their integration into the embedded system design. Vivado is also used to create the bitstream for the FPGA. The ARM CPU is tightly coupled to the FPGA and is programmed using XILINX's SDK. It can be used to load the bitstream onto the FPGA and debug the system.

XILINX also provides the SDSoC [213] tool to further abstract hardware-specific implementation details from the developer. It supports heterogeneous embedded systems, consisting of an ARM CPU and an FPGA, such as the Zynq [100] system. The advantage of SDSoC is that developers of computer vision algorithms can implement a complete design using one software tool. The tool compiles software functions written in C/C++ with Vivado HLS to ACs and generates a complete hardware system based on the selected platform, including DMA, interconnects, hardware buffers and more. It also creates the software stack that controls the AC from the ARM CPU. An FPGA engineer can still create a new hardware platform that can be used in SDSoC, for example, to integrate I/Os.

A similar tool from XILINX is called SDAccel [214]. This tool was created to program FPGAs, which are connected to x86-based systems, using the OpenCL programming model. Using OpenCL, the FPGA can be used like any other OpenCL device in the system, for example a GPU or CPU. The integration of the different vendor drivers is done via Makefiles. This simplifies the programmability of heterogeneous systems. Like SDSoC, SDAccel creates the hardware platform including data movers and the software stack that controls the implemented ACs. The OpenCL kernel itself can be written in OpenCL or in C/C++. The main difference to other compute devices is that the FPGA bitstream must be generated before the OpenCL program is executed.

The OpenCL SDK [210, 211] from Intel, formerly Altera, provides software developers with an environment that abstracts away the underlying hardware details while enabling an efficient use of FPGA resources without requiring knowledge of an HDL. The SDK provides the programmer with a higher level of abstraction and the development tools postpone costly hardware compilations to the end of the design process [215].

Janik et al. [211] present an overview of Intel's OpenCL SDK. The process of speeding up an algorithm is not simple, as it requires knowledge of the behavior and structure of the OpenCL standard, as well as an understanding of parallel computing paradigms. A major advantage offered by the SDK compared to HDL tools is the fast creation of a functional correct implementation. However, the initial design does not achieve the same speedup. Once the functional model is created, the optimization of the program becomes the focus of the task. This is quite different from traditional HDL designs, where you can't determine if the design is functional correct until a later stage in the design process. The authors show that it is possible to achieve a high speedup when comparing their optimized FPGA implementation to a CPU implementation for a 1024×1024 matrix multiplication.

Research Tools

Besides the commercial tools, there are also university research tools, such as ROCCC [216] and LegUp [217], to efficiently implement FPGAs with high-level languages. Villarreal et al. [216] presented ROCCC 2.0, a C-to-VHDL compilation tool to generate ACs for FPGAs. It allows the user to define self-contained modules that can be reused. A module can be imported as C code, VHDL code or as a netlist. Creating and importing hardware modules in C is done without adding explicit commands to the C code and allows the creation of ACs without requiring the low-level details of an HDL. The proposed compiler produces an efficient pipelined hardware based these modules. They show that the modules generated by the ROCCC compiler are competitive with handwritten VHDL code in terms of clock frequency, while productivity increases by 15.

Canis et al. [217, 218] presented LegUp, one of the first and most known research tools. It is a SoA HLS compiler which automatically generates high-performance FPGA hardware from C++ code. It supports hardware synthesis using the well-known multithreading approaches Pthreads and OpenMP. Without requiring any code changes, it can produce multiple hardware cores that can execute concurrently. Although it was one of the first HLS tools, it is still being developed further, as shown in the release version [219] of 15.09.2020. Just recently, a well-known vendor of microchips, called Microsemi, joined the toolchain. Like the vendor tools, LegUp HLS offers a rich set of user-constraints to specify the desired hardware architecture.

- Outer loops can be pipelined, or inner loops can be unrolled, to improve performance.
- Functions with a FIFO dataflow that have streaming inputs and outputs can be parallelized.
- C++ arrays can be partitioned into registers or BRAMs to achieve a better memory bandwidth.
- Larger hardware operations such as floating-point operations can be shared.

Windh et al. [220] focused on an investigation of five HLS tools to improve the productivity of efficient FPGA code development. They compare several vendor tools, such as Vivado HLS [212] from XILINX and the OpenCL SDK [210] from Intel, with various research tools such as Bluespec [221], ROCCC [216], and LegUp [217]. They describe the user interaction of these tools, show how the code is expressed and compiled, and discuss the resulting resource usage using an image dilatation filter and an AES (Advanced Encryption Standard) algorithm.

Nane et al. [222] have presented a comprehensive survey and categorization for commercial and academic hardware compilers and HLS-based toolchains. Figure 2.22 shows the classification of different HLS tools, including only some of the tools mentioned in the survey paper. They experimentally evaluated three academic HLS tools (BAMBU [223], DWARV [224], and LegUp [217, 218]) against a commercial tool. The objective was to provide a fair comparison of HLS tools, even though they were created in different compiler frameworks and target different FPGAs families. The results show that each HLS tool can significantly improve performance with benchmark-specific optimizations and constraints. However, software engineers must consider that optimizations to achieve high performance, like enabling loop pipelining and removing control flow, are significantly different from software-oriented optimizations, like data reorganization for cache locality. The performance results showed that academic and commercial HLS tools are not that far apart from each other in terms of quality, and that no single tool provided the best results for all benchmarks. In general, the commercial compiler supports more features while being more robust than the academic tools. For example, they allow multiple input and output languages, customization of generated kernels in terms of interface types, memory bank usage and throughput.

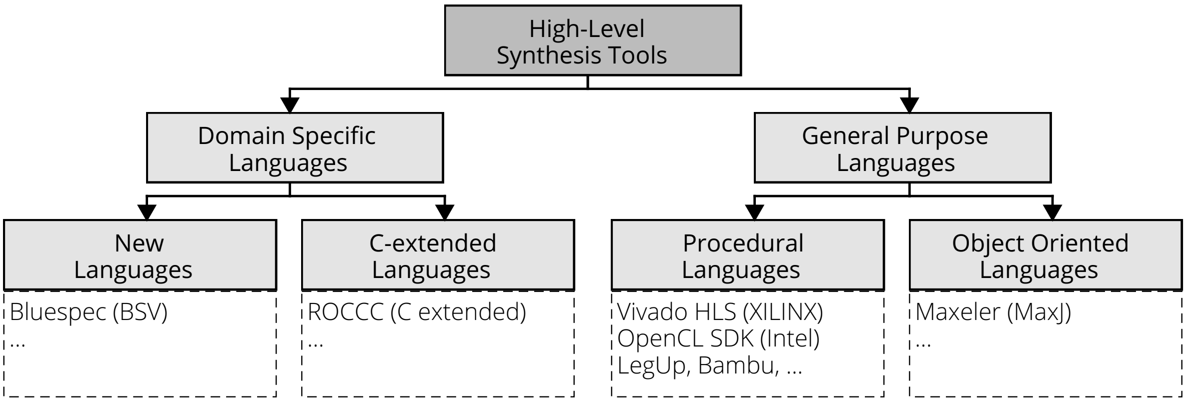


Figure 2.22: Classification of HLS tools based on the input language [222].

Case Studies and Analysis Tools

Winterstein et al. [225] presented a case study for Vivado HLS [212] using a K-Means clustering and a filtering algorithm. The K-Means clustering algorithm is dataflow-centric with regular control flow and memory accesses, while the filtering algorithm uses dynamic memory management based on recursive traversal of a pointer-linked tree structure. The performance gap between the HLS and RTL implementations of the K-Means clustering algorithm is approximately a factor of 2 in terms of area-time product, which is a remarkable result considering the large difference in design time. For the filtering algorithm, the resource consumption of both implementations is similar, but the latency is initially decreased by a factor of 30. They were able to improve latency by a factor of 8 by applying code transformations to partition and privatize data structures accessed via pointers to promote parallelization and enable pipelining of the loop.

Hosseinabady et al. [226] present an automation technique to efficiently map an OpenCL description onto an ARM-FPGA-based SoCs from XILINX. They propose a work-group synthesis technique to automatically divide a large dataset into small blocks and map the OpenCL description to the FPGA, which does not have enough space to store a large dataset. The provided framework is based on a transpiler that transforms the OpenCL kernel into C/C++ code. Vivado HLS [212] is then used to synthesize the code. They define the data access patterns of the OpenCL kernels, including intra- and inter-work item access patterns and further optimizations for parallelism and memory usage. Their OpenCL implementation shows a speedup of up to 89.8 % compared to Intel's OpenCL running on a Cyclone V.

Hill et al. [215] compare traditional VHDL application development with the OpenCL design flow of the OpenCL SDK of Intel for image processing kernels. The OpenCL SDK provides the programmer with a higher level of abstraction and the development tools postpone costly hardware compilations to the end of the design process. They implemented three algorithms for their case study: (1) the Sobel kernel (2) the Canny edge detector [18] and (3) the SURF feature extractor [40]. On the one hand, the SDK increased productivity by a factor of 6. On the other hand, the VHDL designs required 59 % to 70 % less hardware resources, while maintaining similar timing constraints (255 MHz < f_{max} < 325 MHz). The difference in fps for the OpenCL and VHDL kernels is between 2 % and 10 % on a Stratix-V D8. In addition, they were able to demonstrate portability of the OpenCL code to two other FPGA platforms without further modifications, while achieving similar performance in terms of operating frequency and resource utilization.

Wang et al. [227] presented an FPGA-based performance analysis framework for OpenCL applications on Intel FPGAs. They proposed a performance model that detects the main architectural features, predicts the performance with different optimization combinations, and identifies the bottlenecks of the OpenCL kernel code. They demonstrate the efficiency of their model with multiple use cases (matrix multiplication, K-Means and MapReduce) by considering many combinations. The results show that the proposed model has a high accuracy in predicting the performance acceleration of individual optimizations and can achieve up to two orders of magnitude speedup over a baseline implementation. The tool provides programmers with understandable metrics and guides them in optimizing code to address performance bottlenecks.

In the above-mentioned work [215, 226, 227, 225], there were unfortunately no comparisons to optimized implementations of other platforms, such as multicore CPUs or GPUs. Ayat et al. [228] looked at OpenCL using the Intel SDK and implemented a Sobel filter as a use case. Their implementation was compared with a GPU and a multicore CPU implementation for different image and kernel sizes. The FPGA has the advantage over the GPU of providing good data reusability for image processing applications by implementing custom buffers. Their results also show that each device needs its own optimization, since the same code will not perform very well. They show that the kernel size does not increase the computation time for FPGAs as much as for GPUs or CPUs. However, their timing results seem to be quite slow. Additionally, they do not show any power measurements of the different platforms.

2.5.3 FPGA-based OpenVX Tools

OpenVX [32, 229] is more than just a library thanks to its graph-based approach and memory management. OpenVX is a low-level programming framework domain that provides software developers with efficient access to computer vision hardware acceleration. It has been designed to maximize functional and performance portability across different compute devices, such as CPUs, DSPs, GPUs or ASICs, in the embedded and HPC domain. Some vendor implementations rely on APIs like OpenCL or OpenGL to target a device, while others build directly on top of the device drivers. Figure 2.23 gives an overview of the layers in the OpenVX framework [229]. The key features of OpenVX are:

- a library of predefined and customizable vision functions.
- a graph-based execution model for combining functions that enable both task- and data-independent execution.
- a set of memory objects that abstract physical memory.

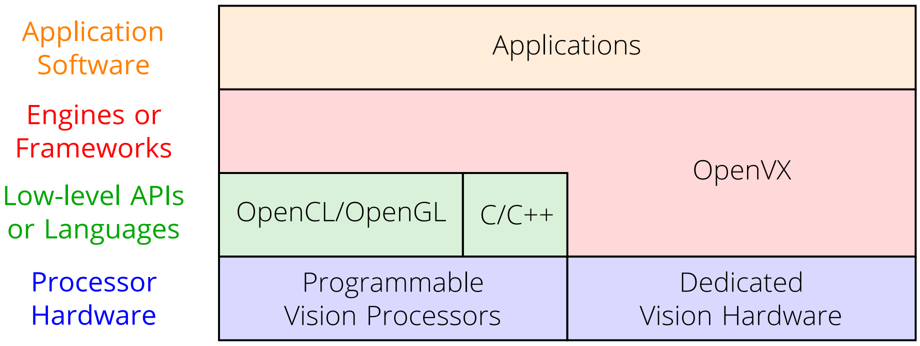


Figure 2.23: Overview of the layers in the OpenVX framework [229].

In contrast to the various low-level APIs, such as OpenCL, OpenVX frees the programmer from memory management and application scheduling. Implementation details are limited to edge and node creation and parameter tuning. This leaves the developer of an OpenVX compliant toolchain with the task of optimizing the graph in terms of execution time, for example by reducing memory transfers and creating a schedule. There are many implementations from different vendors, such as AMD, NVIDIA, Synopsys, or QUALCOMM. However, it is not so easy to use different vendor libraries at the same time.

Research Tools

Several researchers have proposed OpenVX based frameworks to facilitate the development process of computer vision applications for different systems [230], but only a few for FPGAs. The most promising OpenVX frameworks that use FPGAs are ADRENALINE [231], JANUS [232], JANUS SVP [233], AFFIX [234] and HIPA^{CC}VX [235].

Tagliavini et al. [231, 236, 237] proposed **ADRENALINE**, which contains techniques based on graph analysis and image tiling aimed at accelerating the execution of image processing applications represented as OpenVX graphs. Its architecture contains RISC (Reduced Instruction Set Computer)-based PUs, which share multi-banked data memory, and are connected by a logarithmic interconnection. Its low-level API is OpenCL, and the system is evaluated using the STHORM architecture connected to a XILINX Zynq system that uses the FPGA as interconnection to bridge to the ARM core.

Omidian et al. [232, 238] presents **JANUS**, which is a compilation system for OpenVX that can analyze and optimize the compute graph to find area/throughput trade-offs and map onto FPGAs. It also combines module selection and replication methods as well as changing tile size with node combining and splitting. Omidian et al. [233] expanded this approach for a SVP, including a runtime environment, a library of OpenVX C/C++ kernels and a library of prebuilt VCIs (Vector Custom Instructions) for the FPGA. The user selects the vector size, the scratchpad size, and the size of the PRR.

Taheri et al. [234] presented a framework for turning a high-level OpenVX graph specification into an FPGA implementation, which has been improved in the **AFFIX** [239] framework. AFFIX receives an algorithm representative DAG in a textual format developed by a user including the desired SIMD size. It outputs a heterogeneous implementation of the vision algorithm using Intel's OpenCL SDK. It adds high and low-level optimization methods to improve the efficiency.

While **HIPA^{CC}** [186, 188, 187], introduced in Section 2.4.2.4, allows portability and efficient generation of high-performance accelerator code, application scheduling is left to the programmer. A recent addition called **HIPA^{CC}VX** [235] aims to fuse the graph-based approach and execution model of OpenVX with the code generation implemented in HIPA^{CC}. They extended the OpenVX specification such that programmers can register HIPA^{CC} kernels as custom nodes to OpenVX programs. The HIPA^{CC}VX framework consists of an OpenVX graph implementation and routines that verify and optimize the input OpenVX applications. It generates device specific code for a target platform using the HIPA^{CC} code generation.

In total, 9 OpenVX functions are implemented as OpenCV kernels since they cannot be fully described with HIPA^{CC}. Excluding these functions, they list 46 functions in their table. It is hard to tell if all these functions have been implemented. Thanks to HIPA^{CC}, HIPA^{CC}VX can address different architectures, such as CPUs, GPUs, or FPGAs. However, in the evaluation, they are only addressed one at a time. This means that there is no special mapping and scheduling in comparison to this thesis. Due to the various kernel-level optimizations, HIPA^{CC}VX achieves very good performance.

Comparison

Table 2.10 compares the frameworks described above with each other and with the proposed `DECISION` framework, which integrates the proposed `HiFlipVX` library. `DECISION`, JANUS, AFFIX, and HIPA^{CC}VX provide the best performance in terms of computation time because they use optimized ACs rather than generic PUs. They are not as bandwidth constrained as the other approaches because they can transfer data directly between different computer vision functions. Using the Canny edge detector as an example application, ADRENALINE achieves 109 Mpx s^{-1} running at 16 cores at 433 MHz, and JANUS SVP achieves 21 Mpx s^{-1} On the contrary, `DECISION` and AFFIX achieve about 2 Gpx s^{-1} thanks to vectorization.

JANUS SVP seems to be a good option for software programmers, since almost no knowledge about the hardware is required. The user selects only the vector size, the scratchpad memory size, and the size of the PRRs. HIPA^{CC}VX and `DECISION` have the advantage of using an OpenVX compliant interface. In contrast, AFFIX requires the user to use OpenCL constructs and channels. HIPA^{CC}VX, AFFIX and `DECISION` provide a large set of image processing functions, which are also available as an open-source library. In contrast, the `HiFlipVX` library also supports CNNs and feature detection functions. Furthermore, it enables numerous optimization and tuning possibilities through its high number of parameters. In addition, `HiFlipVX` has even been used for large applications such as MobileNets [21] and AKAZE [20] with up to 163 ACs.

Table 2.10: Comparison of FPGA-based OpenVX frameworks. SVP (Soft Vector Processor), VCI (Vector Custom Instruction)

	ADRENALINE	JANUS	JANUS SVP	AFFIX	HIPA^{CC}VX	`DECISION`
Publication	Tagliavini 2015 [231, 236, 237]	Omidian 2017 [232, 238]	Omidian 2018 [233]	Taheri 2019 [234, 239]	Özkan 2021 [235]	Proposed 2022 [27]
Architecture	ARM, Sthorm, intercon. via FPGA	FPGA, AC	ARM, SVP (MXP), VCI (AC)	Intel FPGA, AC	CPU, GPU, FPGA	CPU, GPU, FPGA
Inter-connection	logarithmic between PUs	-	SVP to VCI (3 ports)	-	-	NoC, -
Library	41 SW functions, set for Sthorm	Sobel, Canny, Harris	25 kernels (SVP & HLS)	46 functions	max. 55 functions	66 feature, image, CNN functions
Abstraction	OpenCL as backend	HW knowledge	pure software programmers	OpenCL constructs & channels	OpenVX compliant	OpenVX compliant
Performance	medium	high	low	high	high	high
PRRs	-	-	✓	-	-	(✓)
Heterogeneous	-	-	-	-	-	✓

Of all the approaches, JANUS SVP is the only one that considers PRRs. However, the architecture and model of the `DECISION` framework provide the opportunity for PRRs. In addition, `DECISION` builds a flexible and adaptable NoC-based infrastructure with a minimal runtime system. However, this flexibility does not come with the performance drawbacks of JANUS SVP or ADRENALINE. In addition to FPGAs, `DECISION` and HIPA^{CC}VX can also address CPUs and GPUs in a heterogeneous system. However, only `DECISION` supports the simultaneous use of heterogeneous devices. JANUS, AFFIX and HIPA^{CC}VX show different approaches to optimize kernels or the OpenVX graph, which can be used to complement this work. In contrast, this work proposes `APARMAP`, which can distribute applications to mesh-like and partition-based FPGA clusters.

2.5.4 OpenCL-based Tools

Due to the good usability of OpenCL, it is available on a wide range of platforms. OpenVX-based frameworks such as AFFIX [239] and ADRENALINE [237] also use OpenCL as a low-level API. However, they only target a specific platform and are often not general enough and do not have a comprehensive runtime system. Other frameworks, such as HIPA^{CC}VX [235], can access a variety of platforms, but do not take advantage of the wide range of application distribution options.

This section deals with toolchains that are either based on OpenCL or integrate OpenCL into their concept. Section 2.4.2.1 already dealt with libraries such as ViennaCL [172] and VexCL [171], which are based on OpenCL, or OpenCV [178], which integrates OpenCL for acceleration. These libraries often try to abstract the host code and provide implemented kernel functions. However, a toolchain includes more components such as a runtime system, memory management, scheduler, or hides data transfers. The following subsections will discuss two frameworks that take a more comprehensive approach and compares them with this thesis. These are SnuCL [240, 241] and StarPU [242, 243].

Research Tools

Jungwon Kim et al. [240] proposed SnuCL, an OpenCL framework for heterogeneous CPU/GPU clusters. Junghyun Kim et al. extended SnuCL in SnuCL-D [241] and SnuCL-Tr [244]. SnuCL allows an application to utilize compute devices of a compute node as if they were in the host node. Standard OpenCL functions are used to manipulate memory objects. The SnuCL framework then maps the buffers in the device memories of the respective devices. In doing so, it minimizes the overhead of copying memory and takes care of memory consistency. They use a parallel loop scheduling algorithm to distribute a single kernel unto multiple devices [245]. They use code transformations to detect buffers written by a kernel and to distribute kernel code. The SnuCL runtime uses OpenMPI as a lower-level communication API. The resulting system can accelerate eleven different benchmarks linear to the number of devices used, except for programs that are limited by their sequential parts or memory transfer delays. For large clusters, SnuCL can cause performance degradation due to its centralized task scheduling model. SnuCL-D extends the framework by a distributed approach that replicates the entire OpenCL program on each device to reduce overhead.

Augonnet et al. [242] proposed StarPU, a task programming API for hybrid CPU/GPU architectures. Their approach is based on a task-based model represented as DAG. It targets and schedules heterogeneous systems including CPUs and GPUs, and manages data dependencies and data transfers between different devices. There are several functions in its API for executing and registering kernels and buffers. In addition, the library contains functions to initialize and terminate the StarPU environment. Buffers must be registered to pass control to the runtime system and unregistered to return control back to the user. Kernels are registered with their buffers, access modes (read/write), and targeted architectures. StarPU has its own C prototype to create these kernels. This prototype is also needed as a wrapper to execute OpenCL and CUDA kernels. A kernel is submitted as a task in a non-blocking manner. The runtime manages all dependencies with the preceding and succeeding tasks. The task scheduler considers data transfers and benefits from caching techniques and data reuse. It can be used with a specific strategy that allows customizing the policy for different use cases. To improve the schedule, the programmer can provide additional hints about task priority and set weights for tasks and data transfers. An extension called StarPU-MPI [243] uses MPI to enable data transfers for data dependencies across the boundary of compute nodes.

Comparison

Table 2.11 lists the related work and shows the benefits and drawbacks of each approach. A low level of abstraction requires boilerplate code which is typical for OpenCL programs. In contrast, the OpenVX standard allows a high-level description of a graph, hiding the implementation details. Heterogeneous refers to the ability of the framework to utilize multiple heterogeneous devices simultaneously. The tables shows that no other framework than `DECISION` provides a high abstraction level for acceleration of computer vision pipelines on heterogeneous systems supported by a memory model, minimal runtime system, automatic device and kernel profiling, and scheduling and mapping.

Table 2.11: Comparison of OpenCL-based toolchains.

	OpenCV 2008 [178]	ViennaCL 2016 [172]	SnuCL 2016 [244, 241]	StarPU 2011 [242, 243]	AFFIX 2019 [239]	HIPA^{CC}VX 2021 [235]	`DECISION` 2022 [27]
CPU	✓	✓	✓	✓	✗	✓	✓
GPU	✓	✓	✓	✓	✗	✓	✓
FPGA	✗	✗	✗	✗	✓	✓	✓
Abstraction	low	low	low	mid	low	mid/high	high
Memory model	✗	✗	✓	✓	✗	✓	✓
Profiling	✗	✗	✗	✗	✗	✗	✓
Scheduling	✗	✗	✗	✓	✓	✓	✓
Heterogeneous	✓	✓	✓	✓	✗	✗	✓

Compared to SnuCL, the focus of this work is on optimizing at design time rather than at runtime. This has the advantage of a minimal overhead, which is negligible compared to that of OpenCL. The scheduling algorithms of this work focuses on scheduling and mapping task graphs to different devices rather than a single kernel on multiple devices as done by SnuCL. One reason for this approach is that compared to their work, this work also integrates FPGAs. FPGAs play their full potential over GPUs if multiple functions are pipelined to enable TLP and drastically reduce bandwidth consumption. Nevertheless, their approaches for work-item and work-group optimization could be incorporated into the scheduling approach of this work to enhance the design space. Additionally, SnuCLs inter compute node communication approach could be a reasonable extension for this work. In terms of programmability, this work has a big advantage over SnuCL because it uses OpenVX as a frontend and thus has a standardized interface. This allows the integration of not only the `HiFlipVX` library developed in this work, but also libraries from other vendors.

In contrast to StarPU work, this work integrates FPGAs into the toolchain. By adding FPGAs, more efficient and flexible systems can be created. At the same time, new concepts are required due to the fundamental difference in architectures. For example, through pipelining at function level, FPGAs need a more complex scheduler. A further difference is that in this work kernel libraries for object detection are developed and integrated. Using a standard API, like OpenVX, further libraries of different vendors can be integrated. The OpenVX concept allows this work to make many optimizations at design time. For example, compared to their work, the mapping and scheduling in this work is calculated at design time. Design time optimizations allows many computations to be done in advance, thus reducing the overhead of the runtime system. However, it also reduces the number of API function calls of the user, such as registering buffers. In addition, the wrappers for the kernel functions in this work are generated automatically and do not need to be created by hand. The different parameters of the object detection library that are set in this wrapper allow further optimizations through fine tuning at design time.

3 `HiFlipVX`: Object Detection Library

This chapter focuses on the design and implementation of the various object detection algorithms and individual functions covered in this thesis. The background information about the more complex algorithms can be found in Section 2.1. Based on the research and the exploration of the different algorithms and their functions the `HiFlipVX` library was developed.

`HiFlipVX` enables a faster development of computer vision applications on FPGA-based embedded or HPC systems. The name stands for **Hi**gh-Level Synthesis **F**PGA **L**ibrary for **I**mage **P**rocessing. It is an open-source C++ and HLS-based FPGA library for object detection and is available at [246]. It stands out due to its high parameterizability, performance, portability, and resource efficiency, and is partly based on the OpenVX standard. OpenVX is an open, royalty-free standard for cross-platform acceleration of computer vision applications [32]. `HiFlipVX` contains 66 computer vision functions with 42 based on the standard and 24 developed within this research. The additional functions were mainly extracted from different algorithms discussed later in this chapter and transformed to generic and reusable library functions. They also include functions that target the streaming capability of FPGAs, such as data-width converter, multicast, scatter and gather functions.

Section 3.1 describes the implementations of the individual library functions. These have been divided into pixelwise, filter, conversion, analysis, feature, and neural network functions. Section 3.2 describes the more complex algorithms implemented using VHDL, `HiFlipVX`, or both. These include the FAST [17] corner detector, Canny [18] edge detector, ORB [19] feature detector, AKAZE [20] feature detector, FREAK [24] feature descriptor, and MobileNets [21] neural network. Section 3.3 evaluates the individual functions and algorithms in terms of their performance, resource utilization, accuracy, and scalability [12, 13]. Furthermore, different combinations of feature extraction algorithms were compared in terms of their repeatability and performance [22]. In this thesis, the most promising combinations were further developed, with significantly better results than the other combinations. Two of the feature detectors and one feature descriptor were selected based on their performance and characteristics for embedded and HPC systems, implemented using `HiFlipVX` [25, 26, 23], and compared with the SoA. In addition, the neural network extension of the library has been evaluated based on its accuracy, performance, scalability, and resource efficiency [14]. Section 3.4 summarizes the implementation and evaluation of this chapter.

3.1 `HiFlipVX` Library Functions

The `HiFlipVX` library contains 66 computer vision functions divided into six groups.

- **Image Pixelwise**: Operates on its input images pixel by pixel, e.g., for arithmetic operations.
- **Image Filter**: Operates on an input window of pixels to compute an output pixel.
- **Image Conversion**: Converts image properties, like color, depth, width, and resolution, or connects vision functions.
- **Image Analysis**: Analyzes all pixels of an input image, e.g., to calculate its histogram or standard deviation.
- **Feature**: Extracts a set of features or operates on them and includes complete algorithms.
- **Neural Network**: Functions used in common CNN algorithms.

In addition to the OpenVX standard, most functions in `HiFlipVX` support different vectorization options (1, 2, 4 and 8) and additional data types (8 bit, 16 bit and 32 bit signed/unsigned integers) for their input and output parameters. Internally, the library uses bitmasks to create data types of arbitrary precision, since they are not part of the C++ standard. This thesis did not use the arbitrary precision data types from XILINX, to be more vendor independent. For bit-widths above 64 bit and to perform vectorization, the library uses an own template-based data type. Due to the use of directives and macros for optimizations, `HiFlipVX` can also be executed on a CPU outside of the XILINX environment, for an easier testing and verification. To further increase the usability, the library uses static assertions to throw an error if parameter values or data types with undefined behavior are used. The following subsections describe the six different function groups from the `HiFlipVX` library separately.

3.1.1 Image Pixelwise Functions

Table 3.1 shows the implemented pixelwise functions, which have several characteristics in common. They perform their operations on the input image(s) pixel by pixel. The bit-width of the input and output images can be 8 bit, 16 bit and 32 bit, but needs to be the same. The pixel values can be unsigned or signed. Independent of the bit-width, 1, 2, 4 and 8 pixels can be computed in parallel in a vector. Due to the template-based implementation, higher vector sizes would also be possible. A shared, template-based function is used to implement all pixelwise operations for an easy expandability of new functions. It includes the verification of data types and template parameters, reading the input vector(s) and writing back the result. The function is pipelined to exploit temporal parallelism and the operation is executed on each element of the vector in parallel. The library performs all arithmetic operations using fixed-point numbers, to reduce resource consumption while maintaining precision.

If the result of an arithmetic operation cannot be represented with the chosen bit-width, overflow or underflow occurs. Therefore, the library contains different policies for the functions that need them. On the one hand, overflow can be ignored (`wrap`) or the minimum/maximum representable number is used (`saturate`). On the other hand, underflow can be ignored (`truncated`) or rounded to the nearest integer value. Resource efficient operations for saturation are chosen depending on the type of arithmetic operation and the signedness of the values.

Table 3.1: `HiFlipVX` image pixelwise functions. Non-standard are marked with ★.

Name	Operation
Data Object Copy	$out(x,y) = in_1(x,y)$
Bitwise AND	$out(x,y) = in_1(x,y) \wedge in_2(x,y)$
Bitwise XOR	$out(x,y) = in_1(x,y) \oplus in_2(x,y)$
Bitwise OR	$out(x,y) = in_1(x,y) \vee in_2(x,y)$
Bitwise NOT	$out(x,y) = \overline{in_1(x,y)}$
Arithmetic Addition	$out(x,y) = in_1(x,y) + in_2(x,y)$
Arithmetic Subtraction	$out(x,y) = in_1(x,y) - in_2(x,y)$
Absolute Difference	$out(x,y) = \lvert in_1(x,y) - in_2(x,y) \rvert$
Weighted Average	$out(x,y) = (1 - \alpha) \cdot in2(x,y) + \alpha \cdot in1(x,y)$
Magnitude	$out(x,y) = \sqrt{in_1(x,y)^2 + in_2(x,y)^2}$
Pixelwise Multiplication	$out(x,y) = in_1(x,y) \cdot in_2(x,y) \cdot \alpha$
Constant Multiplication ★	$out(x,y) = in_1(x,y) \cdot \alpha$
Min	$out(x,y) = \begin{cases} in_1(x,y) & in_1(x,y) < in_2(x,y) \\ in_2(x,y) & in_1(x,y) \geq in_2(x,y) \end{cases}$
Max	$out(x,y) = \begin{cases} in_1(x,y) & in_1(x,y) > in_2(x,y) \\ in_2(x,y) & in_1(x,y) \leq in_2(x,y) \end{cases}$
Phase	$\theta(x,y) = atan2(in_1(x,y), in_2(x,y))$ $out(x,y) = \begin{cases} \theta(x,y) + \pi & \theta(x,y) < 0 \\ \theta(x,y) & \theta(x,y) \geq 0 \end{cases}$
Threshold	$out(x,y) = \begin{cases} 1 & in_1(x,y) > t_0 \\ 0 & in_1(x,y) \leq t_0 \end{cases}$ $out(x,y) = \begin{cases} 0 & in_1(x,y) > t_1 \\ 0 & in_1(x,y) < t_0 \\ 1 & (in_1(x,y) \leq t_1) \wedge (in_1(x,y) \geq t_0) \end{cases}$

There are two functions in the table that are multiplied by the constant value α. Its value is set as a template parameter at synthesis time, which leads to an optimized resource usage. For example, the library uses a shift operation instead of a multiplication when the scalar value is a multiple of two. This saves resources for the multiplication operation, since fixed shifts can be performed by rewiring. However, right shift operations performed on negative numbers can cause a rounding error. For example, –3 shifted by 1 would be –2, but –3 divided by 2 would be –1. Therefore, the library performs a shift operation that produces the same results as the division. The value of α is represented as a fixed-point value with a 16 bit fraction for arithmetic operations.

To reduce the resource consumption of the magnitude function, the library contains an HLS-based integer square root function as shown in Listing 3.1. Here, N is the output bit-width that is half of the input bit-width. For each output bit, two additions and one comparison are performed. The shift values are signal connections, computed at synthesis time and the

OR-operation simply concatenates the result bits. An integer square root is used to reduce resource utilization, since for lower bit-widths the library does not need a higher precision to comply with the OpenVX standard. Only for a 64 bit data type double precision floating-point is needed to comply with the standard.

```
  A1 = 0; // Intermediate result
  A2 = 0; // Square of intermediate result

  for (n = N - 1; n < N; n--)
#pragma HLS unroll

      // (A1 + B1)^2 and add new bit at position n
      B1 = 1 << n;
      B2 = B1 << n;
      AB = A1 << n;
      A2_next = A2 + B2 + (AB << 1);

      // Store if A2_next does not exceed value
      if (A2_next <= input)
          A1 |= B1;
          A2 = A2_next;
```

Listing 3.1: Integer square root function. Each stage computes 1 bit of the resulting vector.

As shown in Table 3.1 the phase is calculated using the 2-argument inverse tangent $atan2(x,y)$. The library uses the CORDIC (COordinate Rotation DIgital Computer) algorithm with a 16 bit fraction part, to reduce the resources consumption while maintaining the desired accuracy. Dinechin et al. [247] show that this accuracy is sufficient for many cases. The range of the resulting angle is between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$. For negative angles, π needs to be added to the result. Finally, the value is quantized using a parameterizable constant value. For example, a quantization of four would divide the angles into north, east, south, and west. Therefore, the angle is rotated in advance according to the quantization.

The threshold function takes an input image and creates a Boolean image using the threshold values (t_0 and t_1). As shown in Table 3.1, two different threshold types exist. Default values for the output pixels are defined by the standard to represent a Boolean. If the output format is an unsigned integer, `true` is equal to the maximum possible value. If it is a signed integer, `true` is equal to -1. In both cases `false` is equal to zero.

3.1.2 Image Filter Functions

One of the most important concepts in the field of signal processing is spatial convolution. Spatial convolution is also considered as image filtering. This filtering could be smoothing, differentiation, edge detection, calculating the mean, etc. To calculate an output pixel, an operation is performed on a $n \times n$ large window within each input image. In many cases this operation is a multiplication with a kernel matrix, where the elements of the window and the kernel matrix are multiplied pixel by pixel.

Table 3.2 shows the implemented filter functions that share several common features. Therefore, `HiFlipVX` uses a generic base structure (function) for all its filters. A parameter selects the correct kernel function at compile time. This approach simplifies the creation and implementation of new filter functions. To create a new filter, it needs a kernel and a wrapper function. The second one selects the required parameters of the base structure. This way all optimizations, such as vectorization and line buffering, can be reused.

Table 3.2: `HiFlipVX` image filter functions. Non-standard are marked with ★.

Box	Gaussian	Non-Maximum Suppression
Sobel	Convolve	Segment Test Detector 7 × 7 ★
Erode	Scharr 3 × 3 ★	Determinant of the Hessian 3 × 3 ★
Dilate	Hysteresis ★	Fast Explicit Diffusion 3 × 3 ★
Median	Conductivity 3 × 3 ★	Oriented Non-Maximum Suppression 3 × 3 ★

In the following, the different parameters of this base structure are described in more detail. Then the generic sliding window approach of the base structure is presented. To implement filter functions on FPGAs in an effective way, a sliding window approach is needed, as shown in Figure 3.1. Then there is a description of an approach for separable filters that can be used to save resources when certain symmetries are present in the kernel matrix. Finally, the implementation of certain kernel functions is described.

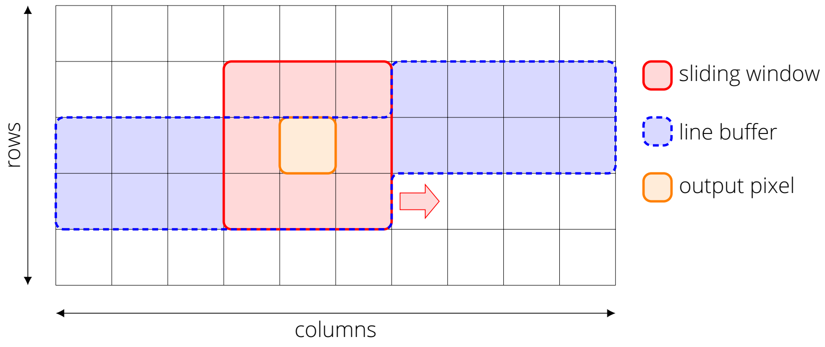


Figure 3.1: Sliding window approach for a 3 × 3 kernel.

Parameter

The base structure can be configured generically for one to two input images and one to two output images. In addition, there are numerous parameters, like the image rows (I_R) and columns (I_C), that are resolved at compile time to provide high variation and optimize resources at the same time. All the constants and variables, which are used within this section, are shown in Table 3.3. Within the base structure, most of the constants are checked for validity at compile time. Additionally, further restrictions of these parameters are made in the wrapper functions. The most used template parameters are described in more detail below.

Kernel function: Optimized kernel functions operate on an input window and compute a single output pixel. Beside these kernel functions there is also a forwarding unit, which forwards one of the input images to one of the outputs. This can be used to reduce the number of needed buffers in specific algorithm implementations. A separate kernel function

Table 3.3: Constants (uppercase) and variables (lowercase) used in `HiFlipVX`.

I_C	image columns	LB_W	bit-width of all line buffers
I_R	image rows	T_C	trip count
I_W	image pixel bit-width	P_I	pipeline interval
I_{MAX}	max possible pixel value	P_D	pipeline depth
K_S	kernel size	P_F	FIFO pipeline depth
K_R	kernel radius	t_0	threshold (lower border)
K_σ	kernel scale	t_1	threshold (upper border)
SK_σ	scaled kernel size	c_W	window column
WC_T	window columns total	x_V	vectorized x-coordinate
WC_L	window columns left	x	x-coordinate
WC_R	window columns right	y	y-coordinate
O_C	loop column overhead	v	pixel value
V_S	vector size	r	response value

(or forwarding unit) is needed for each output image of the filter. For example, the Sobel filter calculates the derivatives in x and y-direction using two different kernel functions that operate on the same input window.

Kernel matrix: For each kernel function, a kernel matrix of size $K_S \times K_S$ can be passed. The library generates kernel matrices of variable sizes for different functions, such as the Gaussian or Sobel, at compile time.

Kernel size (K_S): Table 3.2 shows the kernel size of a function if it is fixed. All other filter functions support kernel sizes of 3, 5, 7, 9 and 11. This thesis could not find a common solution of a larger kernel size for the Scharr filter. Both the conductivity and the DoH functions use the Scharr kernel to calculate their derivatives.

Data types: Like for the pixelwise operations, the filter functions offer different data types for their input images, output images and kernel matrices (8 bit, 16 bit and 32 bit). Depending on plausibility, there are restrictions on the signedness for the individual functions.

Vector sizes (V_S): Like the pixelwise operations, all filter operations offer different vector sizes (1, 2, 4 and 8). Besides the resource limitation, the maximum size of a vector is determined by the maximum possible bit-width of the interface, which is limited by the HLS tool (1024 bit for Vivado HLS).

Kernel scale (K_σ): It enlarges the kernel of a filter by increasing the distances between the elements of the kernel matrix and filling the gaps with zeros. For example, a scaled kernel of size (SK_σ) is used for the Scharr filter in the AKAZE algorithm. Using the kernel matrix and K_σ, the scaled kernel matrix is automatically generated at compile time. The maximum value for SK_σ is 11, which is the same as for K_S.

$$SK_\sigma = (K_S - 1) \cdot K_\sigma + 1 \tag{3.1}$$

Step size: The step size or stride specifies by how many pixels the sliding window must be moved to calculate the next valid output pixel. Its value has an influence on the output resolution and its vector size.

Border type: Every filter supports three different border behaviors. Values beyond borders can be undefined, constant zero, or the replicated border pixels. In most feature detection and description algorithms, this thesis achieved better results using replicated border handling.

Filter Structure

The basic filter function has a pipelined structure that uses line buffers and a sliding window approach to create a streaming capable function. Figure 3.2 shows the different stages of this pipeline. These stages can be grouped into two groups. The first one depends on the number of input images and the second one on the number of output images. Therefore, the units of each step of a group are multiplied by the number of its parameters. For example, the number of sliding windows and line buffers depend on the number of input images.

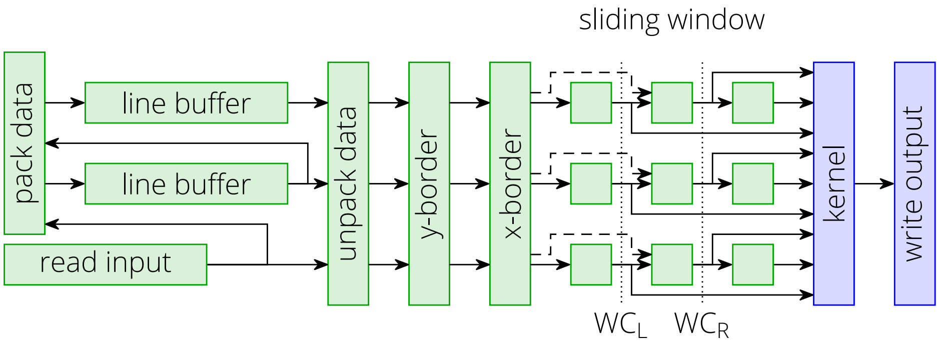


Figure 3.2: The common structure used for all filters shown for a 3 × 3 kernel. Green parts are duplicated by the number of input images and blue parts by the number of output images. Dashed lined paths are used for border handling.

An image filter needs parallel access to a window of pixels, to compute one output pixel in each clock cycle. These observed pixels are stored in a sliding window built of registers. In this window, pixels are shifted from left to right in every clock cycle. The number of rows of this sliding window is equal to the kernel size. The number of columns (WC_T) in the sliding window depends on the kernel radius (K_R) and vector size (V_S), which determines the number of pixels computed in parallel.

$$K_R = \left\lfloor \frac{K_S}{2} \right\rfloor \tag{3.2}$$

$$WC_T = K_R + V_S + V_S \cdot \left\lceil \frac{K_R}{V_S} \right\rceil \tag{3.3}$$

These window columns are divided into three parts, separated by a left border (WC_L) and a right border (WC_R). The input source of the left and middle parts, depend on the proximity of the window to the left and right image borders.

$$
\begin{aligned}
WC_L &= WC_T - V_S - K_R \\
WC_R &= WC_T - V_S
\end{aligned}
\tag{3.4}
$$

To be able to stream data and read each pixel only once from memory, complete image rows are buffered in line buffers. However, data that needs to be accessed in parallel cannot be packed in consecutive memory addresses in the same BRAM. This is because the BRAM is only dual ported, and the function needs one port to read and one port to write in each clock cycle. With an optimized usage of line buffers only $(K_S - 1)$ rows of each input image have to be buffered. To further optimize the resource usage, the complete column of all line buffers is packed into a single vector (LB_W), which depends on the vector size, number of line buffers and input image pixel bit-width (I_W).

$$LB_W = V_S \cdot (K_S - 1) \cdot I_W \tag{3.5}$$

To avoid writing each vector element into a separate BRAM, the HLS `data_pack` directive is needed. The vector is written to or read from the line buffers. Depending on the resulting bit-width of the vector, Vivado HLS uses the minimum number of BRAM. A BRAM has a maximum width of 36 bit. For large kernel sizes this partitioning can lead to a reduction in BRAM usage due to fragmentation, compared to a manual partitioning into 32 bit large data types, since 36 bit is not covered by the C++ standard.

Line buffers and sliding window need to be filled, before the first output value can be generated. The overhead to fill the line buffers is equal to the kernel radius (K_R). Whereas the overhead (O_C) to fill the sliding window is calculated based on the kernel radius and the vector size.

$$O_C = \left\lceil \frac{K_R}{V_S} \right\rceil \tag{3.6}$$

Depending on the y-coordinate and vectorized x-coordinate (x_V) of an image the key operation in each stage is:

1. If $(y < I_R)$ and $(x_V < \frac{I_C}{V_S})$ read in the next input element. This element contains a pixel vector and additional AXI4-stream signals (`last`, `user`). To avoid unnecessary data dragging, the pixel vector is extracted.
2. Read the pixel vectors from the line buffers at x_V and buffer them together with the new input pixel vector.
3. Pack and write pixel vectors to line buffers at x_V, but one row up $(y - 1)$, to be used when the y-coordinate is increased.
4. If parts of the kernel matrix go outside the border, the border handling method is invoked (e.g. replicated or constant zero).
5. The sliding window shifts pixel vectors from left to right. The source changes if window pixels are beyond image borders:
    a) If $x_V = 0$, *border* data is written into middle columns $(WC_L \leq c_W < WC_R)$.

b) If $x_v \geq \frac{I_C}{V_S}$, *border* data is written into left columns ($WCR \leq c_w$).

6. Compute the kernel function for each vector element (pixel) in parallel.
7. If ($y \geq K_R$) and ($x_v \geq O_C$) write the next element to the output. It contains the pixel vector and the generated AXI4-stream signals (`last`, `user`).

Separable Kernels

Some 2-dimensional filters have the advantage that they can be computed by using two 1-dimensional filters in sequence [248]. Such filters are called *separable*, and the library exploits this property to reduce resource consumption. This benefit increases for larger kernels. For example, the window elements for a 7 × 7 kernel would be reduced to an 1 × 7 and a 7 × 1 kernel. Equation (3.7) illustrates this optimization using a 3 × 3 Gaussian kernel.

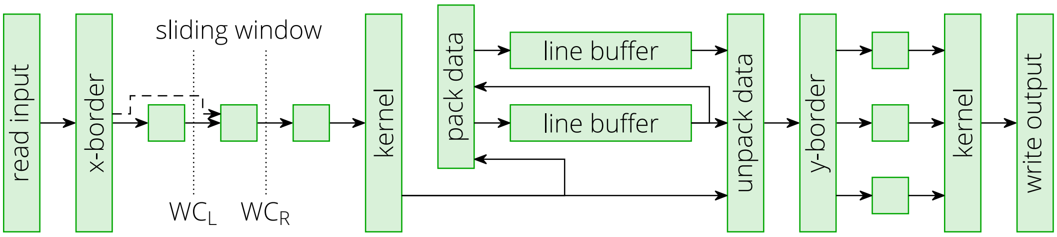


Figure 3.3: The structure used for all separable filters shown for a 3 × 3 kernel. Dashed lined paths are used for border handling.

$$\text{GaussianKernel} = \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} \cdot \frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} = \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} \tag{3.7}$$

Figure 3.3 shows the pipeline stages used for the implemented separable filters. It is like the non-separable pipeline. The separable filter pipeline first reads the input pixel and writes it into a horizontal sliding window with a size of 1 × W_C. This sliding window only needs to check the image boundaries on the x-axis. The next stage computes the horizontal compute kernel and stores its intermediate results in line buffers. Then, data is read from the line buffers and written to a vertical sliding window with a size of ($K_S \times V_S$). This sliding window needs to check the image boundaries on the y-axis. The final stage computes the vertical compute kernel and writes the results back to memory.

`HiFlipVX` provides separable filter implementations for the Gaussian, box, dilate and erode filters. There is a non-separable implementation of the box and Gaussian filter because the computed values of the output pixels differ by a maximum of one in comparison to the separable filter. The deviation results from the fact that the normalization is performed in both kernels of the separable filter, to not increase the data bit-width of the line buffers. Larger Sobel or Scharr kernels would also have advantages when using a separable filter, as their LUT and FF consumption would decrease. Since these filters compute the x and y derivatives, different units, like the line buffer, would be needed for each derivative for the vertical kernel. This would in turn increase the consumption of LUTs and FFs, and double the amount of BRAM needed.

Kernel Functions

The following describes the individual kernel functions of the filters in more detail. More about the functions defined in the OpenVX standard can be found in the manual [229]. Table 3.2 marks all functions which are not defined in the standard with a ★. All filter functions work on a window on each of their input images. Filters like box, median and Gaussian are often used to smooth the image. Other functions, such as the Sobel and Scharr filter calculate the image derivatives. Apart from the median, the above-mentioned functions convolve the image with a kernel matrix. The convolve function can also be used to compute custom or unsupported convolutions. The kernel matrix passed for convolution is also resolved at compile time. Filters like the median, dilate and erode are not linear. While the first one calculates the median of an input window, the last two calculate its maximum or minimum.

`HiFlipVX` optimizes its filter operations based on the coefficient pattern of the kernel matrix. This can be done because the coefficients are fix at synthesis time. For example, the coefficients of the Gaussian kernel are symmetrical on the x and y-axes independent from the kernel size, as shown in Equation (3.7). This symmetry gives the possibility to optimize the number of operations for different functions, such as the Gaussian, Sobel, Scharr, conductivity and DoH functions. Equation (3.8) shows the optimized computation of one pixel ($I(x,y)$) for symmetric 1D kernels, like the Gaussian kernel B, for an input window A.

$$I(x,y) = B(K_R) + \sum_{n=0}^{K_R-1} (B(n) \cdot (A(n) + A(K_S - n - 1))) \tag{3.8}$$

To avoid overflow, the various filter functions are normalized after kernel operation if needed, to prevent an overflow. The library uses shift operations to process normalization, if its value is a multiple of two, like in the (e.g., Gaussian and Scharr filter). Otherwise, the library approximates the normalization by multiplying (*mult*) and shifting (*shift*), to avoid a costly division operation (e.g., box filter). The type of normalization and its values are computed at compile time (e.g., custom convolution and Sobel filter). To compute the *mult* and *shift* values for a 16 bit accurate normalization, first all kernel coefficients are summed up (*sum*). Then, the normalization value is computed as floating-point value. This value is shifted to the maximum value, which can be represented by a 16 bit value.

$$mult = \left\lfloor \frac{1}{sum} \cdot 2^{-shift} \right\rfloor \tag{3.9}$$

The coefficients of larger Gaussian and Sobel kernel matrices are computed at synthesis time by discrete convolution of the 1D kernels using the standard smoothing kernel [1 2 1]. Equation (3.10) gives an example of the discrete convolution. Finally, the two 1D kernels are multiplied together to compute the 2D kernel matrix, as shown in Equation (3.7) for the separable filters.

$$B = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{3.10}$$

The **median** filter implementation differs from the other filters because it requires searching for the median value within the input window. A common algorithm for computing the median is to sort the pixels of the input window and select the median element of the sorted array as output. Multiple sorting networks exist, such as odd-even merge-sort, bitonic-sort, and shell-sort. These networks are a good fit for FPGAs, since they can be implemented with simple comparator networks. The library uses an odd-even merge-sort algorithm [249], since it requires fewer comparators than the others. Since the array size is not a multiple of two, this thesis uses a more generic implementation [250]. Different sorting networks have been proposed by researchers for specific array sizes. Since 3×3 kernel sizes are the most used ones, the library implements the sorting network proposed by Aranda et al. [251] for this specific case.

The **segment test detector** is part of the FAST [37] corner detector. It has been extracted from the FAST to be applied to other computer vision applications. Its window size is 7×7. In this window it extracts 16 pixels in a Bresenham circle of radius three, as shown in Figure 2.3. An image pixel value (v) is detected as a corner if ($S = 9$) continuous image pixels (v_i) in the circle are:

- lighter ($\forall i \in S, v_i > v + t_0$) or
- darker ($\forall i \in S, v_i < v + t_0$).

Their difference must be above a certain threshold (t_0). The strength or response value of a detected corner is the minimum absolute difference ($r_i = |v_i - v|$) between v and all v_i. The hardware computation consists of two parts to identify a corner and calculate its response value (r). Part one first computes the direction (lighter or darker) of v to all v_i. Part two first computes the absolute differences (r_i) of v to all v_i. Then part one checks if all possible continuous pixels ($S = 9$) in the circle ($N = 16$) are in the same direction (16 combinations). Then part one checks if ($S = 9$) contiguous pixels in the circle ($N = 16$) are either lighter or darker (for all 16 combinations). At the same time part two computes the minimum's (r_j) of all r_i for all these combinations. At the end, the maximum of all r_j is selected, where part one identifies that it is a corner. The output of the segment test detector is an image of response values. For non-corner pixels a zero is written to the output. This work has omitted the threshold (t_0) in the hardware implementation to not loose information and because it can be applied in a later function (e.g., feature extraction or threshold).

The **NMS** function searches for local maxima in a squared window. A pixel with coordinates (x, y) is kept if and only if it is greater than or equal to its top left neighbors and greater than its bottom right neighbors. Otherwise, it suppresses the observed pixel and sets it to the smallest possible value of the data type. In addition, a kernel matrix with the same size as the observed window can be provided by the user, to be used as a mask. This mask is passed with the compile-time parameters.

The **oriented NMS** function suppresses pixels that are not a maximum depending to their orientation. Therefore, it gets two images as input. One image of gradients and one image of orientations. Like the NMS function it looks in a window around a pixel to see if it is the maximum. Instead of comparing with all eight pixels around the observed one, it compares with the two pixels that are perpendicular to its orientation. For example, if the orientation shows north, the observed pixel needs to be bigger than the pixels in the east and west. To determine the exact direction of the orientation within the observed window, the function needs the quantization factor of the orientation image. The orientation image is buffered besides with the gradient image although it does not need to be windowed. This is done to prevent external buffers, because the images would otherwise be read at a different clock cycle.

The **hysteresis** filter function suppresses weak pixels that are not within the range of a strong pixel. Strong pixel values need to be above the threshold t_1 and weak pixel values above the lower threshold t_0. All pixels below t_0 are suppressed and all above t_1 are not. If a strong pixel is within the window of a weak pixel, the weak pixel will be strong. Otherwise, it will also be suppressed. The output image stores a zero for all suppressed pixels and the maximum possible value for all strong values to create a binary image.

To compute the **conductivity** function (Lc) using the Perona-Malik diffusion, two values are needed. The first value is the contrast factor (γ), and the second value is the absolute image gradient. To better detect blobs, a conductivity coefficient introduced by Perona-Malik is used that favors wide regions over small ones. It is shown in Equation (3.11) and has been reformulated to reduce computational effort. Other formulas exist to calculate the conductivity function, which, however, showed worse results in the evaluation of this work. Furthermore, they are more computationally and resource intensive, since they need to compute an exponent in addition to the division.

$$Lc = \frac{1}{1 + \left(\frac{|\Delta L|}{\gamma}\right)^2} = \frac{\gamma^2}{\gamma^2 + Lx^2 + Ly^2} \tag{3.11}$$

The library uses the Scharr filter to calculate the first order derivatives. Then the absolute gradient is calculated. If its value is zero, it is rounded up to the smallest representable fixed-point value. This ensures that there is neither a division by zero nor an overflow in the final conductivity coefficient. Three different accuracy levels have been implemented, since the computation of the gradient and division with maximum accuracy leads to a high resource consumption. The contrast factor (γ) is a scalar value. The function receives it already squared to reduce the number of operations. This is since the same contrast factor can be used in different scale levels of a feature detection algorithm. The bit precision of the squared contrast factor is 32 bit, since γ^2 is in the dividend and requires a high precision.

The **FED** function (Lf) describes the diffusion process of an image and needs three parameters as input. These are the input image (Lt), the conductivity image (Lc), and the step size (τ), which is passed as template parameter. The library has an optional output image for the FED function. It is the forwarded Lc image, which can be used by the subsequent FED function as input. The advantage of forwarding the Lc image is that additional buffers are saved, since the diffusion process consists of several consecutive FED functions, and they all need the same conductivity image. To calculate a pixel of the output image, its four adjacent pixels in the input image and the conductivity image are needed. For the diffusion process, the divergence must be calculated first:

$$\begin{aligned} d_{x+} &= (Lt(x+1,y) - Lt(x,y)) \cdot (Lc(x+1,y) + Lc(x,y)) \\ d_{x-} &= (Lt(x,y) - Lt(x-1,y)) \cdot (Lc(x,y) + Lc(x-1,y)) \\ d_{y+} &= (Lt(x,y+1) - Lt(x,y)) \cdot (Lc(x,y+1) + Lc(x,y)) \\ d_{y-} &= (Lt(x,y) - Lt(x,y-1)) \cdot (Lc(x,y) + Lc(x,y-1)) \end{aligned} \tag{3.12}$$

If $(Lt(x+1,y) - Lt(x,y))$ is positive, the current pixel has a lower concentration than its right counterpart and the flow goes in direction of the current pixel. If $(Lt(x,y) - Lt(x-1,y))$ is positive, the current pixel has higher concentration than its left counterpart and the flow goes in direction of the neighboring pixel. If the difference is zero, no flow occurs from or to the corresponding direction. The same applies for the neighboring pixels in y direction. The sum of the conduction coefficients defines how strong the flow in each direction is, as shown in the following equation:

$$Lf(x,y) = Lt(x,y) + \frac{\tau_{i,j}}{2} \cdot (d_{x+} - d_{x-} + d_{y+} - d_{y-}) \tag{3.13}$$

The division by two can be performed as a shift operation or before reading τ, since τ is passed as a constant. The value for τ has a 16 bit fraction part, like α of the pixelwise functions or the contrast factor γ. Tests have shown that an 8 bit fraction does not provide a sufficiently high accuracy. Since τ can take values greater than one, the maximum integer fraction is set to 16 bit. Since the value of τ is a compile-time constant, unnecessary bits are automatically removed by the compiler. During the diffusion process, the stability condition can be violated between consecutive FED functions, as far as it is stable after the last one. An evaluation has shown that the values vary between –0.1 and 1.1. Saturating these values did not change the result noticeably. Therefore, the library limits the range to values that are above or equal zero and less than one. This increases the precision by 2 bit, because of using fixed-point values and omitting the signed bit and the integer bit.

The **DoH** function (Ld) is a blob detector, which is used in many feature detection algorithms. As shown in Equation (3.14), the second order derivatives are needed for its calculation. To compute the derivatives, the library uses the Scharr filter. This thesis provides two approaches to calculate the derivatives. In the first approach, the DoH filter gets the first order derivatives as input (Lx, Ly) and uses them to compute the second order derivatives (Lxx, Lxy, Lyy). In the second approach, the first and second order derivatives are performed in one step. This is achieved by convolving the Scharr kernel matrix with itself beforehand at compile time. Equation (3.15) shows the resulting kernel matrices (Kxx, Kxy, Kyy).

$$Ld(x,y) = Lxx(x,y) \cdot Lyy(x,y) - Lxy^2(x,y) \tag{3.14}$$

$$\begin{gathered}
\mathrm{Kxx} = \begin{bmatrix} 9 & 0 & -18 & 0 & 9 \\ 60 & 0 & -120 & 0 & 60 \\ 118 & 0 & -236 & 0 & 118 \\ 60 & 0 & -120 & 0 & 60 \\ 9 & 0 & -18 & 0 & 9 \end{bmatrix}, \quad \mathrm{Kxy} = \begin{bmatrix} 9 & 30 & 0 & -30 & -9 \\ 30 & 100 & 0 & -100 & -30 \\ 0 & 0 & 0 & 0 & 0 \\ -30 & -100 & 0 & 100 & 30 \\ -9 & -30 & 0 & 30 & 9 \end{bmatrix} \\
\mathrm{Kyy} = \begin{bmatrix} 9 & 60 & 118 & 60 & 9 \\ 0 & 0 & 0 & 0 & 0 \\ -18 & -120 & -236 & -120 & -18 \\ 0 & 0 & 0 & 0 & 0 \\ 9 & 60 & 118 & 60 & 9 \end{bmatrix}
\end{gathered} \tag{3.15}$$

Both approaches have benefits and drawbacks. Parallel execution on FPGAs allows the second approach to be chosen without sacrificing performance compared to other architectures. Moreover, the deviations resulting from the normalization of the first-order derivative in the first approach are minimized. In contrary, the first approach allows the reuse of first order derivatives and finer granularity to reduce fragmentation in cluster-based systems. Unlike the other filter functions, the DoH calculates three kernels on one or two inputs, but with only one output as a result. The function DoH takes signed 8 bit and 16 bit data types as input and supports output bit-widths of 16 bit and 32 bit. To achieve the highest possible accuracy, the internal Scharr filters are normalized after the full DoH has been calculated. However, in the case of an 8 bit input and a 32 bit output, this means that the value is shifted up and not down. An evaluation of the range of values showed that the values of the output image are between –0.25 and 0.125. This means that the value range, including the negative numbers, can be displayed without overflow.

3.1.3 Image Conversion Functions

Table 3.4 shows the different image conversion functions of the library. Most of these functions modify image attributes, such as the format, bit-width, vector size, or resolution. Other functions are more specific for the streaming approach on FPGAs, such as the scatter, gather and multicast functions.

Table 3.4: `HiFlipVX` image conversion functions. Non-standard are marked with ★.

Channel Combine	Scale Image	Gather ★
Channel Extract	Bit-Width Conversion	Scatter ★
Color Conversion	Data-Width Conversion ★	Multicast ★

The **channel combine** function takes multiple unsigned 8 bit planes and combines them to a multiplanar or interleaved image format. Whereas the **channel extract** function extracts a single plane (channel) from a multiplanar or interleaved image format. The main implemented image formats are `RGB`, `RGBX` and grayscale (8 bit unsigned). For grayscale conversion, the library approximates to the BT.601 recommendation using multiplications and a shift operation.

$$\text{gray} = (R \cdot 306 + G \cdot 601 + B \cdot 117 + 512) \cdot 2^{-10} \tag{3.16}$$

The **color conversion** function can convert between these image formats. Additionally, the channel combine and channel extract functions support interleaving two or four 8 bit pixels in an unspecified 16 bit or 32 bit image format. Since C++ does not define 24 bit variables, the library uses 32 bit variables to store `RGB` values in memory (e.g., `[RGBR][GBRG][BRGB]`). It is different for the `RGBX` format where the last 8 bit always remain free (e.g., `[RGB0][RGB0]`).

The **bit-width conversion** function can convert between any signed/unsigned 8 bit, 16 bit or 32 bit format and also supports vectorization. The **data-width conversion** converts between two buffers with a different vector size by increasing or decreasing it. If the input vector is a multiple of the output vector, the input vector is locally buffered and divided into subvectors. These subvectors are written one by one to the output. If the output vector is a multiple of the input vector, it is the other way around. Therefore, the functions latency depends on the image parameters with the lower vector size. However, the vectors do not need to be a multiple of each other. In this case it can happen that the number of input and output pixels are not equal, since the number of image pixels must be multiple of its vector size.

Therefore, the data of the bigger image needs to be aligned. In this case, the size of the local buffer is the LCM (Least Common Multiple) of both vector sizes.

The implemented **scale image** function reduces the resolution of an image. It supports nearest neighbor, bilinear and area interpolation for unsigned 8 bit data type images. Since scaling factors are known at compile-time, the library can calculate all needed coordinates using multiplications and shift operations and can thus avoid divisions at runtime. Even if not all input pixels are needed for nearest neighbor interpolation, they still must be read in because of the streaming approach. The bilinear interpolation needs to buffer pixels of two consecutive rows in BRAM for streaming capability. While the number of buffers for the area interpolation depends on the scale factor.

The scale image function also supports a fast area interpolation for images that are scaled down by a scale factor (σ) of two, four or eight. All pixels in the observed window ($\sigma \times \sigma$) are summed and the mean is calculated by using shift operations. Unlike the other scale image functions, it allows vectorization (V_S) and more data types. The vectorization of the output image decreases according to the scale factor ($\left\lceil \frac{V_S}{\sigma} \right\rceil$). The function uses the sliding window approach of the filter functions. With the difference that the kernel size is an even number ($K_S = \sigma$), and output pixels are not written in each clock cycle. A pixel is written to the output after every (σ) rows and ($\left\lceil \frac{\sigma}{V_S} \right\rceil$) columns. The sliding window does not move outside the image boundaries if the image width and height are multiples of the vector size. To support down scaling where the resolution is not a multiple of the scale, replicated borders are used for the right and bottom image border.

The **multicast** function gets an image and writes it to multiple outputs, depending on how many are needed. A reason this function is needed is that buffers (or FIFO units) between streaming functions can only have one consumer and one producer. The **scatter** function splits one input image into multiple output images and the **gather** function gathers multiple input images into one output image. Both functions are needed to divide work to multiple CUs and combine their results again. Both functions currently support two modes. In `block` mode, the partial images are read in or written out as whole one after the other. In `cyclic` mode the images are interleaved, by changing the port after reading or writing of each pixel. The second mode requires that all partial images have the same number of pixels. All three functions (scatter, gather and multicast) can be vectorized and allow a selectable number of input or output images.

3.1.4 Image Analysis Functions

In Table 3.5 the implemented analysis functions are shown. A common feature of these functions is that an analysis of the whole input image is needed to calculate single pixels of the output and not only on a window of the input image like in the filter functions. Compared to the other function classes, some of these functions require multiple loops to produce the final output. This makes them a challenge for the streaming capability.

In the **integral image** function, an output pixel is the sum of the corresponding input pixel $src(x, y)$ and all other pixels whose x and y-coordinates are lower or equal. Equation (3.17) shows the hardware optimization for the calculation of the integral image. The integral result ($area$) is the sum of the current row (sum) added to the integral value at position $dst(x, y-1)$.

Table 3.5: `HiFlipVX` image analysis functions. Non-standard are marked with ★.

Histogram	Table Lookup	Mean & Standard Deviation
Equalized Histogram	Integral Image	Min & Max Location
Contrast Factor ★	Scalar Operation	

Therefore, the function buffers the integral results of the previous row ($buf(x,y)$) in BRAM. The bit-width is 8 bit for the input image and 32 bit for the output image.

$$\begin{aligned} sum &= \begin{cases} sum + src(x,y) & x > 0 \\ 0 & x \leq 0 \end{cases} \\ area &= \begin{cases} sum + buf(x) & y > 0 \\ sum & y \leq 0 \end{cases} \\ dst(x,y) &= area \\ buf(x) &= area \end{aligned} \tag{3.17}$$

The **min-max location** function finds the minimum and maximum pixel values in an image. If specified it can also output the number of pixels that have the same value as the maximum or minimum value including their coordinates. The coordinates of the minima and maxima need to be buffered locally in separate buffers. To restrict memory usage, the library adds a template parameter for the buffer capacities (N). The maximum and minimum values and their counters can only be output after the complete input image has been processed. To output its coordinates, it needs a second loop, which has a maximum of N loop iterations. The capacity is important for the worst-case execution estimations.

The **mean and standard deviation** function computes the mean (μ) and standard deviation (σ) of an image as shown in Equation (3.18). In a first loop the sum for the mean (μ) is computed. The standard deviation computation is optional and is computed in a second loop. To prevent from using a resource-consuming division operation, the sum is multiplied by the reciprocal of the pixel amount, which is a compile-time constant. This multiplication operation and the square root are done outside of the loops, to not be pipelined, which would consume unnecessary extra resources. A challenge for the streaming capability is that the image must be read in twice. To simplify this and avoid unnecessary buffering of the whole image, there is a separate port for each loop reading the input image.

$$\begin{aligned} \mu &= \frac{\sum_{y=1}^{I_R} \sum_{x=1}^{I_C} src(x,y)}{I_C \cdot I_R} \\ \sigma &= \sqrt{\frac{\sum_{y=1}^{I_R} \sum_{x=1}^{I_C} (\mu - src(x,y))}{I_C \cdot I_R}} \end{aligned} \tag{3.18}$$

The **table lookup** function takes the image input pixels to index into a LUT and stores the indexed value in the output image. It supports 8 bit unsigned and 16 bit signed data types, which are equal for input, LUT and output, and sets the LUT size and offset as template parameters. There are two loops in the implementation. The first loop reads the content of the LUT. The second loop calculates the output pixel as shown in Equation (3.19). If the index is out of range, the library outputs a zero as value, since it expects a complete image in a stream.

$$dst(x,y) = lut(src(x,y) + offset) \tag{3.19}$$

The **histogram** function counts the number of occurrences of each pixel value within a certain range dependent of the number of bins. A pixel with its intensity value v will result in incrementing histogram bin i as shown in Equation (3.20). The function supports 8 bit and 16 bit unsigned data types, and sets the *RANGE*, *OFFSET* and *BINS* values as template parameters. It is separated in three stages (loops). The first stage resets the histogram entries to zero. The second one reads one input pixel in each clock cycle and increments the corresponding histogram entry. It increments two independent histogram buffers alternately, since incrementing a BRAM entry cannot be done in one clock cycle. The third stage sums the histogram bin of both buffers at position i and writes them pixel by pixel to the output.

$$i = (v - OFFSET) \cdot \frac{BINS}{RANGE}, \quad OFFSET \leq v < OFFSET + RANGE \tag{3.20}$$

For the vectorization of the histogram, V_S times more buffers are needed to enable enough parallel memory access or bandwidth. This reduces the latency of the second loop by a factor of V_S. Because each histogram buffer has the same size (*BINS*), vectorization does not change the latency of the first loop. The latency gain from vectorizing the third loop, and thus the output histogram, is disproportionate to the additional resources required. This is because the first and third loops depend on *BINS* and the second on the resolution ($I_R \cdot I_C$), which is much higher. The larger the vectorization, the more histogram buffers must be summed up within one clock cycle of the third loop. This leads to either a reduced frequency or a pipeline interval (P_I) which is larger than one. The achieved value for P_I is about $\log_2(V_S)$, due to the adder tree for summing up the histogram buffer entries.

The **equalized histogram** [252] function modifies an input image so that the intensity histogram of the resulting image becomes uniform, resulting in an enhanced contrast. The hardware function is computed in four loops. The first two loops reset the histogram ($hist(i)$) and create a new one from the input image using the same approach as in the histogram function. Equation (3.21) shows how to compute the equalized histogram (eq). The value cdf_{min} is the minimum non-zero value of the cumulative distribution function ($cdf(i)$), which is computed together with the histogram in the second loop. The third loop computes the $cdf(i)$ values on the fly, since it is the summation of the histogram entries (prefix sum), to calculate the equalized histogram ($eq(i)$) from it. The computation of α is done between the second and third loop, since it does not depend on i and to keep resource consumption low. It needs the number of image pixels and the maximum possible image value (I_{MAX}). The computation is done using fixed-point numbers with a 24 bit fraction. The multiplication is done using shift operations.

$$eq(i) = \left\lfloor ((cdf(i) - cdf_{min}) \cdot \underbrace{\frac{I_{MAX} \cdot 2^{24}}{I_R \cdot I_C - cdf_{min}}}_{\alpha}) \cdot 2^{-24} \right\rfloor, \quad cdf(i) = \sum_{i}^{BINS} hist(i) \tag{3.21}$$

The **contrast factor** specifies the nth percentile of the smoothed gradient histogram of an input image. It is used for the conductivity function to indicate how many details of an image should be suppressed. The implemented function gets the gradient magnitude of an image as input and computes a single value as output. First of all, the histogram of the input image is calculated (*hist*). Its implementation is similar to that of the normal histogram function. There is a template parameter that specifies how many border pixels should be skipped. The reason lies in the preceding filter functions, whose border handling can lead to non-accurate input pixels on the image borders. However, the entire input image must still be read, due to the streaming approach. Only pixels with a value greater than zero are considered for the histogram. The number of considered pixels of each vector element must be counted for later computation. In the next step these counters are added together and multiplied by the given percentile to obtain the required threshold (t_1).

$$t_1 = percentile \cdot \sum_{n=1}^{V_C} counter \tag{3.22}$$

The value of the percentile is passed as a template parameter to the contrast factor function. Its value consists of a 16 bit fraction part. A value greater than or equal to one would not make sense, since it would always lead to the same result ($y = 1$). Then, it is iterated over all ($V_C \cdot 2$) histograms in parallel and their entries are summed (sum_{bin}) until the threshold (t_1) is reached. The last loop iteration (bin) before sum_{bin} reached t_1 is used for further calculation. To compute the value of the contrast factor (y), the library makes some simplifications, which results from using fixed-point numbers and a fixed histogram size.

$$y = \frac{hist_{max} \cdot bin}{bin_{max}} \approx \begin{cases} 1966 & sum_{bin} < t_1 \\ bin \cdot 2^7 & \text{8 bit input} \\ bin \cdot 2^6 & \text{16 bit input} \end{cases} \tag{3.23}$$

The division is replaced by a shift operation by using a fixed histogram size and thus fixed values for $hist_{max}$ and bin_{max}. For input images with an 8 bit data-width, it requires a histogram with a maximum of 256 entries (8 bit). For an input image with a 16 bit data-width, 512 entries were sufficient to achieve a reasonable accuracy (9 bit). Using different input images and random numbers, it was empirically found that the value for h_{max} is always less than 0.5. Therefore, the gradient values were normalized to 0.5. Additionally, the final value needs to be shifted by the bit-width of the input pixel (8 bit and 16 bit) and by the bit-width of the output variable (16 bit). For reasons of plausibility, a minimum value of 0.03 (or 1966 as fixed-point) is applied for the contrast factor [20].

The **scalar operation** function allows for conditional flow within the OpenVX graph. As shown in the standard it allows different logical, comparison or arithmetic operations with two inputs and one output. The function allows different integer or floating-point data types. For a modulus operation by zero the result is promoted to a zero. For a division operation by zero the result is promoted to the maximum value of the input data type. If the output type is an integer and the input a floating-point, the result will be saturated and rounded to zero. For all other conversions, the result is truncated.

3.1.5 Image Function Latency & HLS Directive Usage

The latency of the different image processing functions can be calculated in the same way. All library functions use the `pipeline` directive to optimize their throughput. Here the pipeline interval is described by P_I and the pipeline depth by P_D. In case of the implemented library functions the goal for P_I is always one. The pipeline depth depends on the maximum number of sequential operations and the achieved frequency. Since all loops below the `pipeline` directive are unrolled automatically, there is no need of using the `unroll` directive within these loops. Outside of these loops the library needs this directive, for example, for the compile time computation of kernel matrices of some filter function. Additionally, all internal and callable library functions are inlined using the `inline` directive, to let the compiler optimize latency and resource usage.

The goal of all implemented functions is to process V_S number of pixel per clock cycle. However, there is a small overhead for filling line buffers (K_R) and sliding window (O_C), if any. Apart from minor variations in the number of pipeline stages, the latency does not change between the separable and non-separable filters. The latency of a function depends on the input or output image with the highest resolution ($I_R \times I_C$). Some of the analysis functions consist of multiple loops. For each of these loops, the latency is calculated as follows:

$$\text{Latency}_{\text{loop}} = (I_R + K_R) \cdot \left(\frac{I_C}{V_S} + O_C \right) \cdot P_I + P_D \tag{3.24}$$

For bit-widths above 64 bit and to perform vectorization, the library uses the template based data type shown in Listing 3.2. To allow a vectorized data type to be expanded to a signal with full bit-width, the `data_pack` directive is needed. Additionally, the library uses the `data_pack` directive for internal buffers and FIFO units, to reduce the fragmentation of the utilized BRAMs. In the latest toolchain of XILINX, named Vitis, the `data_pack` directive has been replaced by the `aggregate` directive. With the help of the global `__VITIS_HLS__` macro, set by the XILINX tool, the library automatically selects the correct directive.

```cpp
template<typename T, vx_uint8 vector_size>
struct vx_data_pack {
  T data[vector_size];
};
```

Listing 3.2: Generic data type used for vectorization and to data types wider than 64 bit.

Since all functions are streaming capable, it is easy to connect different functions with each other, as shown in Listing 3.3. For example, by using the `dataflow` and `stream` directives, multiple functions can be connected within one IP-core to achieve function-level or loop-level parallelism. All arrays used between these functions are converted to FIFO units at synthesis time, to stream data. A small depth allows to use LUTs instead of BRAMs for these FIFO units, since BRAM is often a limiting resource. The library uses this approach to provide more complex functions like the ORB algorithm as a single library function. The `DECISION` framework needs this to create an optimized CU.

```
void Example(vx_image_data<vx_uint8, 2> in[PIXELS / 2],
  vx_image_data<vx_int8, 2> out[PIXELS / 2]) {
#pragma hls interface ap_ctrl_none port=return
#pragma hls interface axis port=in
#pragma hls interface axis port=out

  static vx_data_pack<vx_int8, 2> lx[PIXELS / 2];
#pragma hls stream variable = lx depth = 8
#pragma hls data_pack variable = lx
#pragma hls resource variable = lx core = FIFO_LUTRAM
  static vx_data_pack<vx_int8, 2> ly[PIXELS / 2];
#pragma hls stream variable = ly depth = 8
#pragma hls data_pack variable= ly
#pragma hls resource variable = ly core = FIFO_LUTRAM

#pragma hls dataflow
  Scharr3x3<...>(in, lx, ly);
  Magnitude<...>(lx, ly, out);
}
```

Listing 3.3: Example application showing how to manually connect two `HiFlipVX` functions with a vector size of two. Functions operate in parallel by streaming data.

As shown in Equation (3.25), the latency, for a `data_flow` region consisting of multiple functions, is different compared to a single function. The equation consists of the trip count (T_C) that includes the overhead of the line buffers and the sliding windows, a very deep pipeline, and the pipeline interval. The largest overhead comes from the row buffers, because they must be filled before the first output pixel of the last function can be computed ($\sum_{i=1}^{N} K_{R_i}$). Since the sliding window is refilled for each row, the function with the largest overhead dominates the other functions and limits throughput ($\max_{1\leq i\leq N} O_{C_i}$). Since the output within the pipeline of a single function is usually written last, the pipeline steps of all functions can be summed without adjustment ($\sum_{i=1}^{N} P_{D_i}$). In addition, the pipeline steps of the FIFO units that lie between the functions must be added ($\sum_{i=1}^{N-1} P_{F_i}$). Also, the sliding windows of all functions must be filled partly before the last function receives its first input pixel ($\sum_{i=1}^{N-1} O_{C_i}$). Pipelining makes it possible to calculate a pixel vector in each clock cycle, so that the overhead of the sliding windows can be added to the overall pipeline. A different vectorization for the individual functions would not make sense, due to the resulting imbalance in latency of the individual functions.

$$\text{Latency}_{\text{data\_flow}} \approx \underbrace{(I_R + \sum_{i=1}^{N} K_{R_i}) \cdot (\frac{I_C}{V_S} + \max_{1\leq i\leq N} O_{C_i})}_{\text{trip count } (T_C)} \cdot P_I + \underbrace{\sum_{i=1}^{N} P_{D_i} + \sum_{i=1}^{N-1} O_{C_i} + \sum_{i=1}^{N-1} P_{F_i}}_{\text{pipeline depth}} \tag{3.25}$$

For interface parameters, the library implements the `vx_image_data` data type. Compared to the `vx_data_pack`, it contains a `last` and a `user` signal. Both signals are one bit wide and can be turned on when needed. The `last` signal is needed for XILINX DMA IP-cores, to identify the end of an image or buffer. Additionally, the `user` signal is needed for the XILINX video DMA to identify the start of an image.

An `interface` directive is only needed in wrapper functions, which instantiate the library functions and set the template parameters. There is an example implementation for each library function inside `HiFlipVX`. For the SDSoC tool it sets the `ap_fifo` protocol for all ports. For XILINX Vivado HLS it sets the AXI4-Stream `(axis)` protocol as interface for the ports. Additionally, it deactivates the control port of all IP-Cores in Vivado HLS `(ap_ctrl_none port=return)`. This port should not be deactivated for SDSoC. The `(__SDSCC__)` macro is globally set by the SDSoC tool and is used by `HiFlipVX` to automatically switch between the two XILINX tools. Setting the `ap_fifo` ports and using the C99 style for arrays, the library does not need any additional SDSoC directives `(pragma SDS)`. When using an AXI4-stream interface, the different IP-cores can easily be connected in the Vivado tool. The `DECISION` framework uses this approach to achieve less fragmentation and to parallelize and thus speedup the synthesis process.

In some cases the library uses a `resource` directive to specify, if LUTs or BRAMs should be used for internal memories or FIFO units, as shown in Listing 3.3. The use of these directives should be used with caution, since it can also have a negative effect. In most cases, it is advisable to give the tool the choice, because then it can select according to the total resource usage, bit-widths and selected frequency. Internally, the `array_partition` directive is needed if the LUT and BRAM memories do not provide the required bandwidth. The library uses this, for example, for the LUTs of the vectorized histogram functions. This directive is also used to completely partition C++ arrays into registers, like for the sliding window.

When using more complex tools like SDSoC instead of Vivado HLS there are some smaller restrictions that basically affect the interfaces of the function. One of these limitations is that only `structs` with a vector size bigger than one are synthesizable. This has been solved by automatically using native data types instead of `structs` for these kind of interfaces, when using this tool. This is also possible, since SDSoC adds the `last` and `user` signals to the AXI4-stream interface by itself. Furthermore, interface arrays need a known number of elements with this tool.

3.1.6 Feature Functions

The left two columns of Table 3.6 contain all functions that create or work on a set of features. The right two columns list all feature detection algorithms that have been created using `HiFlipVX`. These algorithms consist exclusively of other functions from the library. In the next section of this chapter, these algorithms will be discussed in more detail. The individual feature functions emerged from the exploration of these algorithms. However, generic and parameterizable functions have been developed that can be used in many other applications.

A feature has certain properties. To increase the maximum performance on FPGAs, the individual elements of a feature can be reduced in bit-width by using integer and fixed-point variables. The data type of this library is therefore only 64 bit wide. It contains the x and y coordinates, response value (r), orientation (α) and a class ID. This ID describes either the scale (σ) of a feature or its evolution level. Figure 3.4 shows the data type and the length

Table 3.6: `HiFlipVX` feature functions. Non-standard are marked with ★. The right two columns contain complex algorithms consisting of multiple library function.

Feature Extract ★	Feature Deserialize ★	FAST Corners	AKAZE Contrast ★
Feature Compare ★	Feature Retain Best ★	Canny Edge	AKAZE Features ★
Feature Gather ★	Feature Multicast ★	ORB Features ★	

of the individual elements. For the 16 bit large coordinates, the library uses a fixed-point data type whose configuration depends on the image resolution. For example, for an image resolution of 1920 × 1080 pixels, it consists of an 11 bit integer part and a 5 bit fraction part. The fraction part can be calculated, for example, using a SR, which is part of the feature extract function. The response value represents the strength of a feature and is calculated by the feature detector. Its a 16 bit integer values, whose interpretation depends on the algorithm itself. For example:

- blobs by the DoH function as part of the AKAZE algorithm
- edges by the Sobel function as part of the FAST corner detector or ORB algorithm
- corners by the segment test detector function as part of the Canny edge detector

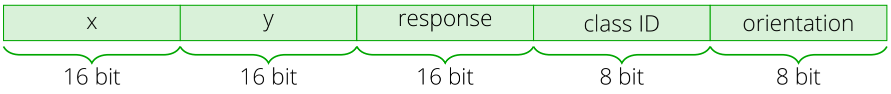


Figure 3.4: Fields of a feature consisting of fixed-point values with variable size.

The orientation of a feature is needed for its rotation invariance. It is usually calculated in the descriptor, as in the FREAK algorithm, but can also be calculated beforehand, as in the phase function. Tests have shown that quantization with up to 256 different values is sufficient to obtain a reasonably accurate angle for the orientation. Multiscale feature extraction algorithms introduce a scale to achieve scale invariance. The scale often results from the scale space of the algorithm. The library uses a fixed-point number, whose integer part depends on the maximum scale size. Since 8 bit may not be sufficient, the level of the scale space can be specified instead. The real scale of a feature can then be derived from this value.

A challenge in implementing the individual feature functions is that the exact size of an array of features cannot be determined at compile time. However, the calculation of the latency and the resulting WCET (Worst-Case Execution Time) are crucial for many systems. Therefore, in all library functions, the user specifies the maximum size for the input and output feature arrays. However, a feature array can also consist of fewer features. In this case an additional feature is appended to mark the end of the array. This allows the receiver to finalize calculations and not wait forever. To distinguish it from other features, its elements are set to the maximum possible value. In addition, it can happen that a single feature is marked as invalid by a function and must be recognized by other functions. For example, this is necessary with the feature extract function, since it can happen with a vectorization that individual elements of the output vector are marked as invalid.

Feature Extract

The feature extract function is needed to create the initial feature array. Therefore, it is essential for most feature extraction algorithms, regardless of the type of features (blob, edge, corner). For this reason, the library includes several functionalities in this function. As input the function gets an image consisting of response values. The response value (r) indicates the strength of a feature. Optionally, the function can accept an image with orientations (or angles) (α), which are then stored in the feature. Otherwise, the value is set to zero and calculated at a later time, for example in the descriptor. The output of the function is an array of features consisting of the already mentioned elements. The user determines the maximum size of the array.

Whether an input pixel is selected as a feature is determined by a fixed threshold (t_0) defined by the user. Its response value must then be above this threshold. To determine the exact x and y coordinates of a feature from its input image, a sampling factor is needed. This parameter tells with which factor the input image was scaled compared to the original image. The parameter is necessary because in a multiscale algorithm there are usually several feature extract functions, which work on differently scaled image sizes. Furthermore, the actual scale value (σ) is needed. This can be different from the sampling factor in some algorithms, since the image does not need to be resized when the scale is increased. As already mentioned, the evolution level (or class ID) can also be passed here, if the bit precision of the element is not sufficient to represent the scale. The scale can then be derived again from this ID.

Another parameter determines the minimum distance a feature must have to the image border. This is important because a feature has a certain scale, and it can lead to inaccuracies through border handling of the preceding functions. Furthermore, the radius that most descriptors need to describe a feature is much larger than its scale. Thus, at this stage of an algorithm, some feature can already be sorted out to save resources on the one hand and to minimize latency on the other hand. In addition, it can otherwise happen that features close to the border displace other features due to capacities and the threshold, although these will be sorted out at a later point in the algorithm.

Optionally, the feature extract function can also do an NMS and a SR. One reason for the integration of both functionalities is that they work on a window of the same input image. In addition, there are certain dependencies that are difficult to separate without using too many extra resources. For example, the NMS sorts out pixels that should not be used by the SR. However, the NMS output image cannot be used for the SR as input because the SR needs the original unsuppressed response values. At the same time, the SR creates coordinates as an output, which would have to be buffered in addition to the actual response image if the NMS would be executed after the SR.

For NMS, a pixel is kept if and only if it is greater than or equal to its top left neighbors and greater than its bottom right neighbors. In addition, there are three options how to create this window. The first one works on a quadratic kernel window. In the second one, the user can specify a mask, which indicates whether the NMS should consider an element of the window. In the third version, a circle (σ) is specified in addition to the kernel window. This has the advantage that one could use, e.g., the scale (σ) of the feature, to find the strongest feature in its range. The corresponding mask for the circle is calculated at compile time. Whether an element of the mask window is inside the circle depends on its distance from the center of the mask in x and y direction and the given radius.

$$mask_{x,y} = \begin{cases} 0 & x^2 + y^2 \leq \lfloor \sigma^2 + 0.5 \rfloor \\ 1 & \text{otherwise} \end{cases} \tag{3.26}$$

To increase the accuracy of the position of a feature, a SR can be calculated. To find this position, the Gaussian elimination method is applied to the DoH matrix (H) as an interpolation method. For this purpose, the gradient values (Lx and Ly) as well as the elements of the Hessian matrix (Lxx, Lyy and Lxy) are needed first. These values can be calculated using the finite difference scheme for gradient and Hessian matrix estimation [253], which is used in both AKAZE and SURF. To apply this scheme, it needs the surrounding elements from the matrix containing the calculated DoH.

$$\begin{aligned} Lx &= \frac{H(x+1,y) - H(x-1,y)}{2} \\ Ly &= \frac{H(x,y+1) - H(x,y-1)}{2} \\ Lxx &= H(x+1,y) + H(x-1,y) - 2 \cdot H(x,y) \\ Lyy &= H(x,y+1) + H(x,y-1) - 2 \cdot H(x,y) \\ Lxy &= \frac{H(x-1,y-1) + H(x+1,y+1) - H(x+1,y-1) - H(x-1,y+1)}{4} \end{aligned} \tag{3.27}$$

Having all five values the Gaussian elimination can be applied. The result of the Gaussian elimination method r_x and r_y is the difference between the pixel location and the subpixel location and must be therefore added to the pixel location of the center element.

$$\begin{aligned} r_y &= \frac{r_{y,num}}{r_{y,den}} = \frac{Lx \cdot Lxy - Ly \cdot Lxx}{Lyy \cdot Lxx - Lxy^2} \\ r_x &= \frac{-Lx - Lxy \cdot r_y}{Lxx} = \frac{Lx \cdot r_{y,den} + Lxy \cdot r_{y,num}}{Lxx \cdot r_{y,den}} \end{aligned} \tag{3.28}$$

For a valid feature, the number of r_x and r_y must not be greater than 1, or they will no longer be in the subpixel range. One problem when using the Gaussian elimination is that it requires two divisions. Since the number of fraction bits of the feature is generally quite small, it does not need the full precision of a division. The library splits r_x and r_y into their numerator ($r_{x,num}$ and $r_{y,num}$) and denominator ($r_{x,den}$ and $r_{y,den}$) parts. Looking at the equation, r_x is no longer directly dependent on r_y, which would otherwise lead to a reduction of the accuracy, due to the division. It is easy to see that the comparison of $r_{x/y,num} \leq r_{x/y,den}$ is equivalent to $r_{x/y} \leq 1$. The simplified division function requires bit shift, subtraction, a comparison and an *OR* operation to approximate the result. For precision, the library only needs one bit to cover the integer range from -1 to 1, and $16 - \lceil log_2(I_C) \rceil$ bits for the fraction part of the final x and y coordinates. In this case the columns (I_C) of the original image must be used.

Feature Deserialize

The feature deserialize function turns an input stream of feature vectors into an output stream of single features. One reason for this function is that one of the following functions cannot operate on vectors. Furthermore, the resource consumption of the subsequent functions can be reduced compared to a vectorized function. Moreover, vectorization might not lead to a noticeable improvement in the latency of the entire algorithm. For example, depending on the maximum number of features to be processed, the feature compare and feature retain best functions may require significantly fewer clock cycles than the feature extract function, which depends on the image resolution.

The number of maximum features may differ between the input array and the output array. One reason for this is that individual features of the input array vectors may be invalid. Also, the array sizes always represent their maximum sizes, which differ from the actual sizes. The invalid features are sorted out using a comparison function. The feature deserialize function has been written generically so that different comparison functions can be provided. For this purpose, the data type, a comparison function and the invalid element need to be specified. In the wrapper function, these properties are passed.

Feature Compare

The feature compare function compares one feature array with itself and zero to two other feature arrays. If the response value (r) of a feature of the main array is greater than the response value of all input features within its scale (σ), then it is passed to the output. Otherwise, the feature is discarded. The maximum size of the different input arrays may differ. However, the maximum size of the output array must be equal to or smaller than that of the main input array. To be able to compare features with each other, a certain number of features must be stored in internal buffers. The library uses a ring buffer whose maximum size depends on certain parameters:

- number of input arrays (N)
- scale of the main input feature array (σ)
- number of columns in the sampled input image of the preceding feature extract function (I_C)
- radius of the NMS of the preceding feature extract function (K_R)

Its calculation is shown in Equation (3.29) The buffer size cannot be larger than the input feature array size. However, this is only the maximum possible size. In almost all cases, the required buffer size is much lower, which is why the library allows to limit the buffer size.

$$\text{WorstCaseBufferSize} = \frac{I_C}{K_R} \cdot \sigma \cdot N + 1 \tag{3.29}$$

Since the features of the individual input arrays arrive in sorted order of their coordinates, the number of comparisons in the implementation has been greatly reduced compared to the SoA. This is because when searching through the feature array to see if another feature is within its scale, it can be aborted if the y value is out of range. This reduces the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot \log_2(n))$. However, the theoretical maximum possible latency of the implemented function depends on the actual buffer size (B), the number of input arrays (N) and internal prefetch registers. As shown in Equation (3.30), the theoretical maximum is not equal to the average case. To limit the maximum possible latency and to bring it to a reasonable value, another parameter is introduced. With this parameter the function terminates the output of new features, so that the WCET, until the last output feature is generated, is predictable. However, all remaining input feature arrays must still be read to clear the buffers.

$$\text{WorstCaseLatency} = 1 + B \cdot (N + 1) - ((B \cdot (B + 1) - 6) \cdot 2^{-1}) \tag{3.30}$$

To increase the performance of the feature extract function there is an additional parameter for an internal parallelization. Unlike the vectorization that is available in most other library functions, this parameter does not depend on the vector size of the input and output arrays/images. Using this parameter, the maximum number of comparisons can be increased to two, four or eight. However, due to fragmentation, the full number of comparisons may not be performed in each clock cycle. Like the feature extract function, a parameter that determines how far a feature must be away from the border, can also be specified in the feature compare function.

Feature Gather

The feature gather function merges several input feature arrays into one output feature array. Like the image gather function, tt has two modes, named `cyclic` and `block`. Since the array size of the different inputs can be either indeterminate or different, a separate function is needed. An EOF (End of Frame) is sent for the output array only if all input arrays were read in, but the maximum size of the output was not reached. As long as not all inputs have been read an input feature is read from one of the active ports and forwarded to the output port in each clock cycle depending on the mode. The port of an input is active if it has not read all its input features. In `block` mode this port changes as soon as either the maximum size of the input array has been reached or an EOF feature has been read.

In `cyclic` mode the input port is changed in every clock cycle as long as there is more than one active port. Changing the input port to be read is done in round robin manner. Which input port is next is determined by a bitmask. This bitmask results from the concatenation of the mask of active ports and the mask of all ports not yet read in this period. This mask is then used to read from the input port with the least significant bit that is set to one. As soon as a new input has been read from all active ports, a new period begins. A port is deactivated as soon as its entire input feature array has been read.

The latency of the function is nevertheless equal to the sum of the maximum size of the input arrays. It is not equal to the maximum size of the output array, because all inputs must always be read, to empty the input buffers. Thus, it can happen that after reaching the maximum size of the output array, features of the inputs are thrown away. This is a behavior intended by the user to limit the size of feature vectors to reduce the computational cost of the subsequent functions.

Feature Retain Best

The feature retain best function keeps the K best features from a vector of N input features. It is useful to limit and reduce the number of detected features for later processing, since feature detection is only one part of many computer vision applications. The proposed function requires four loops in its implementation. In the first loop the entries of the internal histograms are reset to zero. In the second loop, the input features are read and a histogram is created out of the feature response values. In addition, all valid input features must be buffered to be reused at a later stage. Two compile time parameters that determine the maximum possible (R_{MAX}) and minimum possible (R_{MIN}) response value are used to determine the histogram characteristics. Equation (3.31) shows the number of histogram entries ($BINS$), which is between 512 and 1024, and the pointer i of a response value (r) to this histogram.

$$\begin{aligned} \text{SHIFT} &= \lceil log_2(\frac{R_{MAX} - R_{MIN}}{1024}) \rceil \\ \text{BINS} &= \frac{R_{MAX} - R_{MIN}}{2^{SHIFT}} \\ \text{i} &= \frac{r - R_{min}}{2^{SHIFT}} \end{aligned} \tag{3.31}$$

All values of Equation (3.31) except i are calculated at compile time. The division required to calculate i is a simple shift operation. The third loop calculates the prefix sum (h_{sum}) of the histogram as long as its value is below or equal to the desired maximum number of output features (K) and stores the corresponding bin entry (h_{bin}). The fourth loop iterates through the buffered input feature vector and writes all features whose bin entry (i) is below h_{bin} to the output. Additionally, ($K - h_{sum}$) input features of the entry ($h_{bin} + 1$) are written to the output if h_{sum} is below K. Equation (3.32) shows the total number of clock cycles of this functions, where P_D is the sum of the pipeline stages of all four loops.

$$Latency = 2 \cdot BINS + 2 \cdot (N + 1) + P_D \tag{3.32}$$

3.1.7 Neural Network Functions

As shown in Table 3.7, seven different neural network layers have been designed and implemented for the library. In addition, it contains three different modules to create the various layers of the MobileNets [21] algorithm. However, these modules consist of the mentioned seven functions of the library. MobileNets was not implemented in a single function because the resulting `dataflow` region is too large for the HLS tool to manage. The implementation of these modules will be discussed in more detail in the next section. The I/Os of the different functions are the input vector, the output vector and, if required, the weights vector and the biases vector. The remaining library function parameters are template parameters, such as the input and output image size, kernel size, IFM (Input Feature Map) and OFM (Output Feature Map).

Table 3.7: `HiFlipVX` neural network functions. Non-standard are marked with ★. The MobileNets modules consist exclusively of the other functions.

Batch Normalization ★	3D-Convolution	Activation
Depthwise Convolution ★	Fully Connected	Softmax
MobilNets Modules 1 - 3 ★	Pooling	

Like the other functions of the library, the streaming capability is the main condition. Thus, large neural networks can be executed on FPGAs without the bandwidth of the main memory becoming a bottleneck. This has some advantages for FPGAs and leads to different implementation possibilities, but also to some limitations. A constraint to achieve maximum performance is that all functions should be pipelined and reach a pipeline interval (P_I) of one. Nevertheless, neural networks are very computationally intensive. Therefore, this thesis explores different approaches of parallelization, to achieve a high performance with an efficient use of resources, as shown by the implementation of the MobileNets algorithm.

All functions in the library have a built-in vectorization that can be applied to their IFM and/or their OFM. Unsigned and signed 8 bit and 16 bit fixed-point and 32 bit floating-point data types are possible for the inputs, outputs, weights, and biases to suit many hardware designs. The size of the fraction can be configured as a parameter of the function. Functions that require trained coefficients buffer them on first use, if configured, to reduce the amount of global memory access. The fixed-point implementations include guidelines for rounding and overflow. If an overflow occurs, the data can either be truncated or saturated to its maximum/minimum value. For fixed-point arithmetic operations, the data can be rounded to zero or the nearest number.

3D-Convolution

The process of 3D-convolution is the most computationally intensive layer in most feed forward networks. The main goal of the proposed image and loop dimension ordering was to achieve a streaming capable function. Under this constraint this thesis developed a structure that is optimized for performance and resource usage. Therefore, the ordering of some dimensions is different from the OpenVX standard. Listing 3.4 shows the general structure of the hardware implementation. Its total latency can be derived from the total number of loop iterations plus the pipeline stages needed for the calculations. The order of the images and coefficient dimensions are:

- Input image (*SRC*): $BATCH \times SRC_{ROW} \times SRC_{COL} \times IFM$
- Output image (*DST*): $BATCH \times DST_{ROW} \times DST_{COL} \times OFM$
- Weights: $OFM \times IFM \times K_Y \times K_X$
- Biases: $(0) \vee (OFM) \vee (BATCH \times DST_{ROW} \times DST_{COL} \times OFM)$

As shown, different sizes for the Bias are possible. Compared to the filter functions, the kernel size does not have to be quadratic ($K_Y \times K_X$). A stride is set if the resolution of the input and output image differs. In the proposed implementation the stride only effects the condition when a result is written to the output. It has no effect to the latency of the function.

```
for $(b = 0)$ to $(BATCH - 1)$
  for $(y = 0)$ to $(SRC_{ROW} + \lfloor \frac{K_Y}{2} \rfloor - 1)$
    for $(x = 0)$ to $(SRC_{COL} + \lfloor \frac{K_X}{2} \rfloor - 1)$
      for $(ofm = 0)$ to $(\frac{OFM}{V_{OFM}} - 1)$
        for $(ifm = 0)$ to $(\frac{IFM}{V_{IFM}} - 1)$
          ReadInputVector()
          UpdateSlidingWindow()
          UpdateBuffers()
          ComputeConvolution()
          WriteOutputVector()
```

Listing 3.4: General structure of the 3D-convolution function. Input Image Resolution ($\mathrm{SRC_{ROW}} \times \mathrm{SRC_{COL}}$), Output Feature Map (OFM), Input Feature Map (IFM), Output Vectorization ($\mathrm{V_{OFM}}$), Input Vectorization ($\mathrm{V_{IFM}}$), Kernel Size ($\mathrm{K_Y} \times \mathrm{K_X}$)

Loop iterations could also be skipped in dependence of the stride. However, the used HLS compiler only allows "perfect loops", which can be a disadvantage when using HLS. The general equation for calculating a 3D-convolution is:

$$DST_{y,x,ofm} = \sum_{ifm=0}^{IFM-1} \sum_{n=0}^{K_Y-1} \sum_{m=0}^{K_X-1} \left(SRC_{(y+n-\frac{K_Y}{2}),(x+m-\frac{K_X}{2}),ifm} \cdot WEIGHT_{ofm,ifm,n,m} \right) + BIAS_{ofm} \tag{3.33}$$

Parallelization: The performance metric for most 3D-convolution layers is the number of multiplications processed per second. On FPGAs, mostly internal DSPs can be used to process the multiplications. When increasing the number of multiplications, the amount of data that is needed simultaneously and thus the required memory bandwidth increases. To implement an efficient streaming capable function, data of the input image as well as the coefficients should be buffered locally. Usually this shifts the external bandwidth problem to the internal buffers. These buffers are typically implemented with BRAMs. BRAMs have a limited bandwidth for reading and writing data. To increase the bandwidth, data can be distributed over several BRAMs. However, this can lead to fragmentation, if the BRAM is not fully utilized and can therefore limit the data to be stored. For this reason, fragmentation should be kept as small as possible while increasing the number of multiplications.

Various loop variables are suitable for parallelization, as illustrated in Listing 3.4. One possibility of parallelization would be in the direction of the (SRC_{COL}) as for the image filter functions. However, this type of parallelization would increase the bit-width of various buffers and therefore lead to a high fragmentation of BRAM. Additional buffers would also have to be introduced to restructure the input and output data to maintain the streaming approach. Therefore, this thesis concentrated on the parallelization of the inner loops, as shown by the parameters (V_{OFM}) and (V_{IFM}) in Listing 3.4. Both (OFM) and (IFM) parallelization would increase the bit-width of the coefficient buffer. Additionally, the parallelization of (IFM) increases the bit-width of the input buffers. In some cases (V_{IFM}) can be raised to a certain point without causing additional fragmentation of the input buffer.

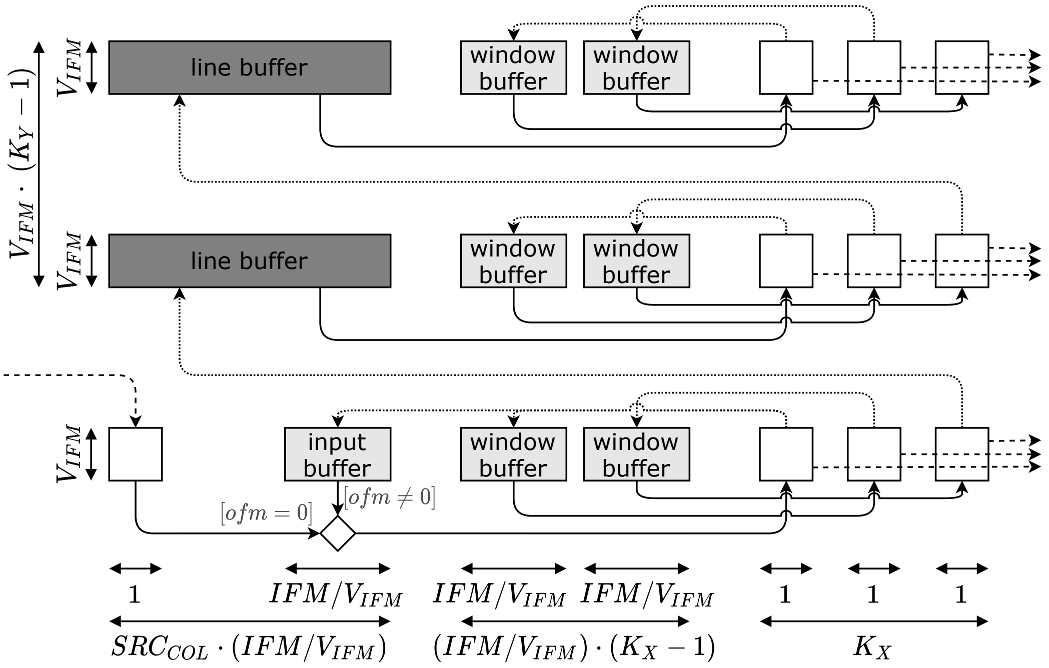


Figure 3.5: The input buffers for the 3D-convolution function with a $K_Y \times K_X$ window/kernel size (here 3×3) and an input vector size of V_{IFM}. The input stage contains input registers on the left (white), big line buffers (dark gray), small input/window buffers (light gray) and sliding window registers on the right (white). The process of buffering the input can be expressed in 4 pipelined stages. 1. reads input vector of size V_{IFM} (dashed lines). 2. updates window registers (continuous lines). 3. updates buffers (dotted lines). 4. sends data to compute stage (dashed lines). Input Feature Map (*IFM*), Input Image Columns (SRC_{COL}).

Structure of Buffers: Figure 3.5 shows the proposed structure needed to buffer the input data to achieve the sliding window effect for the 3D-convolution function. It shows the size of the different buffers, all of which have a depth of (V_{IFM}) elements. Additionally, the image shows the read and write operations between the different blocks by the dashed, dotted, and continuous lines. The line buffers store complete rows of the image including all feature maps ($SRC_{COL} \cdot \frac{IFM}{V_{IFM}}$). Its height of $((K_Y - 1) \cdot V_{IFM})$ elements is stored as one element in BRAM to reduce its usage, to reduce fragmentation. The input buffer is a small line buffer that does not have to store the entire image row. Instead, it is sufficient to only store the ($\frac{IFM}{V_{IFM}}$) elements of the current iteration of (x). The sliding window updates its complete elements in each clock cycle because all feature maps at position x need to be calculated before the window can be moved one element to the right. For this reason, the window buffers are needed, since only one element can be read from each line/input buffer in one clock cycle. Each of the $(K_Y \cdot (K_X - 1))$ window buffers have ($\frac{IFM}{V_{IFM}}$) elements. The different stages of the pipeline of the 3D-convolution are described below in chronological order.

1) Read Input Vector: Reads a vector of V_{IFM} elements from the input image if the following condition is met: $(y \leq SRC_{ROW}) \wedge (x \leq SRC_{COL}) \wedge (ofm = 0)$.

2) Update Sliding Window: In this stage, the data is read from the different buffers and stored in the sliding window, as shown in Figure 3.5 by the continuous lines. Each element in the sliding window of size $K_Y \times K_X$ contains V_{IFM} vector elements. The entire content of the sliding window changes in each clock cycle. This is due to the different feature maps that must be computed before the sliding window can be shifted by one when the loop variable (*x*) is incremented. The left column of the sliding window gets its data from the line buffers and the input buffer. If ($ofm = 0$), new data is read from the input image instead of the input buffer. The other elements of the sliding window get their data from the window buffers. Additionally, the algorithm checks whether valid data should be present in the buffers. Otherwise, a zero is loaded into the corresponding sliding window elements, to apply zero padding. The proposed implementation always applies zero padding of $\left\lfloor \frac{K_X}{2} \right\rfloor$ on both sides in x-direction and of $\left\lfloor \frac{K_Y}{2} \right\rfloor$ on both sides in *y*-direction.

3) Update Buffers: This stage reads the data from the window and writes it to the different buffers, as shown in Figure 3.5 by the dotted lines. The input buffer receives its data from the bottom left element in the window. Since the input data can only be read once, it must be buffered. The line buffer receives its data from the right column of the window in the last iteration at ($ofm = \frac{OFM}{V_{OFM}} - 1$). This moves the data of the image one line up so that it is available again after *y* has been incremented. The window buffer receives its data from the left columns of the window in the last iteration at ($ofm = \frac{OFM}{V_{OFM}} - 1$). The sliding window effect results from the reading and writing between the window buffer and the window.

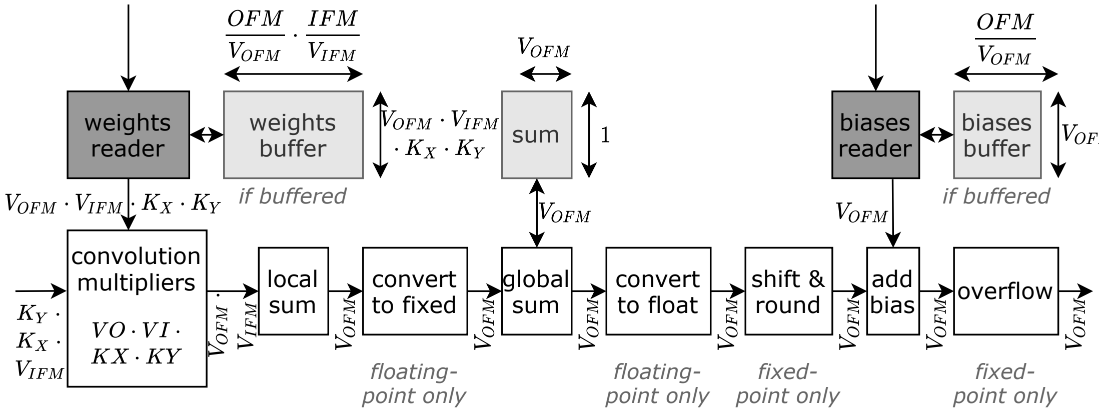


Figure 3.6: Computation stages of the 3D-convolution implementation. The input comes from the sliding window shown in Figure 3.5. Some stages are for floating-point or fixed-point numbers only. Buffers for weights/biases are marked in light gray. Reader functions (dark gray) buffer weights/biases if configured. *IFM* = Input Feature Map; *OFM* = Output Feature Map; V_{IFM} = Input Vector Size; V_{OFM} = Output Vector Size; $K_X \times K_Y$ = Kernel Size

4) Compute Convolution: Figure 3.6 shows the computation stage of the 3D-convolution process. As stated, some of the blocks in the image are only used for fixed-point or floating-point calculations. The gray blocks show the data whose contents needs to be maintained between loop iterations and stored in buffers. The weight and bias coefficients can be buffered within the function if the user sets the appropriate parameter. On first use, they are read from the interfaces and stored in the buffers. If the same coefficients are needed again, they can be accessed from the buffers.

In the first step, the input data is taken from the sliding window and multiplied by the corresponding weights. In total $V_{OFM} \times V_{IFM}$ 2D-convolutions of the size $K_Y \times K_X$ are calculated.

Then V_{IFM} 2D-convolutions of the different V_{OFM} are added together to partially calculate the 3D-convolution of each V_{OFM}.

The operation of calculating a sum over several loop iterations violated the desired pipeline interval (P_I) of one by a factor of five when using floating-point numbers with XILINX tools. Therefore, the library converts floating-point numbers for this summation to a value that is saturated to a 32 bit fixed-point number. The user sets the parameter for the fixed-point position of this variable. In the next step the partial 3D-convolutions are added to the final 3D-convolution until all 2D-convolutions are summed up. Then the result is converted back if the final output should be a floating-point number.

When using fixed-point numbers, the multiplication in the 2D-convolution increases the fixed-point position. Therefore, the value is shifted back to the fixed-point position, while ensuring the overflow policy. This process is done before adding the bias, because it has the same fixed-point position as the output. After adding the bias, the result is checked for overflow and saturated if the corresponding policy is set.

5) Write Output Vector: Writes back a vector of V_{OFM} elements to the output image if the condition of Equation (3.34) is met. The condition includes the stride computation, expressed with the modulus operation. The value for the stride must be an element of the natural numbers.

$$\begin{aligned}\left(0 = \left((y - \left\lfloor \frac{K_Y}{2} \right\rfloor) \bmod (\frac{SRC_{ROW} - 1}{DST_{ROW} - 1})\right)\right) \wedge \left(0 = \left((x - \left\lfloor \frac{K_X}{2} \right\rfloor) \bmod (\frac{SRC_{COL} - 1}{DST_{COL} - 1})\right)\right) \wedge \\ \left(ifm = \left(\frac{IFM}{V_{IFM}} - 1\right)\right)\end{aligned} \tag{3.34}$$

Depthwise Convolution

The depthwise convolution can be considered as a 2D-convolution that is applied to each feature maps of a 3D input image separately. This layer is usually used together with a "pointwise" or 3D-convolution of size 1 × 1, as in MobileNets [21]. This means that for a 3 × 3 3D-convolution, a 3 × 3 depthwise convolution and a 1 × 1 pointwise convolution can be used. The advantage of this approach is that less multiplications and weights are required for the convolution process.

The number of feature maps in the input and output image are the same. Therefore, when comparing with the structure of Listing 3.4, the loop over *OFM* is eliminated. Consequently, the total latency is reduced by that factor and there is only one parallelization term (V_{IFM}). The rest of the basic structure in Listing 3.4 remains. The total number of multiplications and weights is reduced by a factor of *OFM* compared to a pointwise convolution. Therefore, fewer weights must be stored in the internal buffers. On the other hand, the size of biases remains unchanged. It is still possible to choose between the different bias sizes.

Compared to the structure in Figure 3.5 nothing changes for the buffering of the input image and the sliding window. As pointwise convolution only performs 2D-convolution operations, Figure 3.6 omits the summation blocks. This eliminates the need for inter-loop summation and the conversions for floating-point numbers. Except for the number of weights and convolutions, the rest of the structure in Figure 3.6 remains. The conditions when a vector is read from the input image or when it is written to the output image only changes in such a way that the following conditions are omitted: $(ofm = 0)$ for the input and $(ifm = (\frac{IFM}{V_{IFM}} - 1))$ for the output. This also implies that the stride calculation remains the same.

Pooling

The purpose of the pooling layer is to reduce the spatial size of the image to reduce the number of parameters and calculations in the neural network. The pooling operation works independently on each feature map. Like the 2D-convolution a window slides over an input image. To calculate the output, the values in the window are either averaged or the maximum value is taken. With the help of a stride, pixels can be skipped so that the output image becomes smaller than the input image. The following equation is used to calculate the stride:

$$STRIDE_X = \left\lfloor \frac{SRC_{COL} + 2 \cdot PAD_X - K_X}{DST_{COL}} \right\rfloor, \quad STRIDE_Y = \left\lfloor \frac{SRC_{ROW} + 2 \cdot PAD_Y - K_Y}{DST_{ROW}} \right\rfloor \tag{3.35}$$

The window $(K_X \times K_Y)$ can have any size between 1×1 and $SRC_{ROW} \times SRC_{COL}$. Like in the convolution filters zero padding can be applied. The padding size (PAD_X) is in the range between 0 and $\left\lfloor \frac{K_X}{2} \right\rfloor$. The size of the stride $(STRIDE_X)$ is in the range between 1 and K_X. The overall structure of the function is very similar to the pointwise convolution, without the need of buffering coefficients. The total latency only differs slightly, since the padding size is not fixed: $(SRC_{ROW} + PAD_Y) \times (SRC_{COL} + PAD_X) \times \frac{IFM}{V_{IFM}}$. For average pooling, the sum of all window elements is calculated and then multiplied for normalization. Fixed-point values need an additional operation that shifts the result back to the desired fixed-point position (FP).

$$DST_{x,y,ifm} = \left\lfloor \left\lfloor \sum_{n=0}^{K_Y-1} \sum_{m=0}^{K_X-1} \left(SRC_{(y+n-\frac{K_Y}{2}),(x+m-\frac{K_X}{2}),ifm} \right) \cdot \underbrace{\left\lfloor \frac{2^{FP}}{K_Y \cdot K_X} \right\rfloor}_{normalization} \right\rfloor \underbrace{\cdot 2^{-FP}}_{shifting} \right\rfloor \tag{3.36}$$

Activation

The activation layer is a crucial component in CNNs. In general, the function is connected to each neuron in the network and determines whether it should be activated or not. Table 3.8 shows the nine implemented activation functions, which have been defined by the OpenVX standard. For fixed-point numbers, the overflow policy needs to be applied to the following functions: `soft relu`, `square` and `linear`. In addition, an overflow can occur when calculating the absolute function for a signed data type. For fixed-point numbers, the rounding policy must be applied to the following functions: `logistic`, `soft relu`, `square` and `linear`. The logarithmic and exponential activation functions are computed using floating-point operations, due to the high range of possible values and the resulting accuracy loss when using fixed-point numbers. Therefore, conversions are needed for fixed-point input and output images using multiplication operations. As shown in the table, the hyperbolic tangent function is calculated with one repeated exponential function and one division to the reduce resource usage. The activation function can be computed in parallel (V_{IFM}) in a SIMD manner on the 3D input image. The latency of the hardware function is: $SRC_{ROW} \cdot SRC_{COL} \cdot \frac{IFM}{V_{IFM}} + P_D$.

Table 3.8: Implemented activation functions that compute on an input pixel at position *x*. *a* and *b* are compile time parameters.

Name	Operation
logistic	$f(x) = \frac{1}{e^{-x}}$
hyperbolic tangent	$f(x) = a \cdot tanh(b \cdot x) = a \cdot \frac{e^{2 \cdot b \cdot x} - 1}{e^{2 \cdot b \cdot x} + 1}$
relu	$f(x) = max(0, x)$
bounded relu	$f(x) = min(a, max(0, x))$
soft relu	$f(x) = log(1 + e^x)$
abs	$f(x) = \lvert x \rvert$
square	$f(x) = x^2$
square root	$f(x) = \sqrt{x}$
linear	$f(x) = a \cdot x + b$

Batch Normalization

Batch normalization [254] is a technique to improve the stability and performance in neural networks. The core idea is that the inputs of each layer of an image are normalized so that the mean output activation is zero and the standard deviation is one. The batch normalization calculates a mini-batch (B) over a set of pixels values (χ_{ifm}): $B = \{\chi_1, \chi_2, ..., \chi_{IFM}\}$. Considering a three dimensional input image ($SRC_{x,y,ifm}$), the mini-batch would be calculated over the third dimension of size IFM. It first calculates the mean (μ) of the pixel values, as shown in Equation (3.37). Using the the mean value, the variance (σ^2) is calculated, as shown in Equation (3.38). Using the mean, variance, and a set of pretrained values ($\gamma_{ifm}, \beta_{ifm}$), the output image pixels are calculated, as shown in Equation (3.39).

$$\mu = \frac{1}{IFM} \cdot \sum_{ifm=1}^{IFM} \left(SRC_{y,x,ifm} \right) \tag{3.37}$$

$$\sigma^2 = \frac{1}{IFM} \cdot \sum_{ifm=1}^{IFM} \left(SRC_{y,x,ifm} - \mu \right)^2 \tag{3.38}$$

$$DST_{y,x,ifm} = \gamma_{ifm} \cdot \frac{SRC_{y,x,ifm} - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta_{ifm} \tag{3.39}$$

A straightforward way to compute this function would be in three separate loops iterating over the third dimension, nested in the loops iterating over the first and second dimensions. With this approach only one output pixel is generated every three clock cycles for a parallelization degree of one. Therefore, the function contains three subfunctions to compute μ, σ^2 and the result, which are used in a pipelined manner inside the three nested loops. As a result, the overall latency is as follows: $(SRC_{ROW} \cdot SRC_{COL} + 2) \cdot \frac{IFM}{V_{IFM}} + P_D$. Two times $\frac{IFM}{V_{IFM}}$ additional clock cycles are required, since each mini-batch must pass through these three stages in a pipeline manner without using the `dataflow` directive. The input data of a mini-batch (*B*) is stored in a buffer in the first stage to be used for the next two stages. Since there are three stages, three consecutive input vectors of size *IFM* must be stored in buffers. The weight vectors γ and β are read in the third stage at the first use and buffered for further usage.

To calculate μ and σ^2 a sum of values must be computed. Like in the convolution filter, the sum is computed using fixed-point numbers, because a floating-point sum increases the latency by a factor of five. Therefore, floating-point numbers are converted and saturated to a 32 bit wide integer value. Since the normalization ($\frac{1}{IFM}$) is a constant, it can be precomputed to replace the division by a multiplication. Both calculations are easy to vectorize, since only the sum needs to be parallelized. For the parallelization of the last stage which computes the output pixel, the term $\frac{1}{\sqrt{\sigma^2+\varepsilon}}$ can be precomputed once. This has a big impact on the resource usage when vectorizing the function, since the division and square root are the most resource consuming functions. Due to the accuracy, $\frac{1}{\sqrt{\sigma^2+\varepsilon}}$ is calculated using floating-point numbers.

There are different variants of how to compute the Batch Normalization. One of them avoids the calculation of μ and σ^2. In this variant, the two values are passed to the function as additional parameters, as shown in Equation (3.40). Since both values are constants, the value of c_{ifm} can be precalculated after training the neural network, as shown in Equation (3.40). As a result, this variant of the batch normalization is very resource-efficient. The latency of the hardware function is: $SRC_{ROW} \cdot SRC_{COL} \cdot \frac{IFM}{V_{IFM}} + P_D$.

$$DST_{x,y,ifm} = \gamma_{ifm} \cdot (SRC_{x,y,ifm} - \mu_{ifm}) \cdot c_{ifm} + \beta_{ifm}, \quad c_{ifm} = \frac{1}{\sqrt{\sigma^2_{ifm} + \varepsilon}} \tag{3.40}$$

Fully Connected

The fully connected layer is an essential component of most CNNs. It is one of the last layers and is used for the final classification decision. Simplified, it is a 3D-convolution with a 1×1 kernel on an image with a resolution of 1×1. However, the *IFM* and *OFM* can be very large. The weights, biases and input image are buffered on first use. However, since each weight/bias is read only once per image, it is recommended not to buffer them if the weight matrix becomes too large. The summation of Equation (3.41) has been implemented using fixed-point numbers for the floating-point implementation. Therefore, the multiplication result is converted and saturated to a 32 bit wide number before summation and converted back afterwards. Fixed-point numbers were used, since a summation with floating-point numbers increased the total latency by a factor of five. The fixed-point position is set by a parameter. Depending on the degree of parallelization, V_{IFM} multiplications are calculated in parallel and added together. After summation, the data must be shifted back due to the fixed-point multiplication according to the rounding policy. Then the bias is added. When using fixed-point values, the result is converted back to the output format according to the overflow policy. The latency of the hardware function is: $OFM \cdot \frac{IFM}{V_{IFM}} + P_D$.

$$DST_{ofm} = \sum_{ifm=1}^{IFM} (SRC_{ifm} \cdot WEIGHT_{ofm,ifm}) + BIAS_{ofm} \tag{3.41}$$

Softmax

The Softmax layer normalizes an input vector into a probability distribution and limits the output to a range between zero and one. It is used to determine the probability of several classes at once. The calculation shown in Equation (3.42) is done in two parts. The first part computes the sum and stores the exponents of the inputs into a buffer. Due to the high range of values in this function all operations are done using floating-point numbers. However, for the same reason as in the previous functions, the summation is calculated using fixed-point numbers. Therefore, the exponent result is converted and saturated into a 32 bit fixed-point number before summation. For each element in the input vector, V_{IFM} exponents are calculated, stored, and added to the summation. The second part calculates the division of Equation (3.42). For fixed-point numbers, the division result must be shifted (multiplied) to the correct position according to the rounding policy. Depending on the parallelization degree, V_{IFM} output elements are computed. The latency of the hardware function is: $2 \cdot \frac{IFM}{V_{IFM}} + P_D$.

$$DST_{ifm} = \frac{e^{SRC_{ifm}}}{\sum_{i=1}^{IFM}(e^{SRC_{ifm}})} \tag{3.42}$$

3.2 Object Detection Algorithms

This section focuses on the investigation and implementation of the more complex object detection algorithms of this thesis. The background to these algorithms has been described in Section 2.1. In a preliminary investigation, this thesis combined and compared different feature extraction algorithms. This resulted in an optimized algorithm, which is based on the AKAZE [20] feature detector and FREAK [24] feature descriptor. Based on the comparison, the ORB [19] and AKAZE feature detectors, and the FREAK descriptor stood out. First, these three algorithms were implemented in a highly optimized and specialized design in VHDL. With the help of this implementation, several generic HLS-based functions have been created for the `HiFlipVX` library. Using the library, five different algorithms have been implemented: FAST [37] corner detector, Canny [18] edge detector, ORB feature detector, AKAZE feature detector and MobileNets [21] neural network. The remainder of this section will describe the optimized feature extraction algorithm, the three VHDL-based implementations, and the five HLS-based library implementations.

3.2.1 Proposed AKAZE and FREAK based Feature Extraction Algorithm

This section focuses on the proposed feature extraction algorithm, which is based on the AKAZE feature detector and the FREAK feature descriptor. AKAZE is a multiscale feature detection and description algorithm, which creates a nonlinear scale-space using a numerical scheme called FED. The scale-space in AKAZE is divided into octaves (*O*), which in turn are divided into sublevels (*S*). It detects features using a blob detector and describes them with the M-LDB descriptor. This thesis compared the M-LDB descriptor of AKAZE with other descriptors, such as BRIEF, BRISK, or FREAK. The FREAK descriptor is generally faster to compute, has a lower memory load, and is more robust than the other algorithms. It is inspired by the human visual system, more precisely by the retina. It computes binary strings by comparing image intensities over a retinal sampling pattern and adds rotational invariance. The biggest advantage over the M-LDB descriptor is the low memory load. This is because the M-LDB descriptor must store the computed images and derivatives of all scale-space levels in memory, which is critical in an embedded system with limited memory resources and bandwidth.

Both AKAZE [255] and FREAK [256] are provided by their developers as OpenCV-based open-source implementations. This thesis implemented and combined the algorithms in C without external libraries, such as OpenCV, so that they can be used on any embedded device. Several modifications have been made to the algorithms to reduce computation time and increase repeatability, which was evaluated using the Oxford [257] dataset. This work added a function that retains the best features of the feature detector to increase repeatability and reduce WCET, when computing the descriptor. Listing 3.5 shows the pseudocode of the software implementation. The precalculations, which must be performed only once for all images, are not included in this listing. The software implementation normalizes all functions and uses 32 bit floating-point for a high accuracy.

Contrast Factor: The first part of the algorithm calculates the contrast factor. First, the image is smoothed with a 5 × 5 Gaussian filter, and then the gradient magnitude is calculated using the first order derivatives with a Scharr filter. Then a histogram of size 1024 is created from the gradient image. The contrast factor is the index (*bin*) at which 70 % of the gradient histogram is reached. Unlike the original algorithm, the maximum gradient value is not calculated in advance, since this work normalizes all filter functions and thus know the maximum possible value of the implementation. In the original implementation, the entire image must be traversed twice, once to find $hist_{max}$ and another time to fill the histogram. Instead, it is only necessary to iterate through the image once in the modified design. Equation (3.43) describes the calculation of the contrast factor, where $hist_{max}$ is the maximum gradient value, bin_{max} is the maximum histogram value, and *O* is the octave. It already calculates the square of the contrast factor to be used in the conductivity function.

$$contrast\_square_O = (\frac{hist_{max} \cdot bin}{bin_{max}})^2 \cdot 0.5625^O \tag{3.43}$$

Nonlinear Scale-Space: In the second part, a nonlinear scale-space is created. It computes a flow image (*Lc*) with the conductivity function using the squared contrast factor (*contrast_square*) and the image derivatives (*Lx*, *Ly*) computed by a Scharr filter as shown in Equation (3.44). To generate a new image (*Lt*) in scale-space, a number of FED time steps (*N*), which increase for each level, need to be processed. The image (*Lt*) is scaled each time a new octave begins by averaging a 2 × 2 pixel window to one pixel (fast area interpolation). Finally, the smoothed image (*Ls*) is created using the Gaussian kernel. For the proposed configuration, the number of FED-steps (*N*) is zero in the first two scale levels.

```
 // Contrast Factor
Ca = Gaussian5x5(image)
Cb = Gradient(Ca)
contrast\_square = ContrastFactor(Cb)

// Nonlinear scale-space
Ls[0]    = Gaussian7x7(image)
Lt[1][0] = Copy(Ls[0])
Ls[1]    = Gaussian5x5(Lt[1][0])
for (i = 2) to (O · S - 1)
   Lc[i-1] = Conductivity(Ls[i-1], contrast\_square[i-1])
   for (j = 2) to (N[i-1] - 1)
      Lt[i-1][j+1] = FastExplicitDiffusion(Lt[i-1][j], Lc)[i-1])
   Lt[i][0] = ((i mod S) == 0) ? (Scale(Lt[i-1][N[i-1]])) : (Copy(Lt)[i-1][N[i-1]]))
   Ls[i] = Gaussian5x5(Lt[i][0])

// Feature Detection
for (i = 0) to (O - 1)
   for (j = i · S) to ((i + 1) · S)
      Lx/Ly[j] = Scharr(Ls[j])
      Ld[j] = DeterminantOfHessian(Lx)[j], Ly[j])
   for (j = i · S) to ((i + 1) · S)
      Fa[j] = FeatureExtract(Ld[j])
   for (j = i · S) to ((i + 1) · S)
      Fb[j] = (j == 0) ? (FeatureCompare(Fa[j])) : (FeatureCompare(Fa[j-1], Fa[j]))
 features = FeatureRetainBest(Fb)

// Feature Description
integral = IntegralImage(image)
for (i = 0) to (features.size - 1) // Freak Algorithm
   Da = Intensity(integral, features[i])
   Db = Orientation(Da)
   Dc = Intensity(integral, features[i], Db)
   descriptor[i] = Descriptor(Dc)
```

Listing 3.5: Pseudocode of the proposed AKAZE-FREAK optimized algorithm. O (number of octaves), S (number of sublevels), N (number of Fast Explicit Diffusion time steps).

$$Lc = \frac{contrast\_square}{contrast\_square + Lx^2 + Ly^2} \tag{3.44}$$

Feature Detection: Features are detected with the DoH blob detector ($Ld = Lxx \cdot Lyy - Lxy^2$). The first ($Lx, Ly$) and second ($Lxx, Lyy, Lxy$) order derivatives are calculated using Scharr filters. The `FeatureExtract` function generates a feature if a pixel value (response) of the Ld image is the maximum in a 3×3 window and above a certain threshold (t_0). It also calculates SR for more accurate coordinates. This work merges the SR with the `FeatureExtract` function, since both require the surrounding response values. Originally, this was the last function of the feature detection algorithm.

A feature passes the `FeatureCompare` function if its response value is maximum in a σ radius, to avoid replication of adjacent features. Therefore, in the original AKAZE implementation, a feature had to be compared with all other features of the same, upper, and lower level. Unfortunately, this method has a complexity of $\mathcal{O}(n^2)$ for n features, which makes it very computationally intensive. The proposed implementation takes advantage of the fact that features are detected in ascending order with respect to vertical position. Therefore, adaptive search pointers are used to determine the beginning and end of the features to be compared. This decreases the complexity from $\mathcal{O}(n^2)$ to approximately $\mathcal{O}(n \cdot log(n))$. This work compares the response values of the current level with the lower level only if it is in the same octave, and do not mix between octaves as in the original algorithm. The implemented design also dropped comparisons to the upper scale level, since the number of matches only differs by about 1 % and this had almost no measurable effect to the repeatability. To further reduce the number of comparisons, this work performs the border check required for the FREAK retina pattern at this stage. The `FeatureRetainBest` function concatenates all features and partially sorts them by their response value to get the N best features. Checking the FREAK borders before the `FeatureRetainBest` function improves repeatability by generating a stable number of features.

Feature Description: The FREAK algorithm needs an integral image ($I[x, y]$) to calculate the sum of pixels ($i[x, y]$) needed for each retinal pattern. Instead, creating a separate integral image from each scale level of the AKAZE algorithm did not improve repeatability. One possible explanation why this approach does not help, is the loss of data due to the diffusion process and the scaling of the image. The original algorithm for this function is shown in Equation (3.45). However, the optimized software implementation first sums each row and then each column to reduce loop dependencies and apply parallelization.

$$I[x, y] = i[x, y] + I[x, y-1] + I[x-1, y] + I[x-1, y-1] \tag{3.45}$$

Parallelization: To reduce the computation time of the original implementations, this work merged loops and added OpenMP directives for parallelization, for all functions except `FeatureRetainBest` and `ContrastFactor`. For the `FeatureExtract` and `FeatureCompare` functions, it creates a thread for each of the $(O \cdot S)$ levels due to loop dependencies within these functions. Descriptors are created independently for each feature, which eases the loop parallelization of the FREAK algorithm. To parallelize the integral image, the implementation splits the function into two loops, to which OpenMP loop parallelization was applied. The first one computes the prefix-sum of each row independently. The second one takes this intermediate image to create the prefix-sum of each column independently. For all other functions, a standard OpenMP loop parallelization is used. The different filters replicate the border values of their input images when their kernel is outside the image boundaries, which improves the repeatability of the overall algorithm. The hardware implementations of AKAZE and FREAK algorithms will be explained in more detail in the following subsections.

3.2.2 FAST Corner Detector

The FAST [36, 17] corner detector is one of the implemented feature detection algorithms of the `HiFlipVX` library. It is part of the ORB feature detection algorithm, which is explained in a later subsection. As shown in Figure 2.3, it extracts corners from images by evaluating the Bresenham circle around a pixel. The hardware implementation consists of the three `HiFlipVX` library functions shown in Figure 3.7.

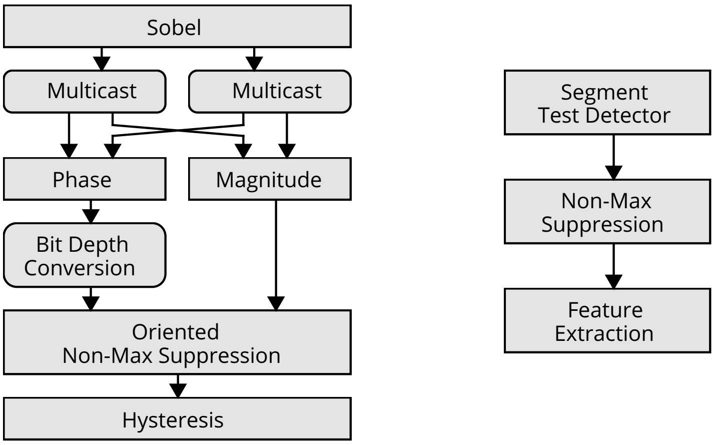


Figure 3.7: Canny edge detector (left) and FAST corner (right) detector HLS implementations.

First, the segment test detector function takes the input image and computes the response values using the FAST9 method. Then the NMS function takes the response values and suppresses pixels in a 3×3 window that are not the maximum in this window. Then the feature extract function takes this image and creates a vector of features. A pixel becomes a feature if its response value is above a certain threshold (t_0). Also, this pixel must have a minimum distance of four to the edges to be stored as a feature. This is due to the window sizes of the previous filters. Therefore, the previous two filters do not require special border handling and can use `undefined` border handling to save further resources. An alternative design of the FAST uses the integrated NMS function of the feature extraction function to save resources.

3.2.3 Canny Edge Detector

The Canny [18] edge detector is also a part of the `HiFlipVX` library. It detects and highlights edges in images and suppresses all other information. The hardware implementation consists of several library functions, as shown in Figure 3.7.

First, a Sobel filter is applied to the input image, which calculates the derivatives in x and y direction and converts the image data type from unsigned to signed. Then, the magnitude and orientation images are calculated using the derivatives. The phase function computes the orientation to detect features regardless of their rotation (invariance). The orientation of an edge is its direction and can be visualized as a line perpendicular to it. Due to the concept of "*one consumer and one producer*", the intermediate results must be duplicated using the multicast function. Based on the orientation and magnitude values of a pixel, non-edge pixels are suppressed from the image. The pixels are then converted to unsigned values bit-depth conversion function. Finally, the hysteresis function highlights all strong pixel values and their weak neighbors and suppresses the rest of the pixels to create a binary image.

In addition to the hysteresis threshold value, the Sobel and hysteresis kernel sizes of the main function are parameterizable to adapt the detector to the image environment. To reduce edge effects, the Sobel and NMS functions use replicated borders, and the hysteresis function uses a constant border handling. As a result, no edges can be highlighted at the border of an image, to avoid false edge detection.

3.2.4 ORB feature detection

This subsection will discuss the VHDL and HLS implementations of the ORB feature detection algorithm. This thesis replaced the BRIEF descriptor of the ORB algorithm with the FREAK descriptor to improve repeatability and performance. The ORB detector is a FAST detector supplemented by a pyramid scheme that makes the detector scale invariant. Scale invariance is needed to detect similar features in different images independent on their size.

Some changes were made to the original algorithm. Before, all configuration parameters were evaluated in the software to tune the two algorithms. Due to the replacement of the BRIEF with the FREAK descriptor, the intensity centroid function has been removed from the implementation. This is due to the different calculation of the feature orientation of the FREAK algorithm. In addition, the Harris [38] detector was excluded according to the optimal configurations resulting from the software optimization. Additionally, it retains the k best features for every scale to reduce the maximum number of detected features, to evenly distribute the detected features to the scales and to keep only the strong features. This function strongly increases the repeatability of the detected features.

In the following, the VHDL implementation is described first. Then, the integration of the VHDL implementation into a VPS (Video Processing System) is shown. In the last part, the resulting HLS implementation, and its differences from the VHDL implementation will be discussed.

VHDL Implementation

In this part, the proposed VHDL implementation of the ORB feature detection algorithm is explained. Figure 3.8 gives an overview of the pipelined hardware implementation for a 4-level pyramid. A single level of the pyramid scheme can be seen on the right of the same figure. The various modules of the hardware design will be explained below.

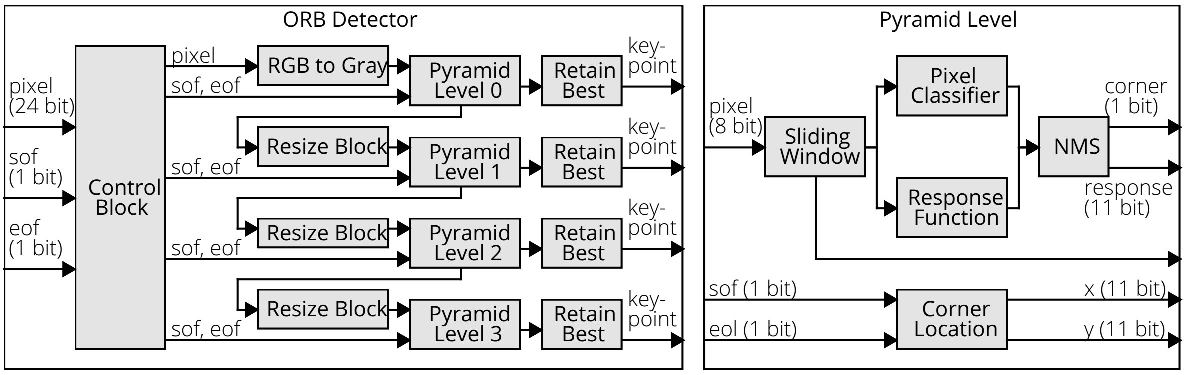


Figure 3.8: Hardware design of the ORB detector. SoF (Start of Frame), EoL (End of Line)

RGB to Grayscale Conversion: This block converts the input pixels from the RGB color format (24 bit) to the grayscale format (8 bit). For its computation, the *ITU-R BT.709-6* standard has been chosen: $g = (0.21093 \cdot R) + (0.71484 \cdot G) + (0.07031 \cdot B)$. The coefficients consider how the human eyes perceive the different color components. This block has a pipeline depth of two clock cycles and a throughput of one pixel per cycle. It has a fixed-point logic, which has a small loss in precision.

Sliding Window Structure: First, the 16 pixels of the Bresenham circle need to be investigated. Therefore, a 7×7 pixel window of the input frame is needed. This window scans the complete input frame pixel by pixel. It only excludes a three pixel wide border (window radius) of the frame. During execution, the last seven rows of the frame are buffered. In every clock cycle, the read and write addresses of these buffers are incremented, while the read address is always one higher than the write address. A row buffer is implemented using BRAMs and its depth is equal to the width of a frame minus seven (window size). Every row buffer is connected to seven shift registers (window), which are connected to the next row buffer. This way, the window slides through the frame and scans it pixel by pixel. This structure causes a delay of seven clock cycles at each row change.

Pixel Classifier and Response Function: The output of the sliding window are the 16 Bresenham circle pixels and the pixel under test. In the next step, a segment test is performed to determine whether the pixel under test is a corner or not. Therefore, the pixels of the Bresenham circle are treated like a vector. From this vector, nine contiguous pixels must satisfy the following condition: the output is equal to 1 if ($I_{p \to x} \geq I_p + t$) or ($I_{p \to x} \leq I_p - t$), where $I_{p \to x}$ is a pixel from the Bresenham circle, I_p is the value of the pixel under test and t is the FAST threshold. This is achieved by using a logical AND for every nine contiguous results. Therefore 16 AND gates are used, whose outputs serve as inputs for an OR gate with 16 inputs. The final output is a signal indicating whether the pixel under test is a corner or not. Furthermore, the score (response) of the pixel under test is calculated for the NMS stage. The value of pixel under test I_p and value of every Bresenham-vector pixel passes through a group of subtractors to calculate the response function. Another output signal was implemented to identify whether the circle is darker or brighter than the pixel under test.

NMS (Non-Maximum Suppression): In the next stage, keypoints that do not have a maximum response value in a 3×3 window are suppressed. The sliding window structure as described above is used, but modified for a 3×3 window. The response of the pixel under test is compared with the other responses in the window and suppressed if the pixel under test is not the local maximum. The eight comparisons are executed in parallel, and the results are combined by a logical AND.

Corner Location: The output from the previously described design is a stream of responses and a signal that indicates whether a response is a corner or not. However, there is no information regarding the coordinates. The `Corner Location` block generates the coordinates synchronized with the stream of output responses. Using the SoF (Start of Frame) and EoL (End of Line) signals, the delay of the previous blocks can be calculated to generate the coordinates of the output response. This core also generates a `valid` signal used in NMS block.

Resize Block: This block scales one pyramid level into a smaller one. To reduce resource consumption, a simple method called fast area interpolation is used, where a 2×2 window of pixels is averaged to form a new pixel. This is done by adding the values of the four pixels and dividing the result by four (shift right by two bits). Accordingly, the width and height of the frame are halved (scale factor of two). The input window is obtained from the `Sliding Window` block, to save resources.

Control Block: A control block is needed to integrate and control the different pyramid levels. It is responsible to generate different control signals for the pyramid levels (SoF, EoL, enable and reset). It waits for the SoF signal and then starts controlling the pyramid levels. Each level contains a state machine, which starts in a waiting state until the previous level launches the start flag after buffering two lines (`Resize Block`) and goes into enable state. In enable state, the enable signal of that pyramid level is asserted ($= 1$). The enable state starts a group of counters, which count the number of columns and rows of the previous level and the number of pixels and lines of the current level. A set of customizable parameters specifies the size of every pyramid level and a parameter called `Stop_Num`. This parameter indicates that the enable stops every time the counter is equal to that number (horizontal or vertical). For example (`Stop_Num = 4`) means that every four counts (either horizontal or vertical) the enable stops. In case of horizontal stop, the FSM (Finite-State Machine) de-asserts the enable just for one cycle. In case of vertical stop, the FSM de-asserts the enable for one complete line of the previous level. `Stop_Num` can be calculated using this following equation, where SF is the scale factor: $stop_{num} = \left\lfloor \frac{SF}{SF-1} \right\rfloor$

Retain Best: The `Retain Best` block is shown in Figure 3.9. It gets (retains) the K best of N keypoints. This is useful to reduce the calculations for the later feature descriptor and reduce the percentage of false matches. The keypoints are filtered according to their response value. The execution is done in three stages.

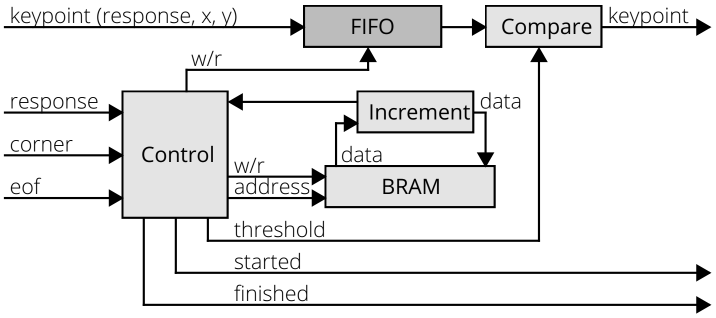


Figure 3.9: Hardware design of the Retain Best block.

In the first stage, all incoming keypoints are buffered in a FIFO. At the same time, a histogram of the keypoint responses is built. This histogram is implemented using a BRAM such that the response is the address of this BRAM. For each address, the value is read, incremented, and rewritten to the same address. When the last keypoint is detected, a signal called EOF is asserted. In the second stage, the histogram is scanned cell by cell, starting from the maximum possible response going downwards, while counting the number of keypoints in each cell. The threshold response is the address of the last scanned cell, when the number of counted keypoints reaches the number to retain. The third stage reads from the FIFO and filters the elements using the threshold response from the second stage. A keypoint is passed to the output if its response value is greater than or equal to the threshold.

Note that several keypoints can have the same threshold response. In this case the number of filtered keypoints can exceed N. Also, note that there is a maximum FIFO depth, which limits the maximum number of keypoints that can be accepted from the previous block. The worst-case execution time of this block would be equal to the histogram depth plus the FIFO depth (e.g., 8192). Two more signals are generated indicating the start and finish of the FIFO filtering stage. This `Retain Best` block is used in every pyramid level, which gives the ability to specify the number of retained keypoints in every level independently.

Integration into a Video Processing System

The VPS is created using the tutorials, IP-cores and codes provided by Avnet. It is a HDMI (High Definition Multimedia Interface) pass-through system that receives an input stream, stores it in frame buffers in DDR memory using a video DMA block, reads it from these frame buffers via video DMA, and sends it as an output stream. The ARM processor is used to configure the video DMA blocks and control memory access. The Avnet IP cores provide a video stream in YCbCr 4:2:2 color format.

Therefore, additional blocks were added to this system to interface with the feature detection algorithm, since the hardware design operates in the `RGB` color space. The first core (`Chroma Resampler`) converts the input to the YCbCr 4:4:4 representation. The second core (`YCrCb to RGB Color-Space Converter`) converts its input to the 24 bit `RGB` representation. The third core (`AXI4-Stream Subset Converter`) adds a dummy zero-byte, to store the `RGB` pixels in an aligned manner in DDR memory. The same IP-cores are used for HDMI input and output in reverse order.

For proof of concept, the 4-level hardware design was integrated into a VPS. The optimal 8-level pyramid was designed and simulated, but not integrated into the VPS. The detector is integrated via an AXI4 stream interface to communicate with other IP-cores. As shown in Figure 3.10, the ORB detector is placed in the input stream path. To avoid buffering the image, the input video stream is duplicated. Two methods were investigated to display the detected keypoints on the output monitor. The first method implements an IP-core that draws the detected keypoints on the frame in hardware. The drawback is that to draw a clear circle of a reasonable size, a large number of image rows must be buffered, which consumes many resources and causes additional delays. The second method writes the detected keypoints to DDR and performs the drawing on the ARM processor, which consumes fewer resources. Note that this would be removed when integrating the detector in a larger system.

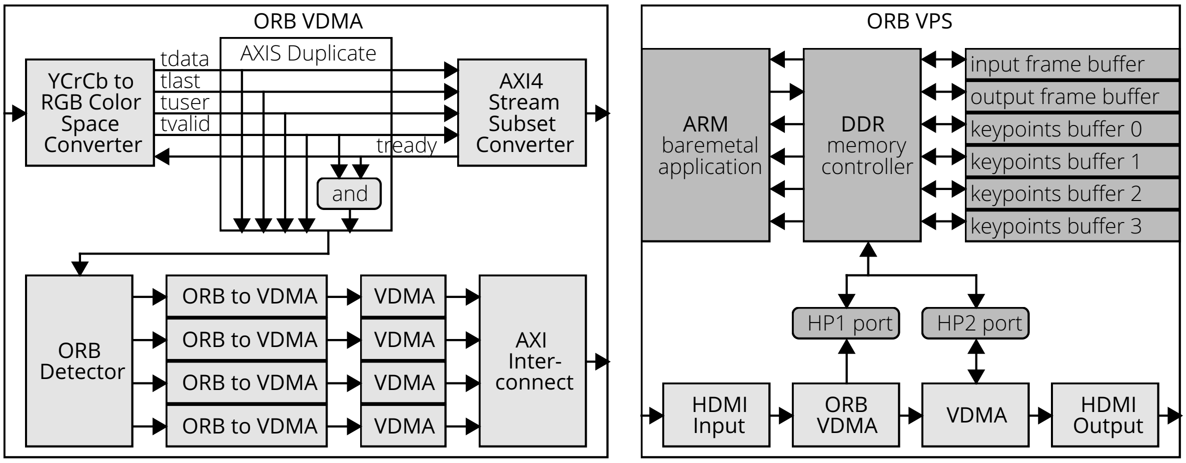


Figure 3.10: Integration of the ORB detector into the video processing system.

Figure 3.10 contains the hardware design of the second method for a 4-level pyramid. This work created an IP-core that interfaces with the `Retain Best` module of the ORB detector. This IP-core buffers the data in a FIFO and converts it to an AXI4-Stream-protocol, to be connected to a video DMA. Both blocks are needed for each level of the pyramid scheme. The ARM processor controls the video DMA blocks and stores each level into an own frame buffer. The video DMA blocks send the keypoints through an AXI interconnect to DDR memory. The

ARM processor performs the drawing of the keypoints, by continuously copying the input frame into the output frame and drawing the keypoints on the output frame.

HLS Implementation

The ORB feature detector and its functions are also part of the `HiFlipVX` library. Figure 3.11 shows its structure. It is a multiscale algorithm and therefore detects features of different scales using the FAST corner detector, described in Section 3.2.3. Due to the generalization of the reusable library functions, there are some differences in comparison with the VHDL implementation. For example, the scale image function from the library has different scaling options (bilinear, area or nearest neighbor interpolation), whereas in the VHDL implementation a fast area interpolation was used to save resources. It is not part of the ORB algorithm, but the `RGB` to grayscale conversion from the VHDL implementation can be solved by the color conversion function from the library.

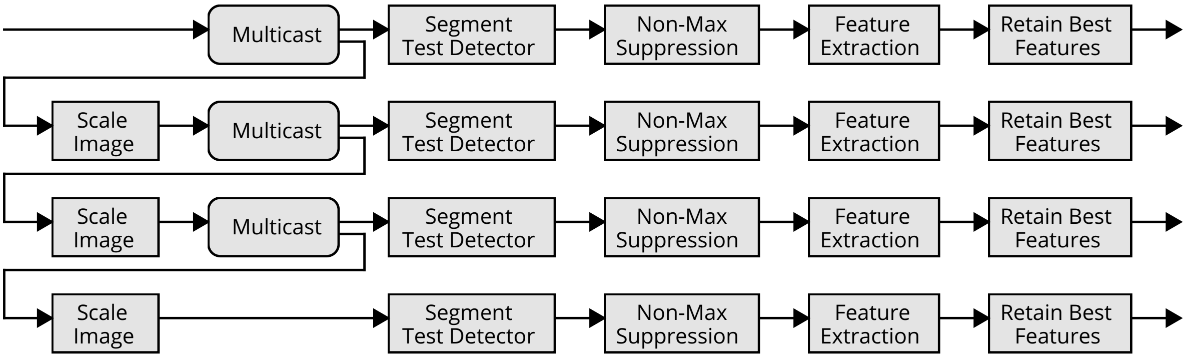


Figure 3.11: Overview of the ORB feature detection implementation.

In Figure 3.8, the FAST algorithm was represented by the `Pyramid Level`. In the HLS implementation, this part is represented by the segment test detector, NMS, and feature extraction functions. Thereby, the `Corner Location` and the `Control` blocks, are replaced by the feature extraction function. However, this makes the format of a feature more generic, since the orientation is added, and the different elements have common sizes. Due to the streaming approach and the independent functions, no higher-level control block is needed in the library approach. The `Pixel Classifier`, the `Response Function` and the `Sliding Window` are part of the segment test detector. However, the library extracted the threshold from this function and executes it in the feature extraction function. In general, the threshold should be executed as late as possible, since it deletes information that is then lost. The many additional options provided by the parameters of the library functions allow to increase the repeatability at the cost of resources or to easily adapt the algorithm for different use cases. In addition, all functions except the feature retain best function can be vectorized, which results in higher performance. Due to the lower latency, there is no need to vectorize this function.

3.2.5 AKAZE feature detection

In this subsection, the different implementations (VHDL, HLS) of the AKAZE algorithm are described and compared.

VHDL Implementation

As mentioned earlier, this thesis reimplemented the AKAZE algorithm in software with 32 bit floating-point numbers. However, an implementation with floating-point calculations is very costly in terms of hardware resources. Therefore, the entire hardware implementation of AKAZE uses fixed-point numbers. This is easily possible because all functions have been normalized to pixel values greater or equal zero and below one. To maintain similar accuracy, all produced images have a 16 bit format, except for DoH images, which have a 32 bit format.

System Overview:

As shown in Figure 3.12, the final system can be divided into three parts: a PC platform, an ARM processor, and the hardware accelerator. Both ARM and accelerator are embedded in the Zynq-7000 chip on a ZedBoard [82]. The PC platform reads the images, sends them to the ARM via Ethernet, and then receives back the keypoints. It also computes the description and matching algorithms that use these keypoints to evaluate the system.

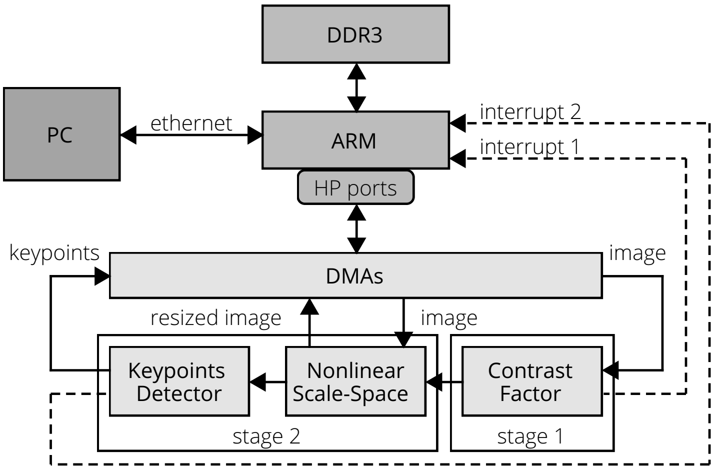


Figure 3.12: Block diagram of the complete system for the AKAZE VHDL implementation.

The accelerator consists of two pipelined stages: stage one calculates the contrast factor and stage two builds the nonlinear scale-space and detects the keypoints. The ARM sends an image (first frame) to stage one, and when it is done, it sends an interrupt to the ARM to send the same image to stage two. Stage two has two outputs that are stored in memory: the keypoints of the first octave and the scaled image for the following octave. When stage two finishes, it sends an interrupt to the ARM to send the resized image back from the memory to stage two to detect the keypoints for the second octave. This process is repeated until all keypoints of the different octaves have been sent to the ARM. While the second stage works with the first image, the first stage computes the contrast factor for the second image. Thus, both stages work simultaneously, but with different images.

Window Generator: Most modules in the implemented hardware design scan the whole image and use a specific number of pixels as a window to generate the results of an output pixel. The window generator consists of row buffers (BRAMs) and pixel buffers (shift registers), which are also the output of the window generator. Its implementation is shown in Figure 3.13 and is also based on a sliding window approach as used in the `HiFlipVX` library. Some multiplexers have been added to the pixels buffer to support horizontal and vertical replicated borders. Additionally, it supports variable image widths to be used by different octaves. In the hardware design of AKAZE, there are five different types of functions that use a window generator: Gaussian, Scharr, Conductivity, FED and DoH. The size of the window generator depends on their kernel size and the scale and ranges from 3 × 3 to 17 × 17.

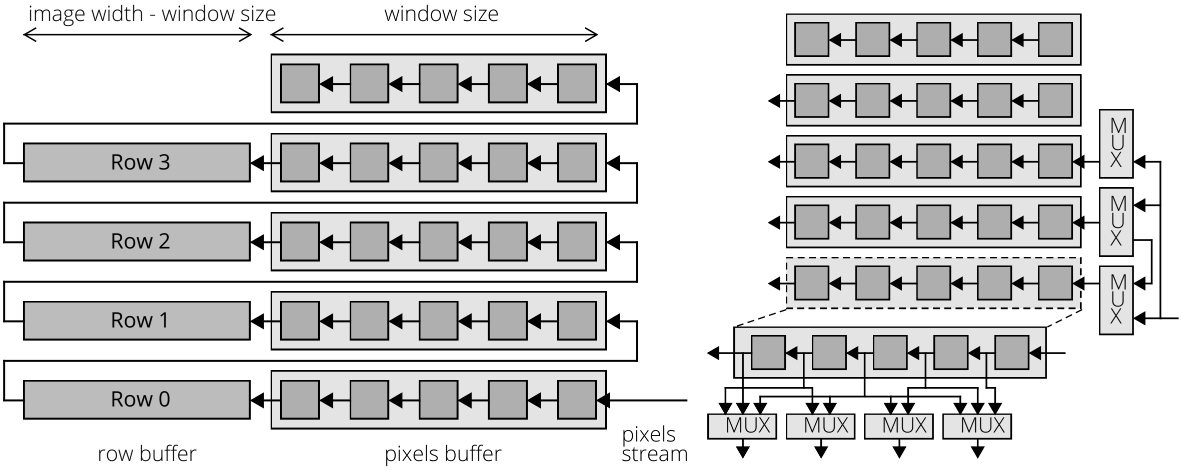


Figure 3.13: Block diagram of 5 × 5 window generator. Shows replicated border handling on the right.

Contrast Factor: The block diagram of the contrast factor calculation is shown in Figure 3.14 (left). It includes a Gaussian to smooth the image and remove noise, as well as gradient calculation including the Scharr derivatives. One adjustment made in this implementation is to approximate the gradient calculation shown in Equation (3.46). In this calculation L_{min} and L_{max} are the maximum and minimum values of the first order derivatives L_x and L_y at the same pixel location.

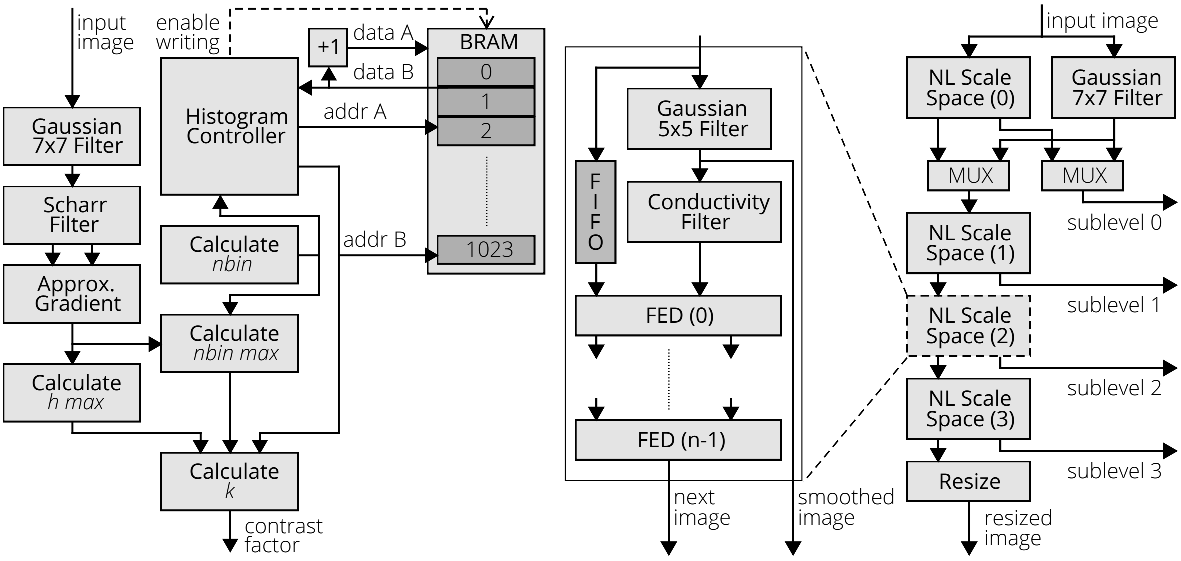


Figure 3.14: Block diagram of the AKAZE contrast factor (left) and nonlinear (NL) scale-space creation (right). FED (Fast Explicit Diffusion)

$$\nabla L \sqrt{L_x^2 + L_y^2} \approx |L_{max}| + \frac{1}{2} \frac{L_{min} \cdot L_{min}}{|L_{max}|} \tag{3.46}$$

The histogram itself is a dual ported BRAM of length 1024, with one port for reading and one for writing. Its controller is a FSM that has three states: filling histogram, reading histogram and halt. For the first state, the controller enables writing and sets both addresses A and B to the index value *nbin* so that the value at address *nbin* is incremented by one. In the second state, the controller reads the histogram values through address B starting from the first index until 68.75 % percentile of the gradient histogram is achieved. Then, the contrast factor is calculated as shown in Equation (3.47), where $i = AddrB$. This work redefined the contrast factor to be 68.75 % percentile instead of 70 %. This replaces multiplication by 0.7 to be multiplication by 0.6875, or in other words multiplication by (1/2 + 1/8 + 1/16), which can be achieved by adding and shifting instead of multiplying.

$$k = \frac{h_{max} \cdot i}{nbin_{max}} \tag{3.47}$$

Nonlinear Scale-Space: The hardware is designed to build four sublevels per octave. Each sublevel is generated by a `Nonlinear Scale-Space` block, as shown in Figure 3.14 (right). This block consists of a Gaussian filter, a conductivity function, and several FED steps. Because the pixel stream of the flow image generated by the conductivity function and the original image must be presented simultaneously, a FIFO is needed before the first FED step to compensate the delay of the window generator of the Gaussian and conductivity functions. The number of FED steps varies according to the sublevel and octave. The flow image of the conductivity function is passed through the FED functions to avoid additional buffers. This is possible because the flow image is already buffered in the FED function. In sublevel zero of octave zero, there are no FED steps. So, this sublevel is built by only a Gaussian filter with window size 7 × 7 to remove the noise of the original image. Therefore, multiplexers are used in octave zero to choose between this Gaussian filter and the normal `Nonlinear Scale-Space` step zero block.

Keypoint Detector: As shown in Figure 3.15, the DoH is applied on each sublevel with a different scale. Taking advantage of the linearity property of the convolution process, the second order derivatives can be calculated without having to calculate the first order derivatives independently. The second order derivative kernel is calculated by convolving the two derivative kernels. If more than one keypoint is present at the same time from different `Local Extrema` blocks, the multiplexer chooses the keypoint with the lower sublevel first. New keypoints are compared to the keypoints stored in the buffer by using adaptive search pointers. The keypoint is a 64 bit vector, which contains the x and y position, the sublevel number and the response value calculated by the DoH. According to measurements, the buffer length is set to a maximum of 8192 keypoints.

HLS Implementation

This part will discuss the HLS implementation of the AKAZE algorithm, which is part of `HiFlipVX`. All contained functions are included in the library and have been described in the first section of this chapter. Therefore, this part will focus more on the differences to the software and VHDL implementation. In addition, the structure of the HLS implementation, which is shown in Figure 3.16, will be described.

A very big advantage of the library is the genericity of the functions. Through the many template parameters, more variations are possible and adjustments to the algorithm are easier

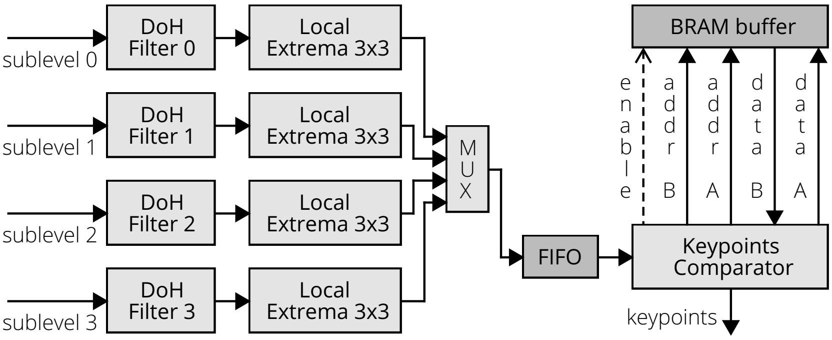


Figure 3.15: Block diagram of the AKAZE feature detection. DoH (Determinant of the Hessian)

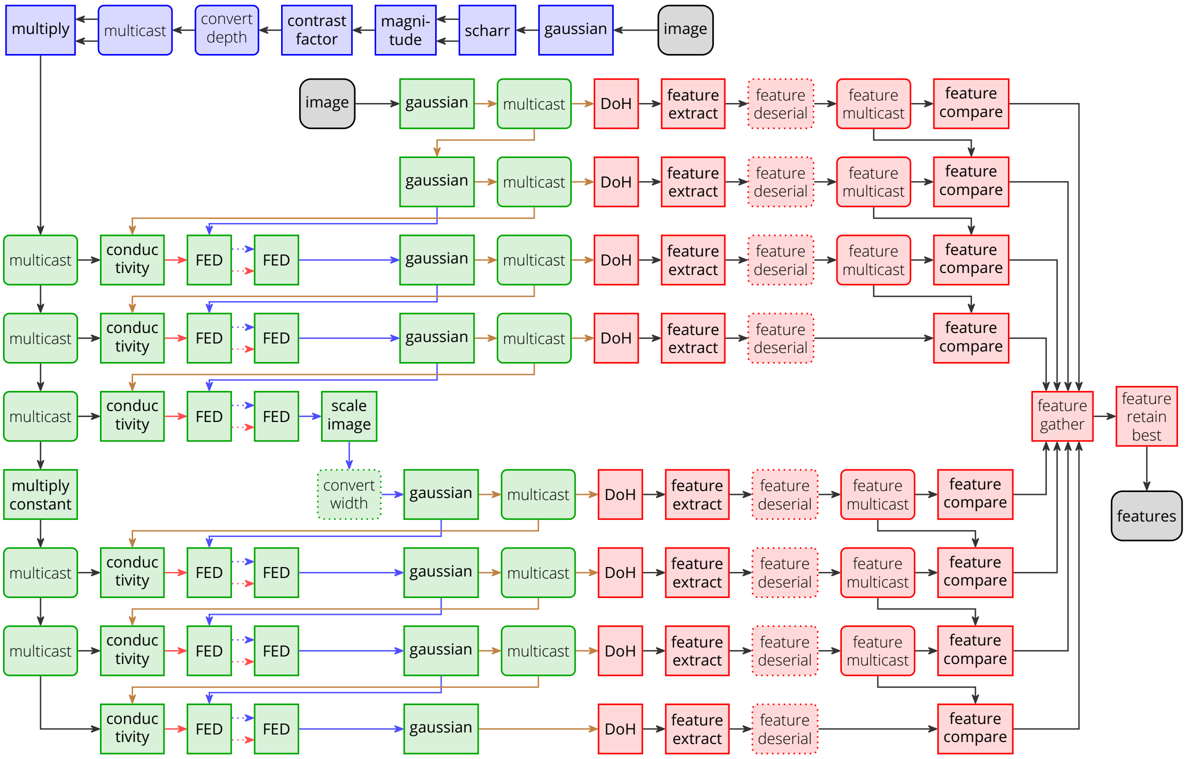


Figure 3.16: HLS implementation of the optimized AKAZE feature detector.

and faster to implement. For example, an adjustment of the data type or the vectorization to address different FPGAs. In contrast, the VHDL implementation is strongly optimized for this specific algorithm and embedded devices. Which has the advantage that many specific optimizations can be made to save resources. For example, more specific optimizations can be applied beyond functions. This becomes clear with the squaring of the contrast factor and its distribution, since one must maintain the genericity in a library function. The VHDL implementation is therefore a good reference for what is possible. What influence this has on the repeatability, performance and resource consumption will be discussed in the evaluation.

In both hardware implementations, the contrast factor, and the rest of the algorithm work on different images. However, the HLS version calculates all octaves simultaneously, which leads to an increase in resources, but also to a higher performance. For example, if the first octave has a vectorization of four and the second has none, then both can run in parallel without one becoming a bottleneck for the other. This is because in the second octave, the pixel count is quartered. Since the scale image function only halves the vectorization, an additional data width converter and a buffer are needed.

The HLS implementation of the Gaussian function forwards its input to the output, which leads to a reduction of buffers. Both hardware implementations compute the DoH in one function without a prior Scharr filter for the first order derivatives, to reduce resource utilization. In the software version, this would reduce performance due to larger kernels. In the HLS implementation, an optional SR can be performed, which is not done in the VHDL version due to resource constraints. In the VHDL version there is only one compare function, whereas in the HLS version there is one for each level. This increases resource consumption, but improves performance. In contrast, smaller buffers can be used, since the entire feature vector must be buffered in the VHDL version.

In addition, a retain best function was added in the HLS and software version to increase repeatability. Compared to the software version, multicast and gather functions are required in the HLS implementation due to the single consumer/producer approach and generic library functions. A major difference between the hardware implementations is that the HLS version is vectorizable to achieve a higher performance. Since vectorization requires more resources and the implementation results in a very deep pipeline, it is important to verify which functions contribute to the bottleneck. Normally, these are the functions that work on the image pixels, since the feature vectors should be much smaller. For example, the feature comparison and feature retain best functions operate on feature vectors. Which is also why this work uses the feature deserialization function before the feature comparison function, to prevent unnecessary resource consumption. To prevent the feature comparison function from becoming a bottleneck, its latency and BRAM usage can be limited.

3.2.6 FREAK feature description

This subsection will describe the proposed VHDL implementation of the FREAK descriptor, and the integral image needed for the descriptor. The implementation can be easily adapted to detectors other than that of the AKAZE algorithm.

Integral Image

The VHDL implementation of the `Integral Image` function differs from the software implementation shown in Section 3.2.1. However, the calculation is like the HLS implementation shown in Equation (3.17), but with some differences in its implementation. An `Integral Image` pixel at position ($I_{x,y}$) is the sum of (1) the prefix-sum of the current input image row ($\sum_{i=0}^{i=x} Img_{i,y}$) and (2) the `Integral Image` pixel of the previous row ($I(x, y-1)$). In the proposed `Integral Image` computation, four pixels can be calculated in each clock cycle. The top level function is shown in Figure 3.17 (left). It buffers the input image and the `Integral Image` stream in FIFO units.

The `Controller` block reads data from both FIFO units and forwards it with a valid signal through the prefix-sum block if the output is ready and the image FIFO is not empty. The controller sends zeros instead of the `Integral Image` to the prefix-sum when calculating the first row. It also sends a write enable signal for the integral FIFO, since the last row

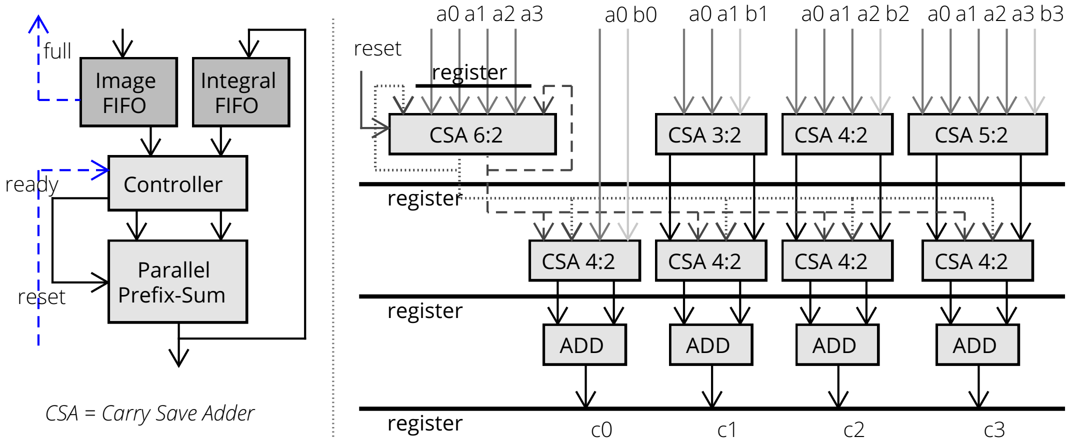


Figure 3.17: Integral Image top-level (left) and parallel prefix-sum of 4 pixel (right).

of the `Integral Image` does not need to be buffered. The `Parallel Prefix-Sum` block is pipelined and gets a reset signal for the prefix-sum at the beginning of each row. To compute four integral pixel per clock cycle, it uses CSAs (Carry-Save Adders), as shown in Figure 3.17 (right). If several variables are added together, CSA stages can be used to reduce the summation to two variables. Instead of computing the complete prefix-sum to hand it over to the computation of the next four pixels, this work uses the intermediate result of a CSA. Using a CSA without a final adder reduces the resource usage and the latency, since both would need to be computed in one clock cycle. The use of CSAs, which increases the implementation effort, is a major advantage of VHDL over HLS, where one would have to rely on the compiler for such optimizations. The latency of a 6-to-2 CSA consists of three `full adder` stages. Depending on the data bit-width of the `Integral Image`, either the adder or the CSA are the critical path.

FREAK Descriptor

Figure 3.18 gives an overview of the VHDL implementation of the FREAK descriptor. It is pipelined, implemented with fixed-point numbers, and generic enough to be combined with any other detector. It includes the different blocks of the algorithm (black), FIFO blocks (gray) and `almost full` signals needed for synchronization (blue). To adapt the design for a different detector, only some constant arrays need to be adapted. The `Send Keypoint` and `Send Descriptor` blocks write the results back to memory. They can be omitted if the descriptor is integrated into a larger design. The number of image columns and rows as well as the base addresses for integral, keypoint and descriptor are set as configuration. The different blocks are explained below.

Boundary: The FREAK reads pixels around a keypoint using a retina pattern, which has a fixed scale, but can be rotated. The `Boundary` block, shown in Figure 3.19 (top), checks whether the pattern accesses pixels that are not within the image boundaries. First, the block buffers incoming keypoints in a FIFO and sends back the `full` signal. At this stage, a keypoint consists of its coordinates (x_k, y_k) and the corresponding scale-space level (id_k). The next stage reads from the FIFO, if it is not empty and the `Pattern (unrotated)` block is not full. If the `last` signal was set and the last keypoint is read, a `last` signal is sent with this keypoint. In the next stage, the id_k of the keypoint is used to determine the maximum radius

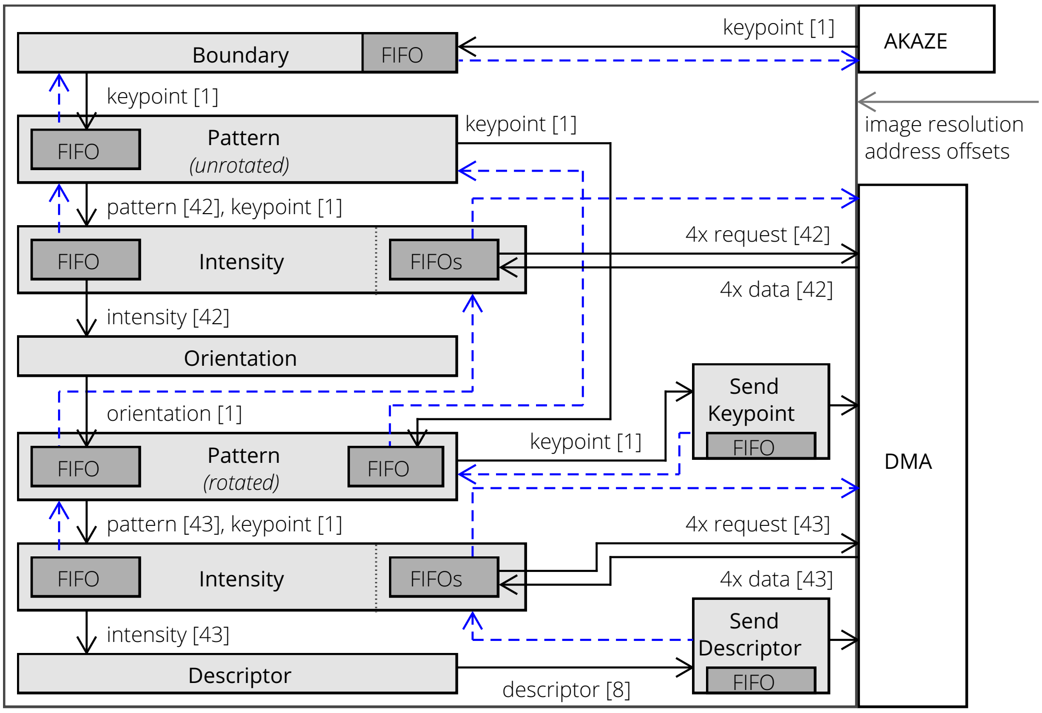


Figure 3.18: Top-level hardware design of the FREAK. Number of data send for each keypoint (square brackets), FIFO flags (dashed lines).

of the pattern, which is stored in a LUT. The last stage checks if the pattern of a keypoint is within the image boundaries. If this is the case, it is sent as a valid keypoint to the `Pattern (unrotated)` block. It also counts the amount of valid keypoints and sends it to the `Send Keypoint` and `Send Descriptor` blocks.

Pattern (unrotated): A pattern consists of 42 coordinates (x_f,y_f) around a keypoint (x_k,y_k) including their corresponding scale (s_f). The original algorithm precomputes all different possible patterns during initialization and then reads the needed pattern from memory. They compute a pattern for each possible scale (σ) and orientation (ω) combination. The `Pattern (unrotated)` block, shown in Figure 3.19 (middle), stores the incoming keypoints in a FIFO. If the FIFO of the first `Intensity` block and the keypoint FIFO of the `Pattern (rotated)` block are not full, the keypoint information can be read. In addition, a pointer is created for each of the 42 pattern points. The last stage gets the pattern points from a BRAM, using this pointer and the keypoint scale-space level (id_k). This work stores all precomputed patterns in a BRAM. These are 42 coordinate-scale pairs for each of the twelve AKAZE scales (σ), because the implementation only needs patterns without orientation in this block.

Intensity: The implemented `Intensity` block is shown in Figure 3.19 (bottom). As shown in Equation (3.48), it computes an intensity value (i) using the pattern and keypoint information. First, it calculates the four corner coordinates for each pattern (x_l, x_r, y_t, y_b). With this information, it computes the divisor of i and the address offsets needed to read the `Integral Image` pixels (I). The divisor is buffered in a FIFO. The next stage gets the integral pixels, computes the dividend (24 bit) and reads the divisor (16 bit) of the FIFO. The last stage is a radix-four division unit, which outputs the 42 intensity values of each keypoint.

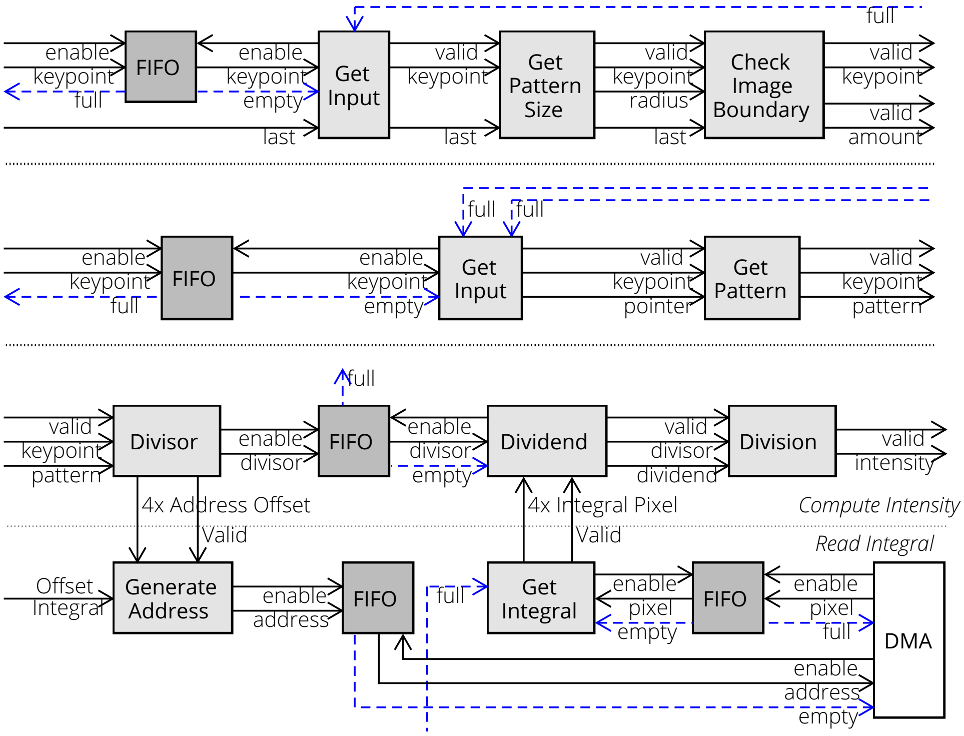


Figure 3.19: Hardware design of the boundary block (top), the unrotated Pattern block (middle) and the Intensity block including a DMA interface (bottom).

$$
\begin{aligned}
x_l &= \lfloor x_f + x_k - s_f + 0.5 \rfloor, y_t = \lfloor y_f + y_k - s_f + 0.5 \rfloor \\
x_r &= \lfloor x_f + x_k + s_f + 1.5 \rfloor, y_b = \lfloor y_f + y_k + s_f + 1.5 \rfloor \\
i &= \frac{I_{y_b,x_r} - I_{y_b,x_l} + I_{y_t,x_l} - I_{y_t,x_r}}{(x_r - x_l) \cdot (y_b - y_t)}
\end{aligned}
\tag{3.48}
$$

In the lower part of the figure, a DMA interface is created to read from memory. First, the integral addresses are created and stored in a FIFO. The full flag does not need to be checked, because the divisor FIFO is already validated. A DMA reads from this FIFO and writes the integral values back to a FIFO if it is not full. The `Get Integral` stage checks if the `Pattern (rotated)` block is ready and sends the integral pixels to the next unit. This thesis decided to read the `Integral Image` from DDR memory due to the maximum pattern size of the proposed configuration. For smaller patterns it could also be stored in BRAM. To adapt the design, the integral offset is not needed, and a BRAM controller must be implemented to provide access to the two intensity blocks.

Orientation: The `Orientation` block computes the orientation (θ_Ω) using fixed orientation pairs (Ω), dependent on the orientation amount (ω), shown in Equation (3.49) and Figure 3.20 (top). The pairs (Ω) are fixed values of the FREAK algorithm, which are not changed by the configuration. This work changed the division from the original algorithm to be a multiple of two, to use a shift operation, which did not have a noticeable effect to the results. The first stage of the hardware design gathers the 42 intensity values in the *Demux* unit. The reason for the *Demux* unit is that the different orientation pairs need different intensity values, and it needs to buffer the values before the next data-beat of intensity values arrives. Then the 45 δ

values are computed in parallel, based on the hardwired pairs (Ω_i, Ω_j). The next unit outputs three of 45 orientation weights (Ω_x, Ω_y) in each clock cycle, since the `Weight` unit needs to be faster than the previous one. The results are first multiplied, then shifted and then summed together, doing three operations for each weight in parallel. This work adapted the shift unit to calculate the same results as a division unit and round to the nearest value. The last unit computes the inverse trigonometric tangent function (*atan*) using the mathematical function $arctan2(d_x, d_y)$. This work implemented a non-pipelined CORDIC algorithm, since it has minimum 42 clock cycles for one data-beat of intensity values. It has 15 bit inputs and an 8 bit output, since there are only 256 different orientations (ω) in the proposed configuration. To achieve the required latency, fifteen repetitions are performed with two clock cycles each.

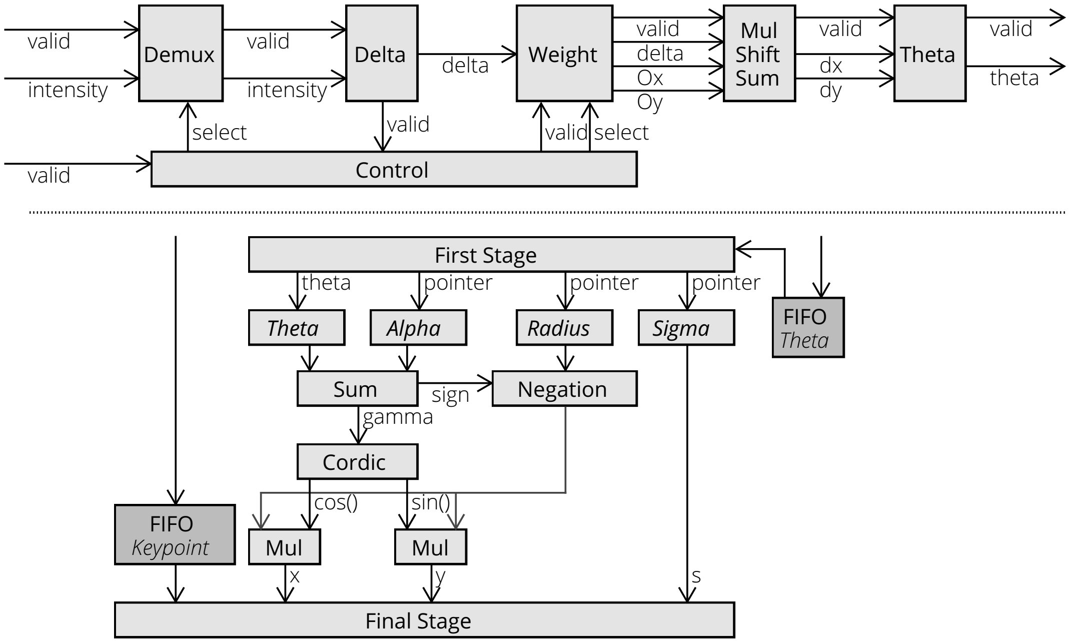


Figure 3.20: Design of the Orientation (top) and rotated Pattern block (bottom).

$$
\begin{aligned}
\delta(n) &= i(\Omega_i(n)) - i(\Omega_j(n)) \\
\alpha_\Omega &= \omega \cdot atan\Big(\sum_{n=0}^{44} \frac{\delta(n)\cdot\Omega_x(n)}{128}, \sum_{n=0}^{44} \frac{\delta(n)\cdot\Omega_y(n)}{128}\Big) + \frac{1}{2} \\
\theta &= [(\alpha_\Omega < \Omega) \rightarrow \lfloor \alpha_\Omega + \omega \rfloor] \land [(\alpha_\Omega \geq \Omega) \rightarrow \lfloor \alpha_\Omega \rfloor]
\end{aligned}
\tag{3.49}
$$

Pattern (rotated): The `Pattern (rotated)` block has far more possible patterns than the the `Pattern (unrotated)` block, due to the 256 orientations (ω), 43 pattern points and twelve AKAZE scales (σ). For this configuration there would be 132 096 pattern points, which would consume 258 BRAMs on a Zynq-7000 series FPGA. Therefore, this work created a mixed design that computes the patterns using partial results, which are stored in BRAM. Figure 3.20 (bottom) shows the pipelined design of the pattern generator. First, it creates pointers for the BRAMs and reads the θ value from a FIFO. The `Theta` BRAM splits a 360° circle of degrees into ω (256) entries. The `Alpha` BRAM contains the 43 pattern angles. The `Radius` (σ_f) and the `Sigma` (s_f) BRAMs contain 516 entries each. Their computation is shown in Equation (3.50). The patternscale constant is part of the proposed configuration. The range of `Gamma` ($\gamma = \alpha + \theta$) is reduced from $[0..4\pi]$ to $[\frac{-\pi}{2}..\frac{\pi}{2}]$ in the *Sum* unit. It creates a negative sign bit, if `Gamma` was in the range of $[\frac{1\pi}{2}..\frac{3\pi}{2}]$ or $[\frac{5\pi}{2}..\frac{7\pi}{2}]$. This work has implemented a pipelined CORDIC algorithm, which computes the cosine (18 bit) and sine (18 bit) of γ (20 bit). To compute the pattern coordinates (x_f, y_f), the CORDIC results need to be multiplied with the radius, which is negated in advance dependent on the signed bit. The pattern scale (s_f) fits completely into the BRAM. The final stage sends the pattern with its keypoint to the next block.

$$\begin{aligned} \sigma_f(n) &= radius(n) \cdot factor(n) \cdot patternscale \\ s_f &= sigma(n) \cdot factor(n) \cdot patternscale \end{aligned} \tag{3.50}$$

Descriptor: The descriptor does 512 different comparisons between the intensity values. Each comparison creates one bit of the descriptor vector. First, all intensity values are gathered, since the comparisons are crisscrossing. Then, 64 comparisons are computed per clock cycle, due to the size of the proposed DMA interface.

Send Keypoint & Send Descriptor: Both blocks are similar and buffer their output in FIFO units. The `Send Keypoint` block converts the ID to the scale size of AKAZE. The keypoint coordinates and scale are packed into a 64 bit wide vector. Both blocks contain an address generator that takes an address offset and increments the address for each incoming data. Since these blocks create data bursts of eight 64 bit vectors, one address is created for either eight keypoints or one descriptor. Before getting the amount of keypoints from the `Boundary` block, the maximum allowed amount of keypoints is set to a predefined value (e.g., 8190). A `last` signal indicates if the last keypoint or descriptor has been read. This signal, a data FIFO and an address FIFO are the interface to the DMA.

Memory Controller: This work implemented a memory controller that includes the DMA blocks from XILINX. To interface between the FIFO blocks and the DMA blocks, AXI4-Stream wrappers have been implemented for the design. Additionally, a block that reads the input keypoints for the descriptor from memory has been created. This block and the `Send Keypoint` and `Send Descriptor` blocks have been connected to DMA blocks and the `Integral Image` to a video DMA block. These DMA blocks are controlled by the ARM processor by an AXI4-Lite interface. The control parameters can be the base address of a memory block and its size. The design gets more complicated, when randomly accessing the `Integral Image`, to compute the intensity values. Here, the function writes the control registers of the DMA blocks from the hardware memory controller. Each one of the two intensity blocks reads four integral pixels from four DMA blocks in parallel. Each group of DMA blocks is connected to one high-performance port through an AXI-interconnect block.

3.2.7 MobileNets

MobileNets [21] was presented by Google and has been developed for mobile and embedded vision applications. It is also part of the `HiFlipVX` library. The MobileNets architecture is based on depthwise separable convolution. This is because a standard convolution can be factorized into a depth convolution and a point convolution. The factorization of the convolution layer leads to a reduction in model size and computational requirements of the algorithm.

The first layer of MobileNets is a full convolution layer. Later layers are a combination of depthwise convolution and pointwise convolutions. All convolutions are followed by a batch normalization layer and activation layer (ReLU). The final fully connected layer has no non-linearity and is followed by a Softmax layer. Before the final fully connected layer an average pooling is used to reduce the spatial resolution. In total MobileNets has 28 layers.

Figure 3.21 shows the hardware implementation of the different parameterizable layers of MobileNets. The different modules are directly connected to each other, resulting in a very deep pipeline. Thereby module one contains the first layer and module fifteen contains the last layer. The input of the first layer in module one is connected to a data-width converter and gets its data from the global memory. To optimize memory bandwidth, it receives an, for example, 64 bit wide input and converts it to the desired vector size (V_{IFM}) for 3D-convolution. The output vector size (V_{OFM}) is then converted to the vectorization (V_{PW}) for the batch normalization and activation layers. All layers and conversion units of the pipeline are connected via very small FIFO buffers. They are marked by a thicker line in the figure.

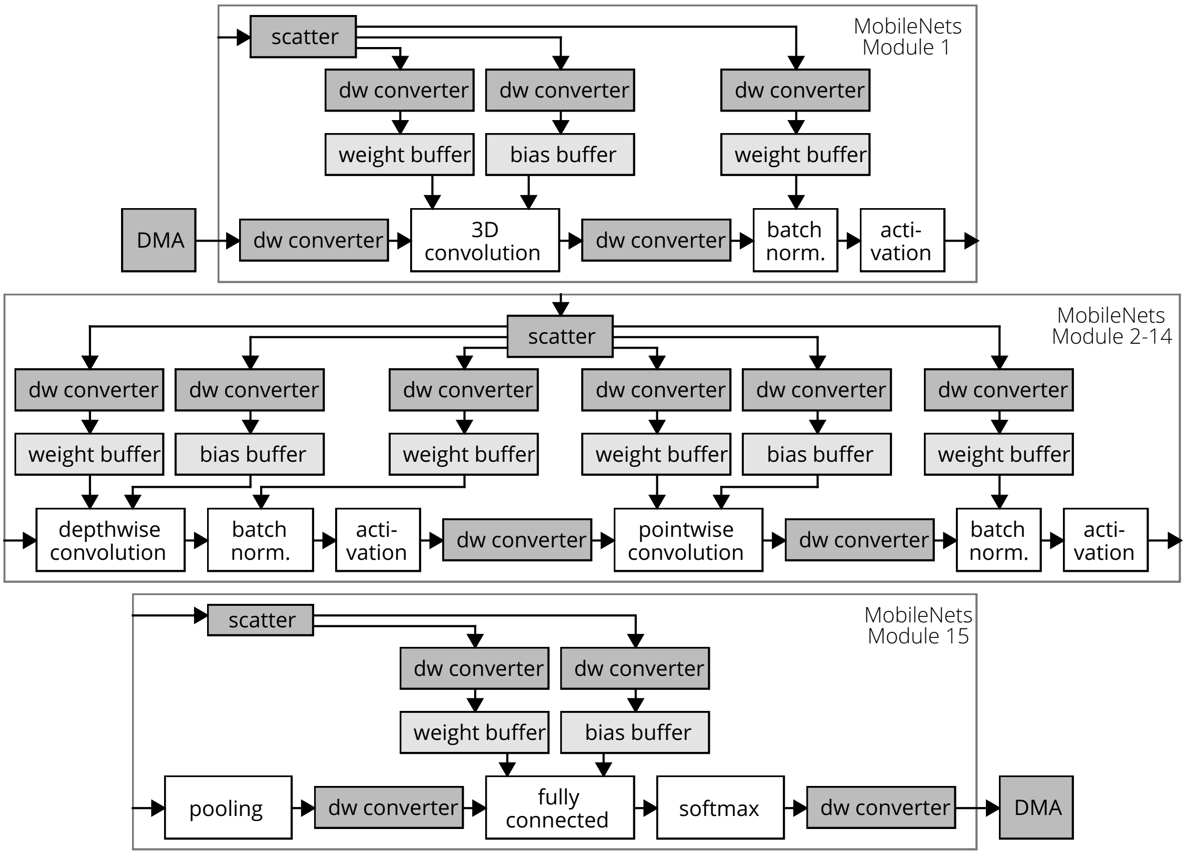


Figure 3.21: Block design of the MobileNets hardware implementation. MobileNets has been separated into 15 modules. The modules are directly connected to each other in the order of their numbering. Local buffers are marked in light gray. Data movers blocks are marked in dark gray. Multiple scatter units can be connected to the same DMA. dw = data-width.

Modules two to fourteen all have the same structure, but with different parameter settings.

Due to the 3D-convolution, the pointwise convolution requires the highest parallelization and thus the two data-width converters to ensure that none of the module's functions becomes a bottleneck. The first three layers, which include a depthwise convolution, batch normalization and activation layer, all have the same degree of parallelization (V_{DW}). The pointwise convolution gets a vector size of V_{IFM} and outputs a vector size of V_{OFM}. The last two layers have a parallelization degree of V_{PW}. With these four vectorization parameters ($V_{DW}, V_{IFM}, V_{OFM}, V_{PW}$) the optimal configuration for the available number of resources can be found, as it will be shown in the evaluation. Data-width converters could also be used to connect the different modules with each other. However, they were not needed for the final configuration.

Module fifteen contains the last layer and its output is therefore connected to, for example, a 64 bit wide DMA via data-width converter. This module only needs the parameter V_{DW} for pooling and the parameter V_{IFM} for the input of the fully connected layer. The Softmax layer is not computationally intensive enough to become a bottleneck. In general, all vector parameters must be set so that no single layer becomes a bottleneck, since the slowest layer limits the speed of the others. The different modules contain scatter engines to distribute all coefficients to the local buffers. This allows all coefficients to be preloaded with an optimized utilization of the memory bandwidth. The scatter engine also reduces the number of DMA blocks needed to access memory to load new coefficients and therefore saves resources. They require data-width converters, since each local buffer has a different bit-width, which depends on the degree of parallelization of the corresponding layer. The `HiFlipVX` data-width converter can also convert between widths that are not multiples of each other. Therefore, the alignment of the data for the different local buffers must be adapted to the data type of the scatter engine in global memory.

3.3 Evaluation

One aim of this thesis is to improve feature extraction algorithms and to enable the efficient implementation of object detection algorithms using `HiFlipVX`. First this section evaluates the individual image processing and feature extraction functions of the library, by their resource consumption, latency, and scalability, and compares them to a related work [12, 13]. Then it evaluates the proposed software, hardware, and library implementations of the ORB, AKAZE and FREAK algorithms by their performance, repeatability, and resource utilization [25, 26, 22, 23]. It compares each of them to the state-of-the-art hardware implementations. Finally, it evaluates the neural network extension of the library based on its accuracy, performance, scalability, and resource efficiency [14]. It shows the efficient implementation of the MobileNets algorithm and compares it with the related work. This work uses several FPGA-based development boards to evaluate the library and the implemented object detection algorithms. Table 3.9 shows these boards, which include two Ultrascale+ MPSoC and one Zynq-7000 SoC.

3.3.1 Library Functions

This subsection evaluates the individual functions of the library based on their resource consumption, latency, and scalability. It compares the library to a software implementation in

Table 3.9: FPGA development boards and their available resources.

Development board	FF	LUT	BRAM	URAM	DSP
PYNQ-Z1 Zynq-7000 SoC (Artix 7Z020)	106 400	53 200	140	0	220
ZedBoard Zynq-7000 SoC (Artix 7Z020)	106 400	53 200	140	0	220
Ultrascale+ MPSoC ZCU102 (Kintex ZU7EV)	548 160	274 080	912	0	2520
Ultrascale+ MPSoC ZCU104 (Kintex ZU9EG)	460 800	230 400	312	96	1728

terms of accuracy due to hardware-specific optimizations. It uses the ZCU104 for evaluation, Vivado HLS 2020.1 for IP-core generation, and Vivado 2021.1 for the final design. After version 2020.1, XILINX replaced Vivado HLS by Vitis HLS. However, newer Vivado tools can still use older HLS IP cores. When using floating-point operations, XILINX uses specific IP cores, which unfortunately can lead to errors when combining different tool versions. The evaluation uses the `DECISION` framework because it can create multiple IP cores in parallel and has an easy-to-use frontend. The framework enables all available HLS optimization strategies to create the IP cores. To create the Vivado design, it uses the default optimization settings. It automatically creates a design, connects all IP cores, includes DMA blocks for memory access and generates the bitstream. It configures an AXI4-stream interface for all IP cores and enables the last bit to connect to the DMA blocks.

Table 3.10 shows the default configuration of most parameters in the `HiFlipVX` library. It serves as a basis for evaluating the various parameter settings. Input and output images use unsigned values whenever possible. For example, the output images of the DoH, Scharr and Sobel functions can only have signed values. The pixel values of the image output are truncated in case of an overflow and rounded to zero in case of an underflow, if adjustable. The framework implements the IP cores with a target frequency of 100 MHz and an uncertainty of 20 % in Vivado HLS to ensure that Vivado can generate the bitstream. Some functions, such as histogram, table lookup, or min-max-location, can specify the size, offset, and covered area of their internal arrays.

Table 3.10: Default configuration of the `HiFlipVX` library functions.

Output type	8 bit	Vector size (V_S)	1
Input type	8 bit	Kernel size (K_S)	3
Resolution	1920 × 1080	Kernel scale (K_σ)	1
Underflow	to zero	Step size	0
Overflow	truncate	Border type	constant
Frequency	100 MHz	Separable	true
Uncertainty	20 %	Factor (α)	0.25
Array bins	256	Quantization	3
Array range	256	Shift	0
Array offset	0	Upper threshold	100
Precision	low	Lower threshold	25

3.3.1.1 Image Pixelwise Functions

Table 3.11 shows on the one hand the actual resource consumption (C) and on the other hand the estimated resource consumption and latency from synthesis (A & B). The table shows that the synthesis estimates of Vivado HLS sometimes deviate quite strongly from the actual results. In contrast, the results between the synthesized and the implemented design in Vivado differ only by their LUT amount. Here, the synthesis estimates a total of 11 % more LUTs than needed in the final system.

Table 3.11: Vivado HLS synthesis (A), Vivado synthesis (B), Vivado implementation (C) results of the pixelwise functions.

	FF			LUT			DSP			Latency
	A	B	C	A	B	C	A	B	C	A
Data object copy	38	80	80	148	61	53	0	0	0	2 073 603
Bitwise not	38	80	80	156	61	52	0	0	0	2 073 603
Bitwise and	38	98	98	165	78	68	0	0	0	2 073 603
Bitwise or	38	98	98	165	78	68	0	0	0	2 073 603
Bitwise xor	38	98	98	165	78	68	0	0	0	2 073 603
Threshold (binary)	30	40	40	148	50	45	0	0	0	2 073 603
Threshold (range)	31	59	59	161	64	56	0	0	0	2 073 603
Max	38	98	98	176	78	69	0	0	0	2 073 603
Min	38	98	98	176	78	69	0	0	0	2 073 603
Absolute difference	38	98	98	209	89	79	0	0	0	2 073 603
Arithmetic addition	38	98	98	172	78	69	0	0	0	2 073 603
Arithmetic subtraction	38	98	98	172	78	69	0	0	0	2 073 603
Constant mult.	36	70	70	148	58	49	0	0	0	2 073 603
Pixelw. mult.	38	98	98	197	119	107	0	0	0	2 073 603
Pixelw. mult. (saturate)	38	98	98	218	136	126	0	0	0	2 073 603
Pixelw. mult. (round nearest)	38	74	74	157	70	61	1	1	1	2 073 603
Pixelw. mult. ($\alpha = \frac{1}{5}$)	38	90	90	197	142	131	1	1	1	2 073 603
Weighted average	38	98	98	205	81	74	0	0	0	2 073 603
Magnitude	222	152	152	849	317	304	0	0	0	2 073 604
Phase (quantization = 3)	256	45	45	1497	59	53	0	0	0	2 073 604

The Phase function needs to compute a pipelined `atan2` function, which is quite resource intensive. However, the estimate differs greatly from the actual design. Due to a quantization of three, the output pixels can only take eight different values. Therefore, a possible explanation for the large deviation is that the optimization phase of the tool has recognized this and stores the possible results of the `atan2` function in a LUT.

Since the copy function only forwards the data, it determines how many resources the interfaces and basic structure of the pixelwise functions consume. Most pixelwise functions are quite simple and only need a few extra resources for their arithmetic or binary operation. It is striking that some even need less resources than the copy function. One reason for this may be that they do not need all bits of their input(s) or output. For example, the threshold function outputs only Boolean values. Also, some of the functions (copy, not, threshold and constant multiply) have only one input. The use of DSPs can reduce the need for both LUTs and FFs. This becomes visible in the multiplication functions, which either use DSPs or not.

There are two options for the threshold function. On the one hand only a lower limit and on the other hand a lower and an upper limit. Using the pixelwise multiplication function, the table shows the alternative policies for overflow (saturate) and underflow (round to nearest). The resource consumption increases slightly, although this is not always visible, due to the DSP used in the function, which rounds to the nearest value. Both pixelwise multiplication and constant multiplication multiply their input by a constant value. Changing this constant from 0.25 to 0.2 shows the compile time optimizations of the library, since it automatically replaces multiplications which are a multiple of two by a shift operation. The magnitude operation has the highest computational complexity with two multiplications, one addition and one square root. Nevertheless, the resource consumption is relatively low, because the library implements an optimized square root for integer values.

Figure 3.22 selects the pixelwise multiplication and magnitude functions because of their higher resource consumption to evaluate the scalability of some parameters. Therefore, it uses the relative consumption of LUTs and FFs compared to the default configuration. The smaller increase in resource consumption for the multiplication function compared to the magnitude function can have several reasons. On the one hand, because the latter requires fewer resources for its operation compared to the rest of its function. On the other hand, because it uses DSPs for data types with a higher bit-width. Whether the tool uses DSPs or LUTs for an arithmetic operation depends strongly on the selected data type. The pixelwise multiplication function requires one DSP for unsigned 16 bit and three for unsigned 32 bit data types. The magnitude function requires one DSP for signed 8 bit and two for (un)signed 16 bit data types. All other configurations do not consume DSPs.

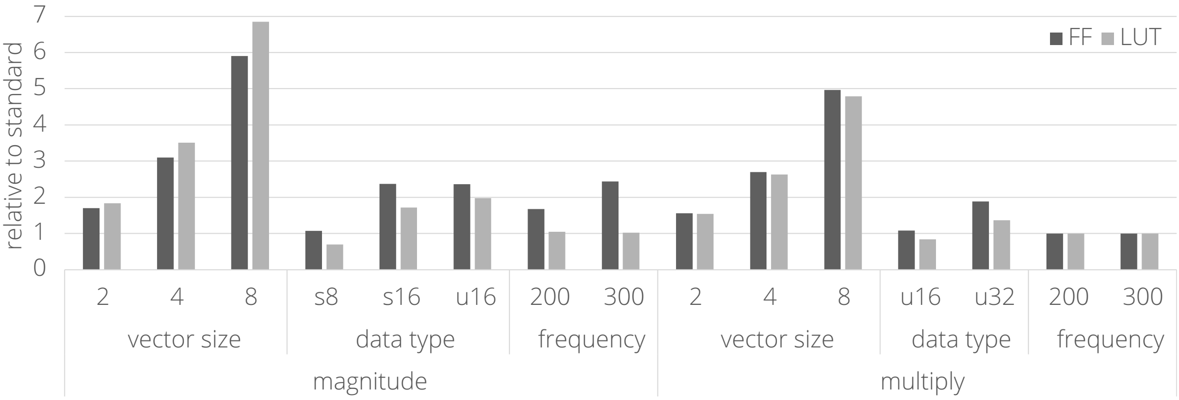


Figure 3.22: Compares relative LUT and FF utilization of pixelwise functions to default configuration. Frequency is in MHz. Data type is unsigned (u) or signed (s).

3.3.1.2 Image Filter Functions

Table 3.12 shows on the one hand the actual resource consumption (C) and on the other hand the estimated resource consumption and latency from synthesis (A & B). Another reason for comparing estimates and actual results is that the `DECISION` framework needs these estimates for its middleend. There is no difference in DSP and BRAM consumption between the synthesized and implemented designs in Table 3.12. This is advantageous because BRAM is usually a limiting factor for `HiFlipVX`-based implementations, along with LUTs. However, Vivado HLS overestimates the consumption by 36.4 % for FFs and 184.7 % for LUTs. The models of this work could integrate these variations. However, the data does not show any regularities. For example, for a selected frequency of 300 MHz, the overestimation was at 144.9 % for FFs and 154.2 % for LUTs. It would be more accurate to use the Vivado synthesis estimates since they get the same values for FF utilization and differ by only 2.6 % for LUTs. For a frequency of 300 MHz, the implemented design required only 43.2 % more FFs and 5.1 % more LUTs for the filter functions compared to the 100 MHz design.

Table 3.12: Vivado HLS synthesis (A), Vivado synthesis (B), Vivado implementation (C) results of the filter functions. FED (Fast Explicit Diffusion), DoH (Determinant of the Hessian) and NMS (Non-Maximum Suppression).

	FF		LUT			BRAM	DSP	Latency
	A	B,C	A	B	C	A,B,C	A,B,C	A
Box	246	147	536	149	143	1	2	2 076 605
Conductivity	2542	1462	2660	1438	1429	1	0	2 076 639
Convolve	280	197	633	170	162	1	1	2 076 605
Dilate	246	162	580	172	164	1	0	2 076 605
DoH$_1$	455	302	1125	372	359	2	2	2 076 605
DoH$_2$	595	465	1682	671	652	2	5	2 079 609
Erode	246	162	580	172	163	1	0	2 076 605
FED	119	168	433	128	121	2	0	2 076 604
FED (forwarding)	127	200	475	162	153	2	0	2 076 604
Gaussian	246	162	624	138	130	1	0	2 076 605
Gaussian (forwarding)	280	221	684	222	212	1	0	2 076 605
Hysteresis	272	163	597	110	106	1	0	2 076 605
Median	305	224	1074	421	417	1	0	2 076 606
NMS	280	197	635	168	163	1	0	2 076 605
NMS (oriented)	201	249	892	260	253	2	0	2 076 604
Scharr	280	221	824	273	260	1	0	2 076 605
Segment test detector	1083	1005	4905	1843	1813	3	0	2 082 616
Sobel	280	221	760	229	219	1	0	2 076 605

Since all filters use line buffers, they require BRAM. On the one hand, the number depends on the number of inputs, which are two for DoH$_1$, FED and NMS (oriented). On the other hand, they depend on the size of the kernel, which is 5 × 5 for the DoH_2, and 7 × 7 for the segment test detector. In addition, the BRAM size can increase with larger data types and vector sizes.

There are two variants of the DoH. The second variant creates a kernel that performs both convolutions in one, by convolving the Scharr kernel with itself, as shown in Equation (3.15). The first variant only calculates the second order derivatives, which means that it needs an additional Scharr filter in advance. Adding the resources of DoH_1 and Scharr, they need 33 fewer LUTs, 55 more FFs, 3 fewer DSPs, and 1 more BRAM than for DoH_2. However, this variant would also require two additional FIFO blocks between the two functions, so that it needs more LUTs and FFs. The disadvantage of the higher BRAM utilization increases when scaling the kernel (K_σ) like in AKAZE. In this case the DoH_1 and Scharr would need a total of ($6 \cdot K_\sigma$) line buffers and the DoH_2 only ($4 \cdot K_\sigma$). Applying Equation (3.25) to the two functions of the first variant to calculate the latency, it requires fewer clock cycles due to the smaller sliding window.

The DSP consumption of the user-defined convolution is not ten as expected, that is, one for each kernel coefficient and one for normalization. This is because the compiler decides whether to use LUTs or DSPs for multiplications. Defining the use of LUTs or DSPs has shown that in most cases it is advisable to let the compiler decide. Thus, it can make decisions based on the total utilized and available resource consumption. The box filter and user-defined convolution have the same kernel parameters set to show the difference compared to a hand-optimized filter.

As the box filter is separable, it requires a DSP for each normalization, one for the horizontal filter and one for the vertical filter. In return for the extra DSP, it needs fewer LUTs and FFs, especially for larger kernels. The segment test detector consumes the most resources, not only because of the larger kernel, but also because it does many comparisons. This is because nine consecutive pixels on a circle of sixteen pixels must meet the criterion, which opens many possibilities. Although the implementation in the table uses a low precision for the conductivity function, the measurements of the AKAZE algorithm showed that this can be sufficient. It calculates a Scharr filter, but also two multiplications, two additions, and one division, resulting in an increased resource consumption. Another special feature of this function, compared to all other filter functions, is that it reads a single scalar value at runtime in addition to the input image.

Figure 3.23 evaluates the scalability of the filter functions. It uses relative values for LUTs and FFs for better readability, and absolute values for DSPs and BRAM to avoid division by zero. Besides the box filter, the Gaussian filter is the only other separable filter in `HiFlipVX`. The second Gaussian function of Table 3.12 forwards the input to the second output. Because of this forwarding it cannot modify its buffered input and therefore cannot use a separable filter. Separable Sobel and Scharr filters would be possible but would require twice as many BRAM when calculating the derivatives in the x and y direction. The Sobel filter therefore calculates two kernels and accordingly has two outputs. On the other hand, the FED function has two inputs, but only considers five pixels of the input window.

The BRAM size scales linearly with the number of line buffers, which depend on the kernel size. When using vectorization, the BRAM consumption shows the advantage of the configurable dimensions of the 18k BRAM. These are 2048 elements for 9 bit wide data, 1024 elements for 18 bit and 512 elements for 36 bit. The figure shows the resource consumption of 36k BRAM, which consists of two 18k blocks. Only for a vector size of eight, the line buffers consume

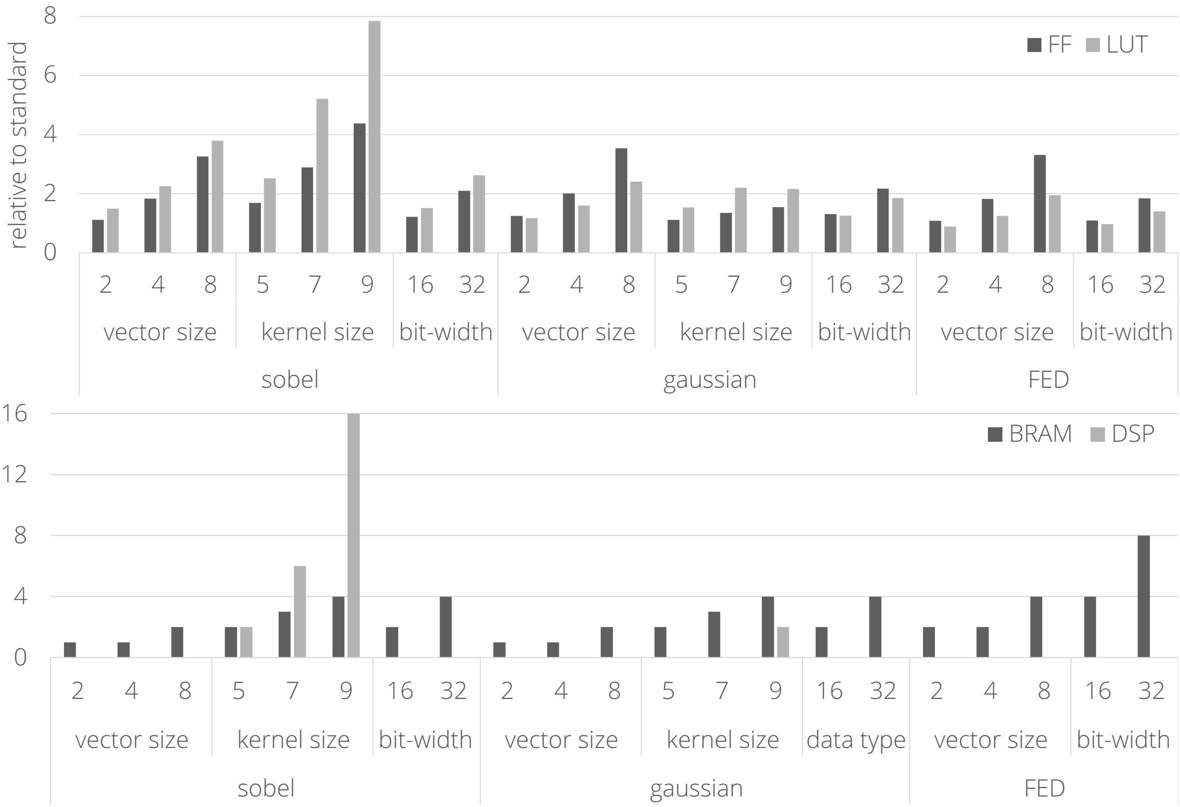


Figure 3.23: Resource utilization scalability results of filter functions.

twice the amount of BRAM because the bandwidth is not sufficient. The FED filter requires twice as many BRAM because it buffers two inputs. Increasing the data type also increases the overall utilization of the BRAM, since the 1920 image columns consume almost all 2048 entries in the 9 bit configuration. The library packs these vectors that operate in SIMD mode using an HLS directive to optimally use the BRAM bandwidth. Increasing the vector size reduces the total number of clock cycles, which also provides the opportunity to lower the frequency and perform DVFS (Dynamic Voltage and Frequency Scaling), thus saving energy [258].

As shown in the figure, the functions scale approximately linearly with an increasing vector size, kernel size or data type. However, the factor is smaller than the respective parameter. On the one hand, the vectorized operations do not make up all resources. On the other hand, the library implementations make use of the kernel coefficient patterns. For example, this reduces the multiplications for the coefficients of a 9 × 9 Sobel kernel, without considering the zero row/column in its center, from 144 to 40. A Sobel filter with a kernel size of nine only consumes 4.4 times more FFs and 7.8 times more LUTs, even though the sliding window is nine times larger and contains twelve times more coefficients. For separable kernels, as used in the Gaussian implementation, the scalability is even better. The coefficient values are higher for bigger Gaussian or Sobel kernels. Therefore, some of the multiplications use DSPs instead of LUTs for these kernels, resulting in a reduction in LUT utilization.

Even though the median filter is quite compute-intensive, it requires relatively few resources due to the optimized variant of the 3 × 3 kernel. For larger kernels, the library uses a sorting network, which involves many parallel operations and a complexity of $\mathcal{O}(n \cdot \log(n))$. Therefore, it needs 11.3 times more FFs and 25.9 times more LUTs for a 9 × 9 kernel that has 9 times more elements than the 3 × 3 kernel.

3.3.1.3 Image Conversion Functions

Figure 3.24 shows the LUT and FF consumption of the implemented conversion functions for different configurations. The channel combine and channel extract functions require most resources when using RGB images, because they need to buffer data. Due to the C++ standard, it is not possible to define 24 bit data types for the interfaces. Therefore, both functions store their data nested into 32 bit data types (RGBR|GBRG|BRGB). The same applies to the color convert function. When converting from RGB or RGBX to gray, there is also a conversion to grayscale values. Therefore, it needs three DSPs for RGB to gray and two DSPs for RGBX to gray.

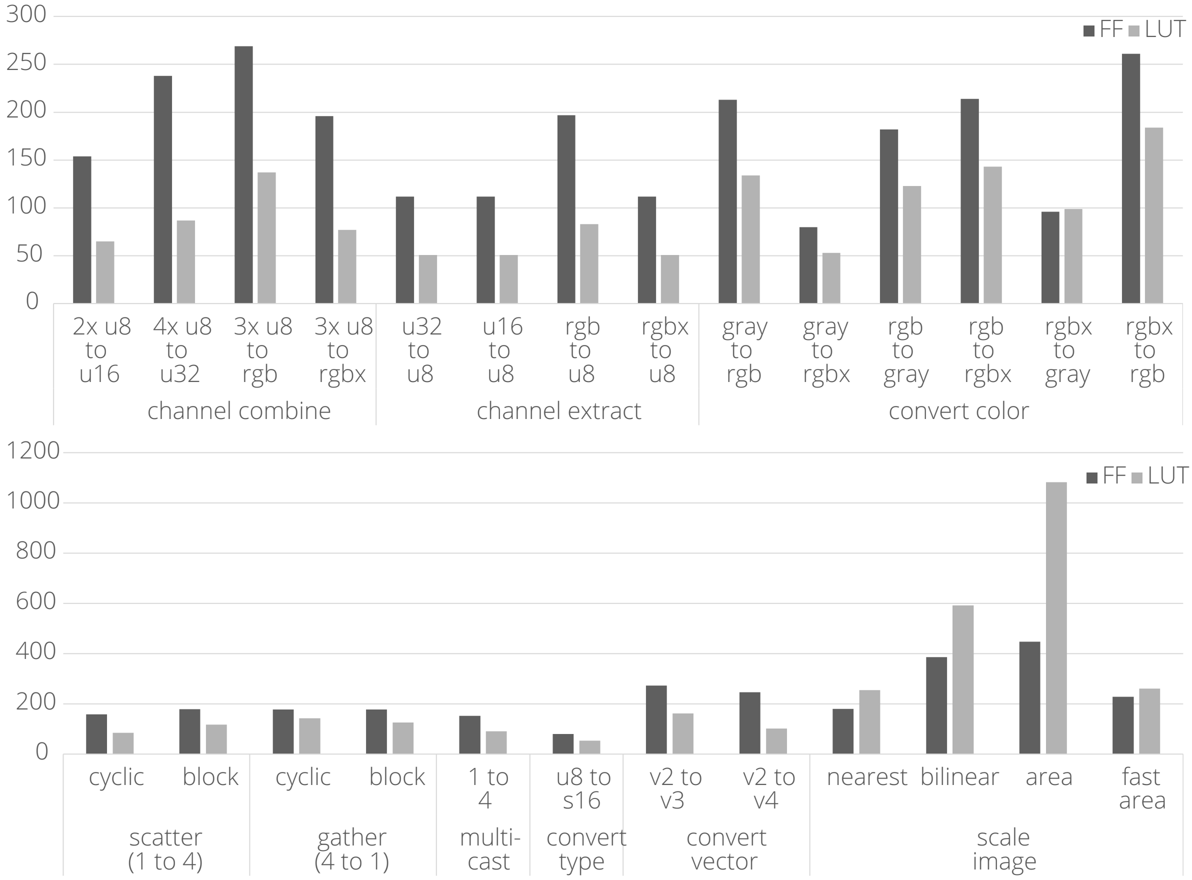


Figure 3.24: LUT and FF utilization of conversion functions.

The library implements four different interpolation methods to scale images. In the table, the first three methods scale the image down to an image resolution of 1280 × 720. The fast area method scales it down to an image resolution of 960 × 540 since the resolutions can only be multiples of each other. However, the fast area interpolation only requires 24 % of the LUTs of the normal area interpolation. In addition, it only needs half as many BRAM blocks (0.5) compared to bilinear and area interpolation. Furthermore, only the bilinear (8) and the area interpolation (7) methods require DSPs. Therefore, an algorithm should select fast area interpolation whenever possible. Apart from that, the chosen method depends mainly on the available resources and the desired accuracy.

The multicast function serves as a reference for the number of additional resources needed for the scatter and gather functions. All three functions as well as the type conversion function and the vector size conversion function can be vectorized. Unlike the XILINX vector size conversion function, this one can also convert between vector sizes which are not multiples of each other. However, the additional resources needed to buffer data is relatively small in `HiFlipVX`. The latencies of the different functions mainly depend on the resolution and the vector size. The latency of the vector width conversion function depends on the smaller of the two vector sizes. The bilinear and standard area interpolation methods, need additional time to fill their line buffers.

3.3.1.4 Image Analysis Functions

Figure 3.25 shows the resource consumption of the implemented analysis functions. The scalar operation function executes only one single operation and therefore needs only few resources. The integral image function needs to buffer one row of its output. Since the data type of the output image is 32 bit, it requires a larger buffer.

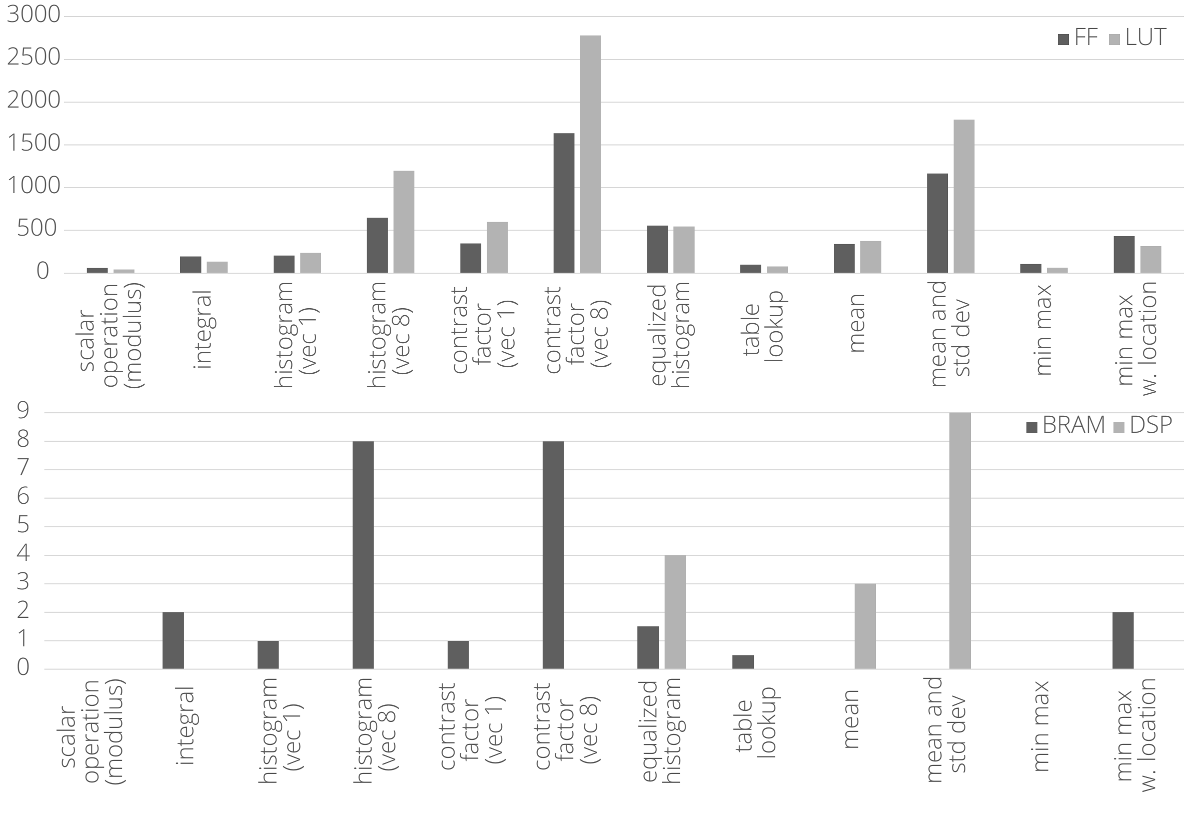


Figure 3.25: Resource utilization of analysis functions.

The histogram, equalized histogram and contrast factor functions all calculate a histogram. To achieve a pipeline with an interval of one, the histogram creation requires double buffering. The library replicates this buffer for vectorization, which increases the BRAM consumption by the same factor. To buffer its result, the equalized histogram, needs one additional buffer.

The table lookup function also buffers its input lookup table. While the min-max-location function needs to store the coordinates of the minimum and maximum, which are 32 bit wide in total.

The library calculates the final mean and standard deviation values using floating-point numbers. That is why the functions consume several DSPs and Vivado 2020.1 is used instead of 2021.1. Implementing these arithmetic operations in a separate library for fixed-point numbers would improve version independence and resource consumption. Since the standard deviation calculation needs a square root, it requires the most resources compared to the other non-vectorized functions A characteristic of most analysis functions compared to other image processing functions is that they iterate through several loops to compute their output. For n image pixels, an internal array size of k, and a vector size of v, the complexity of the functions is as follows:

- integral, min-max, or mean: n
- min-max and locations, or table lookup: $n + k$
- mean and standard deviation: $2 \cdot n$
- histogram, or contrast factor: $\frac{n}{v} + k \cdot \log_2(v) + k$
- equalized histogram: $2 \cdot n + 2 \cdot k$

The histogram, equalized histogram, and contrast factor functions, first need to reset their internal histograms (k). Due to the vectorization of the histogram and contrast factor functions, the pipeline interval needs to be reduced by the factor of ($\log_2(v)$) when summing the histograms (k), to achieve an acceptable frequency. The table lookup function first reads the lookup table (k), and the min-max location function writes the minimum and maximum location array (k) to the output at the end of the algorithm. The equalized histogram function and the mean and standard deviation function read the same image twice in a row. In general, buffering a complete image is not an option.

3.3.1.5 Feature Functions

Figure 3.26 shows the resources required by the individual feature functions, the FAST corner detector, and the Canny edge detector. All functions in the figure have 8192 elements in their input or output feature array. Only for the retain best function the output array has 4096 elements. The resources for the feature functions are overall higher than for the image processing functions, because the feature data type is 64 bit wide and the evaluation works mainly on 8 bit grayscale images. Also, the feature functions need an additional method to detect the end of a feature array, since its size cannot be determined at compile time and the library needs a vendor independent method. The gather and multicast functions in Figure 3.26 and Figure 3.24 show the resulting difference in resources. For example, the feature gather function requires 3.5 times more LUTs for four inputs in cyclic mode than the image gather function.

The deserialization function always converts to a vector size of one but must remove invalid features. When increasing the size of the input vector, both FFs and LUTs increase by the same factor. The retain best function requires one BRAM for the internal histogram and 2 URAM (Ultra Random-Access Memory) (16 BRAM) to buffer the 8192 input features. For

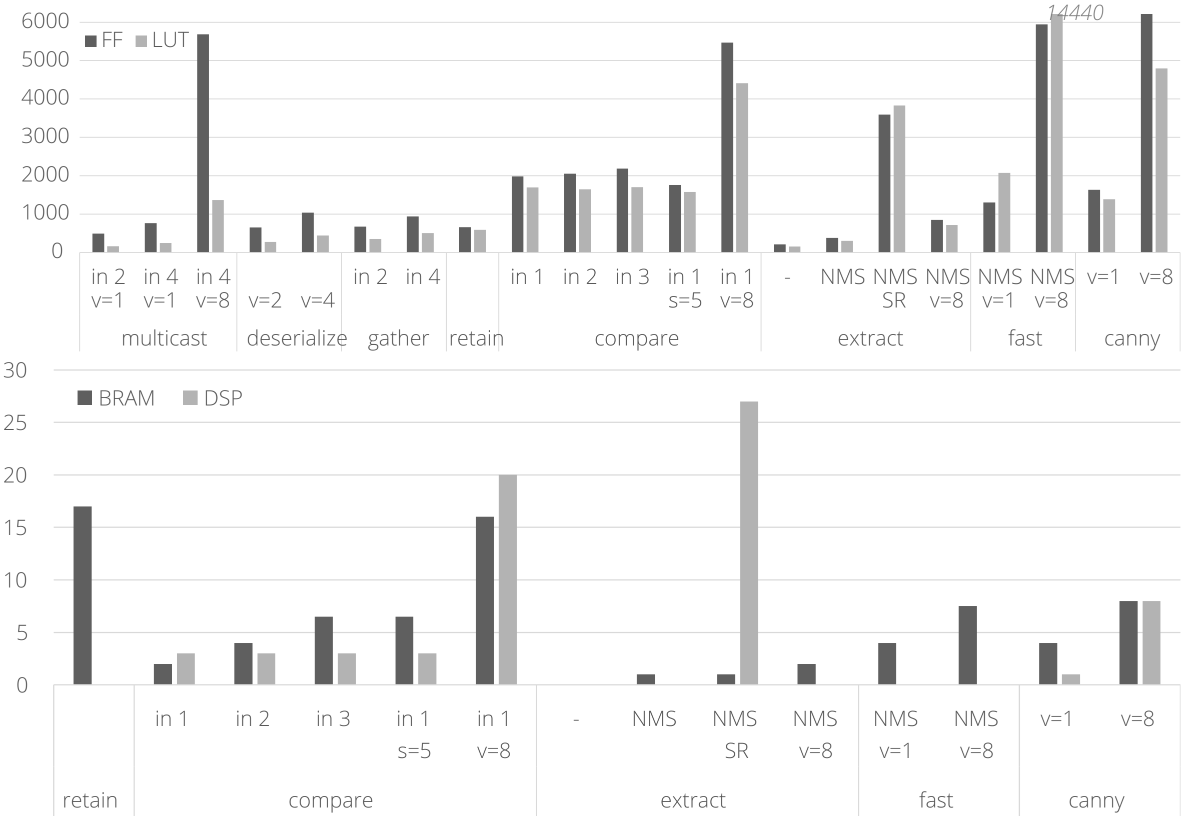


Figure 3.26: Resource utilization of feature functions. NMS (Non-Maximum Suppression), SR (Subpixel Refinement), v (vector size), σ (feature scale), in (number of inputs)

simplicity, the figure shows BRAM instead of URAM. URAM are ideal for storing larger feature vectors, as its non-configurable size of 4096 by 64 bit fits well. The feature extraction function receives an input image and extracts the features. The function offers numerous options, such as an integrated NMS or SR. The function scales very well with increasing vectorization but requires significantly more resources for SR because of the floating-point numbers used. The evaluation of the AKAZE algorithm has shown that the operation may lose too much accuracy by using fixed-point numbers.

The comparison function is the most complex of the six individual feature functions. Using a ring buffer design to store the intermediate results for comparison and the fact that features are sorted with respect to the y position, reduced the complexity from $\mathcal{O}(n^2)$ to approximately $\mathcal{O}(n \cdot \log(b))$, with b denoting the ring buffer size and n the number of input features to compare. The function automatically sets the size of the ring buffer depending on the number of detectable features in the search radius. The user can limit both the WCET and the buffer size since the function rarely reaches these limits. The influence of these two limits will be evaluated later together with the AKAZE algorithm. The parallelization value (v) is only for the internal computation to increase the number of comparisons per clock cycle. The inputs and the output of the comparison function have a vector size of one. The search radius, the number of inputs and the parallelization have an influence on the BRAM size of the ring buffer. Their default sizes in the table are $s = 1$, $in = 1$ and $v = 8$.

The last two functions consist of several individual library functions. The `DECISION` framework creates the function graph and connects the individual IP-cores using small FIFO blocks in

Vivado. The framework does not use the dataflow directive from XILINX to connect them. The Canny edge detector consists of nine functions and the FAST corner detector consists of two. The next subsection will discuss the more complex functions such as ORB or AKAZE. FAST uses the integrated NMS of the feature extraction function, as this is more resource efficient than using two separate functions. The reason for the high consumption of the FAST is mainly due to the segment test detector function. The Canny edge detector scales well with the vector size and requires only 3.5 times more LUTs for an eightfold vectorization. Starting from a certain vector size, the required bandwidth leads to a fragmentation of the BRAM consumption for both algorithms.

3.3.1.6 Comparison to Related Work

This part compares `HiFlipVX` with the xfOpenCV library from XILINX to better evaluate the resource consumption of the library. The comparison uses SDSoC 2017.4 because the tool can generate the xfOpenCV designs more easily. The evaluation board is the ZCU102, and the default function parameters are the same as in the previous measurements. Since SDSoC has a different architecture to stream data, this comparison uses a simpler FIFO interface that does not include the `last` signal.

Figure 3.27 compares the resource utilization between `HiFlipVX` and xfOpenCV for a set of selected functions. It only shows the results of BRAM or DSPs if one of the libraries utilizes these resources. In most cases, xfOpenCV complies with the default configuration of `HiFlipVX` and allows a vectorization of 8 or kernel sizes of 5 and 7. For most functions, `HiFlipVX` allows additional kernel sizes (9 and 11), data types (16 bit and 32 bit signed/unsigned), border types (replicated and undefined) or vectorization sizes (2 and 4), which makes the library much more flexible. The border type of the median filter (replicated) and the data type of the magnitude function (16 bit signed) is different from the default configuration, because xfOpenCV supports only this border and data type, respectively. A small difference between the libraries relies in the input and output data types of the Sobel and Scharr filters. In xfOpenCV the input is always u8 (unsigned 8 bit), while the output can be u8 or s16 (signed 16 bit). In `HiFlipVX` the input is always unsigned (u8, u16, u32), while the output is signed (s8, s16, s32). In general, all filter and pixelwise functions consume less FFs and LUTs for the various configurations. Other filter and pixelwise functions show similar behavior as the selected ones. `HiFlipVX` consumes on average only 0.39 % of FFs and 0.32 % of LUTs for the selected functions compared to xfOpenCV.

The libraries perform more similarly when vectorization is enabled, and the difference is more pronounced at larger kernel sizes. The main reason is that `HiFlipVX` uses separable filters or exploits the kernel coefficients. For the Gaussian kernel, xfOpenCV computes the kernel based on a standard deviation while `HiFlipVX` uses the OpenVX Gaussian kernel. Supporting an arbitrary standard deviation requires using more bits to represent the kernel and therefore require more resources. Therefore, `HiFlipVX` provides a function that can precompute fixed-point Gaussian kernel coefficients for any standard deviation, if needed. The median filter consumes the most resources for larger kernel sizes, which is due to the high number of comparisons in the sorting network. The Gaussian and box filters show the largest difference in DSP usage. The xfOpenCV Gaussian filter consumes up to 51 DSPs for the 8 bit data types while `HiFlipVX` consumes none, due to the simplified Gaussian kernel. Conversely, the `HiFlipVX` box filter consumes two DSPs for each vector element, because it uses separable filters, while xfOpenCV only uses one DSP. Therefore, `HiFlipVX` implements the Gaussian

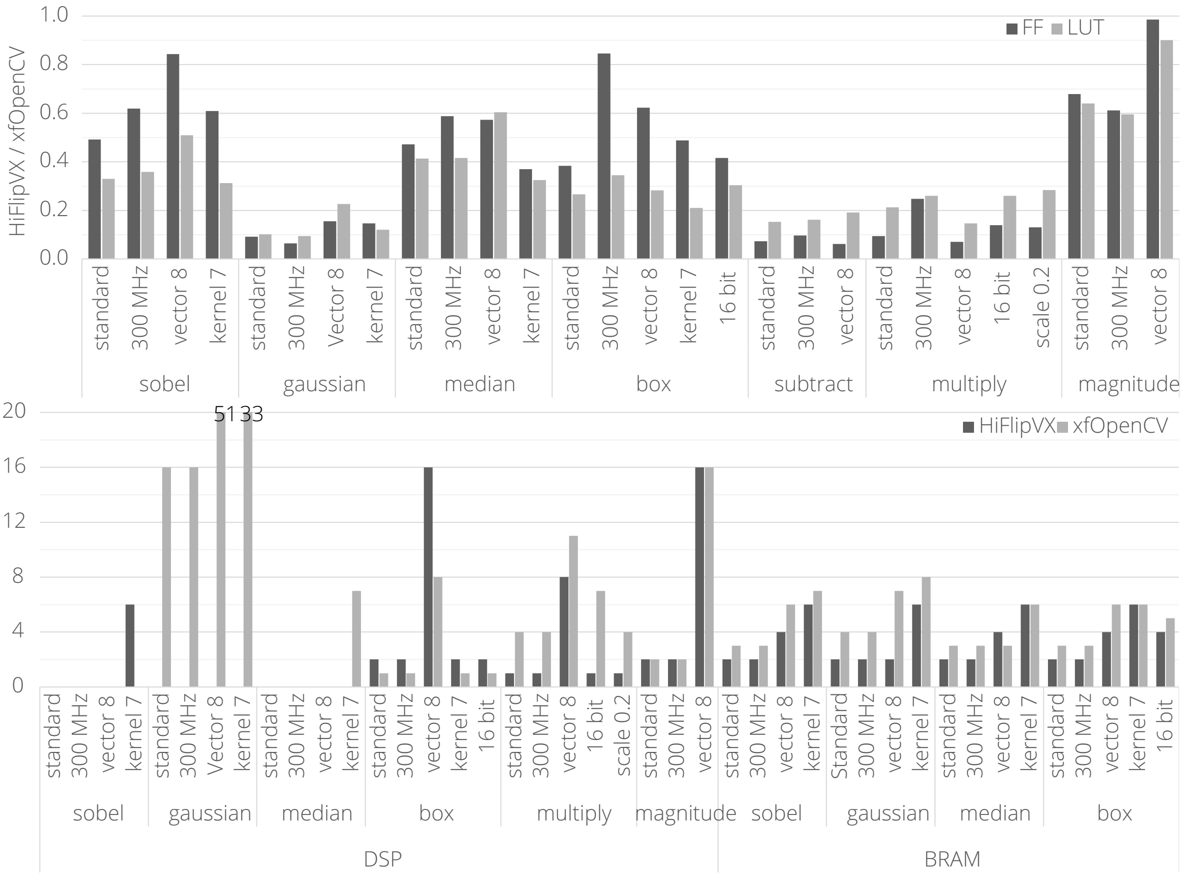


Figure 3.27: Relative FF/LUT and absolute BRAM/DSP utilization compared to xfOpenCV.

and box filter using separable and non-separable kernels. In average, `HiFlipVX` consumes 1.42 times less BRAM for the shown filter functions than xfOpenCV.

3.3.2 Feature Extraction Algorithms

The previous subsection covered the evaluation of the individual functions of the `HiFlipVX` library. This subsection analyzes the complex algorithms of this work in more detail and evaluates the library using some of these algorithms. The first part compares the different combinations of feature detection and description algorithms in software while optimizing their parameters and compares them with the proposed implementation of this work. The following parts evaluate the hardware implementation of the FREAK [24] descriptor, the AKAZE [20] detector and the ORB [19] detector. In addition, both detectors use the FREAK for their evaluation. In addition to simpler algorithms such as the Canny [18] edge detector or the FAST [37] corner detector, the `HiFlipVX` library has implemented both the ORB and AKAZE detectors and integrated them into the `DECISION` framework. This enables reusability and an easy and automated testing of different parameters. This work evaluates all algorithms in terms of their resource utilization, computation time and repeatability, and compares them with the related work. In addition, the comparison between the VHDL and HLS implementations allows a more accurate evaluation of the `HiFlipVX` library.

3.3.2.1 Software Comparison

This part compares the different combinations of feature detection and description algorithms, such as AKAZE, BRISK, ORB, and FREAK. It compares them to the optimized AKAZE-FREAK implementation presented in Section 3.2.1. It uses OpenCV to implement the different algorithms. The proposed AKAZE-FREAK optimization was written in C to be used in embedded systems. It uses 32 bit floating-point numbers, but no external libraries, except for OpenMP. Table 3.13 shows the combinations evaluated. To combine the AKAZE descriptor with other detectors would require major adjustments to the detector due to the data required.

Table 3.13: Comparison of repeatability and computation time of different combinations of feature detectors and descriptors. Best results of each column are bold.

Detector + descriptor (patternscale or patchsize)	Category 1			Category 2			Time (ms)
	Match ratio	Inlier ratio (avg.)	Inlier ratio	Match ratio	Inlier ratio (avg.)	Inlier ratio	
AKAZE + AKAZE	21.6	63.3	85.4	42.5	82.2	85.0	67.0
AKAZE + FREAK (57)	**23.1**	57.2	84.7	46.8	83.2	86.2	70.3
AKAZE + BRISK (2.9)	19.4	59.4	**87.8**	46.6	**85.3**	88.0	77.0
AKAZE + ORB (52)	11.5	37.8	86.2	37.8	84.4	88.5	68.4
BRISK + BRISK (1)	17.3	66.1	85.1	26.4	70.6	87.7	29.9
BRISK + FREAK (26)	21.7	58.7	81.4	31.6	64.0	82.5	29.5
BRISK + ORB (39)	11.2	49.8	82.0	26.5	67.7	86.3	29.9
ORB + ORB (35)	15.0	61.2	82.2	46.6	75.9	84.8	**16.0**
ORB + FREAK (10)	19.3	65.8	84.9	**48.2**	79.3	87.8	**17.7**
ORB + BRISK (0.4)	16.9	**66.8**	85.8	44.9	81.3	**89.5**	25.3
Proposed (31)	**23.1**	**70.2**	**87.9**	**60.9**	**87.2**	**92.8**	24.9

This section uses the Oxford dataset [257] to measure repeatability. The dataset contains eight grayscale images, each with five additional transformations. It divides the images into two groups depending on the type of transformation. Category one changes the images in scale, rotation or viewpoint and category two changes the image in luminance, blur, or compression. The image resolution of the dataset is between 765 × 512 and 1000 × 700. This work performs the timing measurements on an Intel Core-i7 7700 CPU with compiler optimizations (O2) and OpenCV parallelization enabled.

For a fair comparison, the parameter regulating the number of detected features was set to achieve a similar value. For AKAZE, it is the detector response threshold (0.0015); for BRISK, the AGAST (Adaptive Generic Accelerated Segment Test) corner detection [259] threshold (64); for ORB and the proposed, the Retain Best function (2048). The implementation makes an exhaustive search for the parameter that controls the descriptor size, to optimize the descriptor for the new detector. For FREAK and BRISK, it is the pattern scale and for ORB the patch size. Changing other parameters in OpenCV, did not have such strong effects on the results. The proposed algorithm changes additional parameters that are not available in OpenCV. It reduces: the number of octaves to 3 in AKAZE and FREAK, the scale and derivative factor to 1.5, the threshold to 0.0005 in AKAZE, and the smallest feature size to 3 in FREAK.

The table highlights the best two results of each column. These results show the strengths of the proposed optimizations. The implementation uses the k-nearest-neighbor algorithm for feature matching and the following metrics:

- The match ratio is the ratio between the number of detected and matching features of two images.
- The inliers ratio is the ratio between all matches and inliers (correct matches) of all images together.
- The average inliers ratio takes the inliers ratio from each compared image and averages those values, to equally weight all cases (arithmetic mean). This measurement weights the more complicated cases stronger.

The last metric shows that the proposed optimizations mainly improve the complicated cases, as the difference to the second-best result is the largest. In addition, the table shows the average computation time of the combined detection and description process. The proposed implementation achieves its maximum speedup of 3.36 when using four OpenMP threads compared to one. Compared to OpenCV, it reduces computation time by a factor of 2.82 while detecting 18.5 % more features. The Retain Best function reduces the fluctuation of detected features, which leads to better results in complicated cases. Looking at the geometric mean and not the arithmetic mean of the average inliers shows an even bigger difference. Considering the complete dataset, the geometric average of the proposed implementation is 72.57 %. The next best results are the ORB-FREAK combination with 62.99 % and the ORB-BRISK combination with 62.91 %.

3.3.2.2 FREAK

This part evaluates the proposed hardware implementations of the FREAK descriptor and the needed integral image. This work implemented a VHDL design with Vivado 2018.2 using fixed-point numbers and evaluated it on the ZedBoard (3.9). The maximum possible image resolution of the synthesized design is 2048 × 2048. Both the AKAZE and ORB implementations in this work use the FREAK algorithm, to increase their repeatability. Using FREAK instead of the M-LDB descriptor from AKAZE significantly reduces memory bandwidth consumption. On the one hand, AKAZE must store the first-order image and derivatives of each scale for its M-LDB descriptor. On the other hand, FREAK only needs to buffer the integral image.

In the synthesized design, the descriptor reaches 185 MHz and the integral image 204 MHz. The pipeline depth is 257 clock cycles for the descriptor and 8 for the integral image. The integral image would achieve 289 fps at a frequency of 150 MHz and a resolution of 1920 × 1080 for grayscale images. This would result in a bandwidth consumption of $24\,\mathrm{Gbit\,s^{-1}}$. Goebel et al. have demonstrated that this is achievable on the evaluation board [260]. In total, the design reads and writes 3360 bit for each feature from memory. This number consists of 64 bit for the input feature, 64 bit for the output feature, 2720 bit for the input integral pixels and 512 bit for the final descriptor. A larger design would pass the features and their descriptors to the next block without storing them in main memory.

Compared to the synthesized design, the implemented design achieves 166.67 MHz for the descriptor using the DMA controller developed in this work, which uses the XILINX DMA blocks. When selecting a higher frequency, the critical path is within the XILINX interconnection networks required for these DMA blocks. The design achieves 73.4 fps for 2048 feature and their descriptors, when measuring the total execution time including the memory access overhead from the ARM CPU. The main reason for the big gap to the theoretical execution time lies in the DMA blocks, which are not optimized for the random memory access required by the intensity blocks. The execution time scales linearly with the frequency from 100 MHz (44 fps) to 166.67 MHz (73.4 fps), which proofs that the implemented design is not memory bound.

The synthesized descriptor and integral image have a low resource utilization as shown in Table 3.14. The high resource utilization of the pattern rotated block results from the pipelined CORDIC function. For the intensity block, it comes from the pipelined division function. For the descriptor block, it is due to the high number of comparisons. The atan2 function has a lower utilization because the orientation block did not have to pipeline it. Since the patterns are partially precomputed and partially computed, the pattern generator consumes few resources. Precomputing all patterns in software and reading them from memory would increase resource utilization for additional DMA blocks. It would also increase latency and triple bandwidth utilization of the intensity blocks since each integral value (32 bit) would need one pattern (64 bit).

Table 3.14: Resource utilization of the FREAK and integral image hardware implementation including the utilization of the ZedBoard (3.9) in percent.

Module	FF	LUT	LUTRAM	BRAM	DSP
Parallel sum	596	448	0	0	0
Controller	209	51	0	0	0
Integral (sum)	800	515	2	2.5	0
Integral (%)	0.75	0.97	0.01	1.79	0.00
Boundary	334	131	33	1	0
Pattern unrotated	311	12	0	2	1
Pattern rotated	1673	1570	104	4.5	3
Intensity	1183	1719	24	4.5	5
Orientation	1292	1292	1	0	6
Descriptor	2414	1837	0	0	0
Send keypoint	130	59	0	1.5	0
Send descriptor	187	67	0	1.5	0
Descriptor (sum)	8115	9133	184	19.5	20
Descriptor (%)	7.63	17.17	1.06	13.93	9.09

To verify if the proposed optimizations of the integral image computation are useful, a VHDL design with adders was created and the optimization was left to the compiler. When checking the parallel prefix sum, the minimum latency increases from 3 ns to 4 ns compared to the proposed design. At the same time, the LUT consumption decreases from 448 to 266 and the FF consumption from 596 to 304. The frequency of the integral image design would also decrease from 204 MHz to 172 MHz.

Finally, the evaluation compares the software and hardware results using the full dataset. The inliers ratio decreases from 78.7 % to 77.0 %, while the average inliers ratio only decreases from 91.4 % to 90.9 %. This shows the good repeatability in comparison to the other combinations, since the inliers ratio is still better than the best ones from the OpenCV implementations shown in Table 3.13 (ORB-BRISK (88.4 %), AKAZE-BRISK (88.0 %) and AKAZE-ORB (87.9 %)). There are no changes in accuracy for the integral image since it only sums up values.

Table 3.15 compares the FREAK implementation of this thesis with the related work. Apart from the computation of the integral image, the computation time of the FREAK depends on the number of features and not on the image resolution. The proposed design has a higher fps than the other works. However, even higher frame rates would be possible with optimized DMA blocks. Zhao et al. [53] require significantly more resources for lower resolution. However, they implemented the SURF detector but have not separated its results from the FREAK. Bello et al. [52] require fewer resources but achieve a much lower frame rate with only half as many features and do not compute an integral image. Both papers do not compute the orientation, which is important for the rotational invariance of a feature. The orientation computation needs a pattern rotation block, an intensity block, and an orientation block listed in Table 3.14. These require 51.1 % FFs, 50.2 % LUTs, 46.2 % BRAMs and 70 % DSPs from the total design. In addition, there are no repeatability measurements in the other papers, which makes a comparison difficult.

Table 3.15: Comparison of the FREAK with the related work.

	Proposed [22]	Bello et al. [52]	Zhao et al. [53]
FPGA family	Artix-7	Virtex-5	Kintex-7
Resolution	1920 × 1080		800 × 600
Features	2048	1000	
Frames per second	73.4	15	60
Frequency	166.7	108	122
FF	8915	1975	60 044
LUT	9648	6706	147 190
BRAM	22	4	289
DSP	20	14	139

3.3.2.3 AKAZE

This part evaluates the proposed implementations and optimizations of the AKAZE feature detection algorithm in terms of resource utilization, computation time, and repeatability. The first design implemented the algorithm in VHDL for embedded systems with low available resources. This design was generated with Vivado 2016.4 and evaluated on a ZedBoard

(3.9). An important feature here is the reuse of hardware for different octaves to create a more resource efficient design. The second implementation uses `HiFlipVX` to implement the algorithm. The design was generated with Vivado HLS 2020.1 for the IP-cores and Vivado 2021.1 for the implemented design and evaluated on the ZCU104 (3.9). As the `DECISION` framework integrates the AKAZE algorithm, it can easily generate a design with changed parameters, such as the number of octaves or vector size.

Table 3.16 gives an overview of the two implementations and compares them with the related work. It compares the overall resource utilization, achieved frequency and frame rate. To make it more comparable, it also presents the image resolution, the stages implemented in hardware, and whether the design was executed in hardware or in simulation. All designs in the table implement two octaves of the AKAZE algorithm. Both proposed implementations use the FREAK to evaluate their repeatability. However, the FREAK is not part of the resource utilization in this table.

Table 3.16: Proposed AKAZE hardware designs for a vectorization of 4 (1st column), 8 (2nd column) and 1 (3rd column) compared to related work. All implemented for two octaves.

	Proposed [23]		Proposed [25]	Soleimani et al. [261]	Du et al. [262]	Jiang et al. [51]
Device	Zynq UltraScale+ (ZCU104)		Zynq-7000 SoC (7020)	Kintex UltraScale (KCU105)	Kintex-7 (XC7K325T)	ASIC (TSMC 65nm)
Image resolution	1920 × 1080		1024 × 768	1280 × 720	640 × 480	1920 × 1080
Contrast factor	✓	✓	✓	✓	✗	✗
Nonlinear scale space	✓	✓	✓	✓	✓	✓
Detector	✓	✓	✓	✗	✓	✓
Descriptor	(✓)	(✓)	(✓)	✗	✓	✓
Matching	✗	✗	✗	✗	✓	✗
Test on device	✓	✓	✓	✗	✓	✓
FF	79 835	123 944	38 314	65 028	157 122	-
LUT	69 722	127 063	24 945	112 596	196 134	-
LUTRAM	-	-	-	72 276	28 068	-
BRAM	147.5	232	108	524	291	-
URAM	4	4	-	-	-	-
DSP	136	272	157	31	228	-
Frequency (MHz)	214.5	150	100	100	100	200
Frames per second	360	480	98	304	784	127
Million PPS	746	996	77	280	241	263

The embedded hardware design [25] stands out due to its low resource consumption and is therefore the only design that fits on a low-cost FPGA. Comparing it with the FPGA devices used in the table, it stands up well with the achieved frequency of 100 MHz. Theoretically there is even enough space for the FREAK descriptor on the ZedBoard. Therefore, both VHDL implementations presented in this thesis have been combined in a separate design. Unfortunately, this combination did not have enough BRAM when adding all DMA blocks for memory access. A solution could be to create an optimized design for memory access as shown in [263]. Otherwise, the ARM CPU would need to process parts of the algorithm.

The `HiFlipVX`-based design [23] stands out due to its high performance. It uses a vectorization of four and eight to achieve this performance. Considering its performance and resolution, it performs even better in terms of resources. This proves that with HLS resource optimized designs are also possible. The resolution has a very strong influence on the BRAM consumption, which is usually the limiting factor.

The advantage of FPGAs over GPUs is their ability to stream data. However, there is a certain overhead for memory access and filling buffers, sliding windows and pipelines. In the `HiFlipVX` AKAZE implementation this overhead is 13.5 % (17.5 %) for a vectorization of 4 (8), in comparison to reading an image with the same vector size (*rows · cols/vector*). This overhead is very low considering the memory access, the numerous FIFO blocks, and the execution of 95 computer vision functions. The reason for the lower frequency in the second design is the contrast factor function, which accesses sixteen BRAM based histograms in parallel due to vectorization and double buffering.

A design with three octaves and a vectorization of four achieved 308 fps for a frequency of 187.5 MHz. The design consists of 163 vision functions. The overhead is only 17.2 %, which is relatively low considering that 71.6 % more vision functions have to be executed than in the implementation with two octaves. The reason for this lies in the parallel execution of the feature detection part, whereas the creation of the nonlinear scale space is still sequential. When increasing the octaves, the gather function also becomes a limiting factor for the frequency due to the increasing number of inputs. Using a design with a tree of multiple gather functions could remedy this, but at the cost of an increased resource utilization.

Du et al. [262] achieve a high frame rate of 784 fps but use a resolution with 6.75 times less pixels. The `HiFlipVX` design achieves a measured frame rate of 1998 fps for the same resolution. Soleimani et al. [261] achieve a frame rate of 304 fps, but also use a lower resolution. The last metric in the table regarding the millions of calculated pixels per second includes the resolution. Since their work focused only on the nonlinear scale space, they lack the feature detector. Their implementation uses buffers that are twice the size of the image to store the conduction coefficients and the FED image on-chip. As a result, their MMU (Memory Management Unit) already requires 524 BRAM, which greatly limits the scalability of their design.

Jiang et al. [51] are the only ones that use the same resolution but achieve significantly less fps. Since their design is for an ASIC, it is more difficult to compare resources. They use 0.95 million gates and 2.12 Mbit of on-chip memory. For comparison, the Artix chip on the ZedBoard has 1.3 million gates. They will need about 57.5 BRAM if they perfectly utilize the 36 864 bit of each of them. However, FPGA designs usually suffer from fragmentation due to the limited configuration possibilities and cannot use the entire BRAM. Furthermore, they are LUT-based and therefore not as optimized as an ASIC.

Table 3.17 further breaks down the proposed designs by their resources. The `HiFlipVX` design from Figure 3.16 computes both octaves in parallel, while the VHDL design computes them sequentially. In addition, the HLS-based design processes four pixels in parallel. The table lists the FIFO blocks, needed between the nodes, separately. The design uses URAM instead of BRAM, since there was a lot of unused URAM. However, BRAM would also be sufficient. The connection between the Gaussian and FED function needs BRAM (in total 6 edges). Otherwise, it would cause a deadlock. The connection before and after the vector conversion function also needs BRAM (in total 2 edges). This is because the scaling function only outputs pixels in every second row, although it reduces the image size by a factor of four. In addition, it is useful to use BRAM before and after the feature compare function to optimize throughput (in total 22 edges). Almost all other buffers use LUTs instead of BRAM for their FIFO blocks.

Table 3.17: AKAZE resource consumption of (A) embedded design (top) and (B) `HiFlipVX` design with vectorization of 4, latency restrictions and no SR (bottom).

	Stage	LUT	FF	BRAM	URAM	DSP
A	Contrast factor	2704	2932	4.5	0	13
	Nonlinear scale space	13 803	19 806	55.5	0	136
	Feature detection	8438	15 576	48	0	8
B	Contrast factor	3772	3295	7	0	4
	Nonlinear scale space	24 351	27 265	49	0	0
	Feature detection	41 599	49 275	91.5	4	132
	Inter node buffers	7746	13 524	0	39	0
	System	7223	11 634	7.5	0	0

This work tests both designs for repeatability using the Oxford dataset [257]. Figure 3.28 shows a comparison between the original software implementation and the optimized embedded hardware design, using the inliers ratio to determine repeatability. The only difference between the various configuration flags lies in the contrast factor. In general, the modified algorithm shows a similar behavior to the original algorithm for the different images. The software implementation achieves 76 % inliers on average, while the hardware design achieves 73 % for the datasets. In general, the reduction in inliers is partly due to the use of fixed-point numbers in the VHDL design. The lower percentage in some datasets is mainly due to the omitted SR.

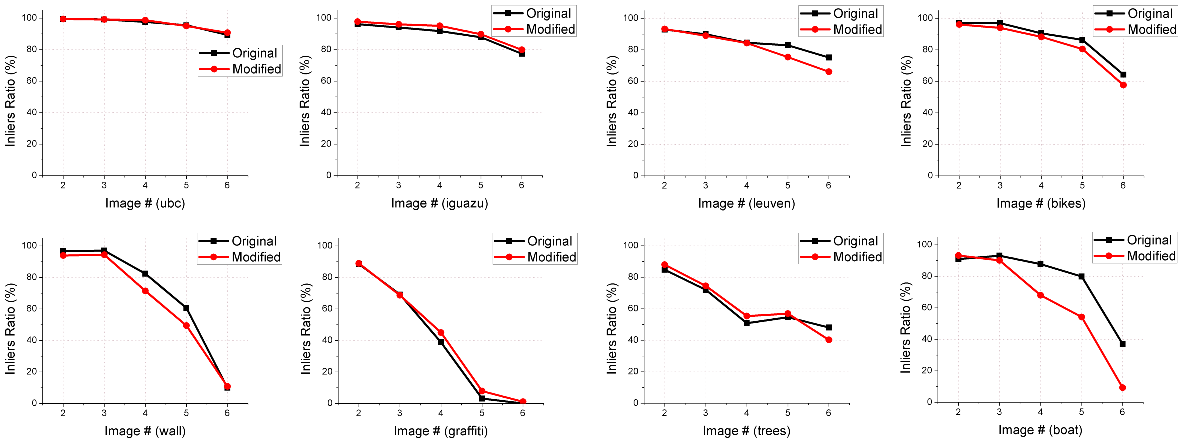


Figure 3.28: Repeatability comparison of the embedded AKAZE design.

Figure 3.29 compares the average inliers ratio of the proposed optimizations presented in Section 3.3.2.1. It compares the software implementation against different configurations of the `HiFlipVX` implementation. While the software implementation uses floating-point values, the hardware implementation computes on 8 bit grayscale images. Only the DoH needs to output 16 bit values, otherwise the accuracy would decrease too much. The library also offers to use accuracies of 16 bit for the images and 32 bit for the DoH output. However, this did not result in a noticeably better repeatability, but consumed significantly more resources in the affected functions. The figure shows the results for the hardware design with two or three octaves, with or without an iteration constraint compare function, and with or without SR.

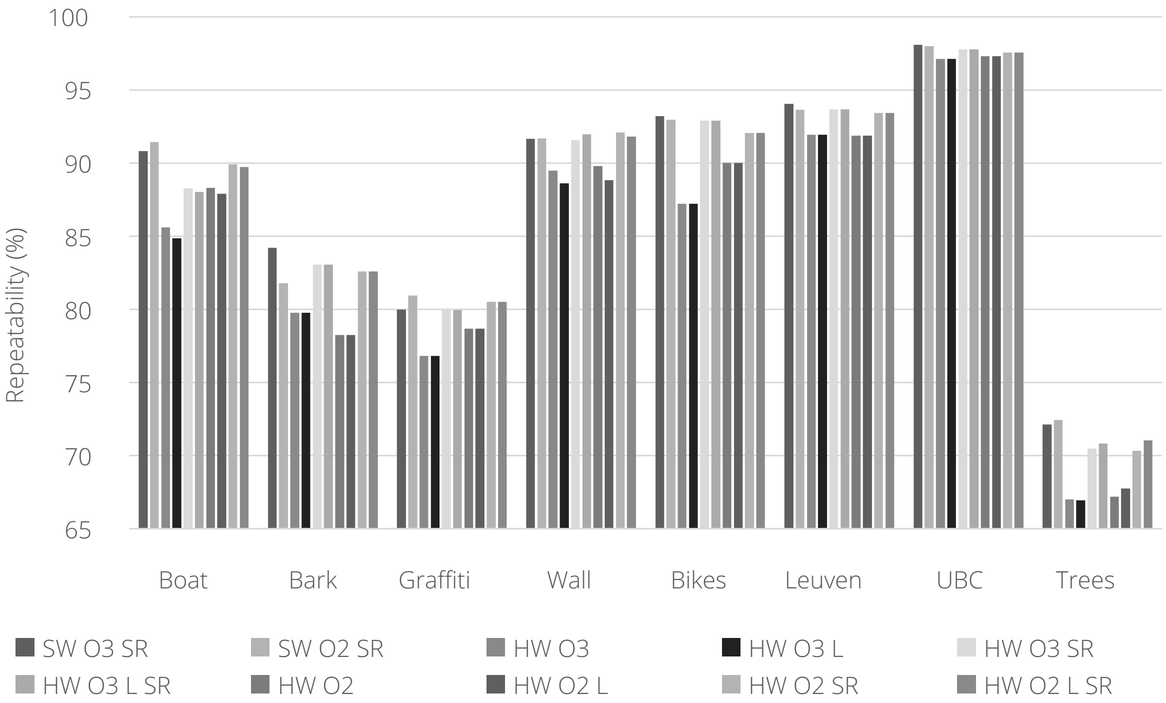


Figure 3.29: Repeatability comparison of the `HiFlipVX` AKAZE design for different configurations. SW = 32 bit floating-point, HW = 8 bit fixed-point for images and 16 bit fixed-point for DoH, O2/O3 = 2/3 octaves, L = latency restriction for compare function, SR = Subpixel Refinement using 32 bit floating-point for both SW and HW.

The software implementation performs best. However, the hardware implementation with SR achieves similar results. The difference is only between 0.05 % to 2 %. The only exception is the "Boat" dataset with about 2.6 % for three octaves. However, they perform better than the hardware implementation without SR in all cases, with an improvement of up to 5.6 % for the "Bikes" dataset for three octaves. The hardware has an average loss of less than 1 % compared to the software if it uses SR. If it omits the SR, it loses an average of an additional 2.6 % for three octaves and 1.6 % for two octaves. This shows that a fixed-point hardware implementation has the potential to produce similar repeatability results compared to a 32 bit floating-point software implementation. The results also show that the difference in repeatability is negligible if the developer carefully chooses the maximum number of iterations for the comparison module. The chosen value for a comparison function is equal to that of the slowest function (DoH) in the scale space level to avoid slowing down the pipeline.

3.3.2.4 ORB

This part evaluates the proposed implementations and optimizations of the ORB algorithm in terms of resource utilization, computation time, and repeatability. More precisely, a VHDL implementation that is highly resource optimized and embedded into a VPS, and a parameterizable HLS-based implementation using the `HiFlipVX` library. The evaluation shows the advantages of the combination with the FREAK descriptor and a comparison with the SoA. Both hardware designs have already been described in Section 3.2.4.

Table 3.13 showed the advantages of using the FREAK descriptor instead of the BRIEF for the ORB. For a fair comparison, some parameters of the two algorithms available in OpenCV have already been optimized with respect to their repeatability. Table 3.18 shows the fine-tuned parameter settings, of which some were only available due to the original source code of the FREAK [256]. It was determined by testing various parameters of the algorithms in nested loops. Due to the FREAK, the ORB does not need the calculation for the orientation or the Harris detector. The configuration increases the number of pyramid levels to eight and adjusts the scale factor to 1.14. The verification of whether a feature is too close to the border of an image was set to zero, leaving this to the FREAK descriptor. The Retain Best functions therefore retain a total of 2500 features, thus leaving about 2000 features after the descriptor. The number of retained features per level depends on the image size and decreases with each level. The number of FREAK octaves corresponds to the number of pyramid levels of the ORB detector. The pattern scale for the retina pattern of the FREAK is set to 25. The size of a feature is its radius in pixels.

Table 3.18: Best configuration for ORB detector + FREAK descriptor.

ORB pyramid levels	8	ORB retain best features	2500
ORB scaling factor	1.14	FREAK number of octaves	8
ORB boundary size	0	FREAK pattern scale	25
ORB patch size	7	FREAK smallest feature size	4

The measurements use the Oxford image dataset [257]. As Table 3.13 shows, using the FREAK descriptor instead of the BRIEF increases repeatability from 84.21 % to 86.83 %. Excluding the calculation for the orientation and the Harris detector further reduces the total computation time. For the same configuration, the ORB implementation of the `HiFlipVX` library achieved an average repeatability of 85.24 %. However, it only uses unsigned 8 bit data, while the software implementation uses floating-point numbers. Scaling the image for each layer causes a rounding error, which could be the reason for this difference. The combination of ORB and FREAK achieves 91.88 % repeatability for the optimized values in Table 3.18, which is comparable to the proposed AKAZE-FREAK implementation. In contrast, for the same number of features, the AKAZE-FREAK combination finds 1.42 times more matches and performs better for the more difficult cases, as indicated by a higher average inliers ratio of 3 percentage points. Compared to the software implementation, the HLS version achieves a value of 89.16 % for the same parameters.

Table 3.19 shows the computation time of the VHDL design for a 1920 × 1080 resolution compared to an OpenCV implementation running on an Intel Core-i5 2400 CPU and a pure C implementation. The latter runs on the ARM processor of the ZedBoard (3.9) without an OS in a single thread. Each system executes the algorithm using the optimized parameters of

Table 3.18 in a four-level and eight-level design. The hardware achieves a speedup of 18 for four levels and 27 for eight levels in comparison to the Intel CPU. It achieves high parallelism due to the deep pipeline and streaming of data. Since all Retain Best functions share the computation to obtain the 2500 best features, the eight-level FPGA design is faster than the four-level one. In addition, the VHDL implementation runs in the proposed video processing system and achieves the minimum requirements of 60 fps at a resolution of 1920 × 1080.

Table 3.19: Comparison of ORB execution time for a 1920×1080 resolution between software and hardware (VHDL).

	frame time (ms)		throughput (fps)	
	4-level	8-level	4-level	8-level
Intel Core-i5 2400	297	397	3.37	2.52
ARM Cortex-A9	431	1257	2.32	0.80
FPGA (100 MHz)	15.69	14.64	63.7	68.3

Table 3.20 shows the resource consumption of the implemented design for a resolution of 1920 × 1080. The VHDL implementation runs on a Zedboard with a frequency of 148.5 MHz. The `HiFlipVX` implementation runs on a ZCU104 (3.9) with a frequency of 214.5 MHz. The VPS results shown in the table use three frame buffers and are independent of the object detection algorithm. The results of the VHDL design show that it easily fits on a low-cost FPGA and even leaves enough resources for further implementations. For example, the proposed FREAK implementation from Section 3.3.2.2 would fit on the same board.

Table 3.20: Resource utilization of the proposed ORB designs.

	[26]			`HiFlipVX`
	4-level	8-level	VPS	4-level
FF	6272	12 915	12 120	13 429
LUT	5700	12 732	9900	13 258
DSP	3	3	8	0
BRAM	60.5	95.5	4.5	14.5
URAM	0	0	0	4

The HLS implementation shown in Figure 3.11 used the integrated NMS of the feature extraction function. This slightly reduced resource consumption compared to using two separate functions. A separation of the two functions would have an advantage in other systems where a fine-grained distribution to different compute nodes is needed. The additional consumption of LUTs and FFs in the HLS implementation is mainly due to the four segment test detector functions. These functions consume 62 % of the LUTs needed for the design. Considering that one URAM is as big as eight BRAM, it reduces the total consumption of memory blocks compared to the other design. This is important because BRAM consumption is often the limiting factor.

Table 3.21 compares the proposed design with the related work and an implementation using `HiFlipVX`. Compared to the related work, this paper implements the multilevel scheme, which is necessary for scale invariance. The resource consumption of the detector increases with the number of levels in the scale space. However, due to the lower resolution, it requires slightly less resources in higher levels. On the other hand, they implement the descriptor in their work. However, Section 3.3.2.2 demonstrated the proposed implementation of the FREAK descriptor in the context of this work. The summed resources of both algorithms are 14 387 FFs, 14 833 LUTs, 80 BRAM and 23 DSPs. Due to the high frame rate of the FREAK implementation, its streaming design, and low memory bandwidth consumption, it would not reduce the overall performance.

Table 3.21: Comparison of proposed ORB implementation to related work.

	Proposed `HiFlipVX`	Proposed [26]	Lee [49]	Fularz [50]
FPGA (3.9)	ZCU104	ZedBoard	Artix-7	ZedBoard
Resolution	1920 × 1080	1920 × 1080	640 × 480	1920 × 1080
Detector	✓	✓	✓	✓
Descriptor	(✓)	(✓)	✓	✓
Multiscale	4-level	4-level	✗	✗
Retain best	✓	✓	✗	✗
Frequency (MHz)	214.5	148.5	100	100
Frame rate (fps)	103	63	100	48
Pixel rate (MP)	213.0	130.6	30.7	99.5
FF	13 429	6272	6411	9543
LUT	13 258	5700	31 677	4118
BRAM	46.5	60.5	31	31

In addition to the multilevel design, the additional BRAM consumption is mainly due to the Retain Best function, which needs to buffer a complete feature vector and exists in each level. Without this feature, the `HiFlipVX` design would require only 20.5 BRAM. In turn, the function brings several advantages. It makes computation time predictable and reduces WCET. In addition, measurements have shown that the function improves repeatability by retaining only the strongest features.

The `HiFlipVX` implementation requires more resources for the same configuration but has other advantages. One of them is the numerous parameters that allow the detector to be easily adapted to other applications or the required performance. For example, by vectorization or a different interpolation method for scaling. When taking out the frequency difference, the performance is still about 13 % higher in the HLS design. The VHDL design, in turn, embeds the ORB detector in a VPS. It processes the video stream, detects the features in hardware, sends them to the ARM CPU and draws them into the output video stream.

3.3.3 Neural Network Extension

The first two parts of this subsection evaluate the image processing and feature extraction functions and algorithms of the `HiFlipVX` library. This part focuses on the neural network extension. First, this work performs a detailed evaluation of the various functions of the library. For this purpose, it evaluates various parameter settings to make general assumptions. The later part evaluates the design of larger algorithms using the MobileNets [21] algorithm. Finally, there is a comparison with related work. This subsection uses the ZCU104 (3.9) and the SDSoC 2019.1 development tool from XILINX to create the various designs. It takes the implementation results of the individual designs from the Vivado project created with SDSoC.

3.3.3.1 Library Functions

This part evaluates the single neural network layers of `HiFlipVX`. Table 3.22 shows the default configuration of the compile time parameters of the library functions.

Table 3.22: Default configuration of the library compile time parameters.

Batches	4	V_{IFM}	1
Input	64 × 64	V_{OFM}	1
Output	64 × 64	Frequency	100 MHz
IFM	32	Data type	unsigned 8 bit
OFM	32	Bias data type	unsigned 8 bit
Bias size	OFM	Fixed-point position	8
Kernel size	3 × 3	Overflow	saturate
Pooling size	2 × 2	Rounding	to zero
Padding size	1 × 1	Buffer coefficients	yes

The left side of the table includes the regular parameters of a neural network, which are not FPGA specific. The library supports two pooling types and nine activation function types. It executes batches (images) in a sequence and not concurrently. The input resolution can differ from the output resolution, but it must be larger. This is only possible for the two convolution functions and the pooling function to implement a stride. Only the 3D convolution and the Fully Connected layers have both IFM and OFM, all other layers have only one feature map (IFM). `HiFlipVX` allows values from 1 to 2048 for the resolution, batch size, and feature map size. The bias size can be 0, OFM, or batches times OFM for the two convolution functions and the Fully Connected layer. Both convolution functions can adapt the kernel size. It is ($n \times m$), where (n) and (m) can differ but must be odd numbers and must be in the range from 1 to 9. The same is true for the pooling size, where the numbers can also be even. Only the pooling function can specify its pooling and padding sizes. Its size can be between 0 and half of the pooling size. The convolution functions use an automatic zero padding equal to half the kernel size ($\left\lfloor \frac{K_{y,x}}{2} \right\rfloor$).

The right side of the table contains parameters that are more specific to an FPGA design, such as changing the frequency. All functions are parallelizable by using the V_{IFM} parameter. However, only the 3D Convolution and Fully Connected layers can also use the V_{OFM} parameter. It enhances the performance improvement possibilities and is necessary since they are the most computationally intensive layers. For both parallelization parameters, the library allows values from 1 to 128. Compared to the other functions of the library, the inputs, outputs, and weights of the neural network extension can also use floating-point numbers (`int8`, `uint8`, `int16`, `uint16`, `float32`). The biases can have a different data type than the other inputs if fixed-point numbers are used and also allow signed and unsigned 32 bit values. Several CNN implementations use this approach. The fixed-point position determines the size of the fraction and must be less than the number of digits of the data type. Signed data types require at least 1 bit for the integer part. For arithmetic calculations, especially fixed-point numbers, the layers must check for overflow and apply the desired rounding method. This is the same as for the image processing functions of the library. It can buffer coefficients (weights and biases) on first use (`buffer coefficients`). In contrast, Figure 3.21 buffers the coefficients outside of the functions to increase the efficiency of the coefficient reading process.

The evaluation uses the MAPE (Mean Absolute Percentage Error) of the hardware implementation in comparison to a floating-point software implementation, to verify their correctness. Table 3.23 shows the results using the default configuration and quantized random input numbers in the range from 0 to 1, where $\{x \in \mathbb{R} | 0 \leq x < 1\}$. The computation of MAPE is problematic when the divisor is zero. Therefore, the measurements do not consider individual results where the divisor is less than 10^{-6}. The fixed-point positions for the data type in the table are 16 (`uint16`), 15 (`int16`), and 24 (`float32`). The MAPE of 0.68 % for the 3D convolution is due to the high number of multiplications and additions needed for each output pixel. The behavior is similar for other functions that add and/or multiply many variables. A similar behavior is observed for other functions where many variables must be added and/or multiplied. The `float32` calculation can have a very small error for functions that need to calculate a sum over multiple loop iterations. This is due to the use of fixed-point arithmetic for this summation to meet the iteration interval of one for the pipelined loops.

Table 3.23: The MAPE between the layers of the `HiFlipVX` neural network extension and a 32 bit floating-point software implementation.

Neural network layer	uint16	int16	float32
3D convolution	0.3413	0.6804	0.000 03
Depthwise convolution	0.0127	0.0261	0.000 00
Pooling (max)	0.0000	0.0000	0.000 00
Activation (relu)	0.0000	0.0000	0.000 00
Batch normalization	0.0390	0.1012	0.000 04
Fully connected	0.0000	0.3421	0.000 00
Softmax	0.2104	0.4245	0.000 01

Table 3.24 shows the resource utilization of the implemented and synthesized designs using the default configuration. In this table, the Softmax and Fully Connected layers have 256 IFM and 256 OFM, since the resolution of these layers is 1×1. As shown in the table, the

difference between the SDSoC estimation and the final hardware implementation is quite large for FF and LUT consumption. SDSoC uses Vivado HLS in the background and thus comes to a similar deviation. The implementation results in the table only include the IP-core of each function and not the surrounding blocks generated by SDSoC. The consumption of DSPs differs in the estimation because the library does not specify whether LUTs or DSPs are used for arithmetic computation, as this can vary depending on the available resources. The Fully Connected layer has many coefficients and therefore requires a lot of BRAM. Therefore, it makes sense not to buffer the weights, because the layer only needs each weight once per batch. The 3D convolution consumes more BRAM than the depthwise convolution because it needs to buffer more coefficients.

Table 3.24: Resource utilization & latency of synthesized (gray) & implemented (black) design for each layer. Fully Connected and Softmax layers have 256 IFM and 256 OFM.

	BRAM		DSP		FF		LUT		Latency
3D convolution	3	3	5	9	1293	439	2114	534	4 326 401
Depthwise convolution	2	2	5	9	872	233	1488	407	135 201
Pooling (max)	0.5	0.5	0	0	162	128	690	191	135 201
Activation (relu)	0	0	0	0	26	26	118	17	131 073
Batch normalization	0	0	14	13	6618	3290	6716	3532	131 102
Fully connected	17	17	0	1	421	347	842	282	65 537
Softmax	0	0	19	19	2568	1987	4610	2939	522

In addition, the table shows the estimated latency per batch. It is well known that the process of 3D convolution is the most computationally intensive part in many CNN algorithms and therefore needs more parallelization. The Softmax function is the least computationally intensive function. A hardware/software co-design would perform this function on the CPU, as it is also quite resource intensive. By adding the proposed multilevel pipelining approach, Batch Normalization can compute the three internal functions in almost the same time as the activation layer. Because of this approach and the computationally intensive operations such as division and square root, the function requires more resources. Depthwise convolution and pooling require some additional cycles because of the internal line buffers.

Table 3.25 shows the resource utilization of the implemented design for the different activation functions using 16 bit unsigned values. As expected, all functions that contain an exponent, logarithm, or division in their equation consume more resources. Using exponential functions instead of hyperbolic functions reduced resource consumption. For the square root function, there is an option for a relaxed mathematical calculation to reduce resource consumption by reducing the precision of the fraction. Consequently, the accuracy is reduced to 0.37 %. Because of accuracy, the function uses floating-point operations for most complex operations. However, due to quantization, a small error rate remains for these functions.

Figure 3.30 shows the relative resource utilization for different parameter settings compared to the default configuration. As expected, changing the frequency mainly increases the FFs (43 % on average), but also the LUTs (8 % on average). However, it has no effect on the BRAM or DSP usage. The library also supports floating-point numbers for high accuracy, for fast

Table 3.25: The MAPE and resource utilization of the implemented design of the activation functions for unsigned 16 bit values.

	BRAM	DSP	FF	LUT	MAPE
Logistic	0	15	1362	2146	0.001 26
Hyperbolic	0	17	1549	2370	0.025 43
Relu	0	0	26	17	0.000 00
Brelu	0	0	26	25	0.000 00
Softrelu	0	28	1418	2112	0.001 44
Abs	0	0	26	17	0.000 00
Square	0	1	28	28	0.000 00
Sqrt	0	0	164	384	0.001 39
Sqrt (relaxed)	0	0	113	243	0.369 90
Linear	0	0	26	25	0.000 00

integration, or for testing. They have no impact on the latency of the various library functions, except for additional pipeline stages, but they have a large impact on resource usage: 432 % more LUTs, 784 % more FFs, and 423 % more DSPs. When using 16 bit fixed-point values to increase accuracy, there is only a small increase in LUTs (25 %), FFs (17 %), and DSPs (2 %). This again shows the importance of quantization in FPGA designs. BRAM usage always scales with the bit-width of the data type used. Increasing the kernel size has a similar effect for 3D and depthwise convolution. In both cases, the DSP amount grows with the kernel size. The BRAM increase depends on the coefficient size ($k_y \times k_x$) and the line buffer amount ($k_y - 1$). LUTs and FFs only increased by 85 % and 65 %, respectively, for 2.78 times the number of weights.

This work examined the parallelization parameters in more detail, as the correct values are important for an efficient design. The Batch Normalization layer scales well with parallelization because resource-intensive functions do not need to be calculated multiple times. Only the increase in DSP usage approximates to a linear behavior. The used DSPs of all other functions scale linearly with the degree of parallelization. The used LUTs and FFs of the Pooling layer scale less than linearly with the degree of parallelization. The Fully Connected layer even shows a reduction in used FFs and BRAM through fragmentation and optimized usage.

The 3D convolution has a combined vectorization of $V_{IFM} \times V_{OFM}$. Different combinations of V_{IFM} and V_{OFM} were evaluated to find an optimized combination. The combined vectorization results in a parallelization (V) of 2 (2 × 1, 1 × 2), 4 (4 × 1, 1 × 4, 2 × 2), 8 (8 × 1, 1 × 8, 4 × 2, 2 × 4) or 16 (16 × 1, 1 × 16, 8 × 2, 2 × 8, 4 × 4). This thesis makes some assumptions based on these combinations. The greater the imbalance between V_{IFM} and V_{OFM}, the more resources are used on average. If, for the same V, V_{IFM} is greater than V_{OFM}, the average usage of LUTs and FFs increases slightly by 6 % and 10 % respectively. In addition, a high V_{IFM} can cause more BRAM to be used if it worsens line buffer fragmentation.

Additionally, one 3D convolution layer has been implemented with a high parallelization to show the performance improvement in comparison to a baseline implementation. The ARM CPU of the ZCU104 executed this baseline implementation with a frequency of 1.2 GHz

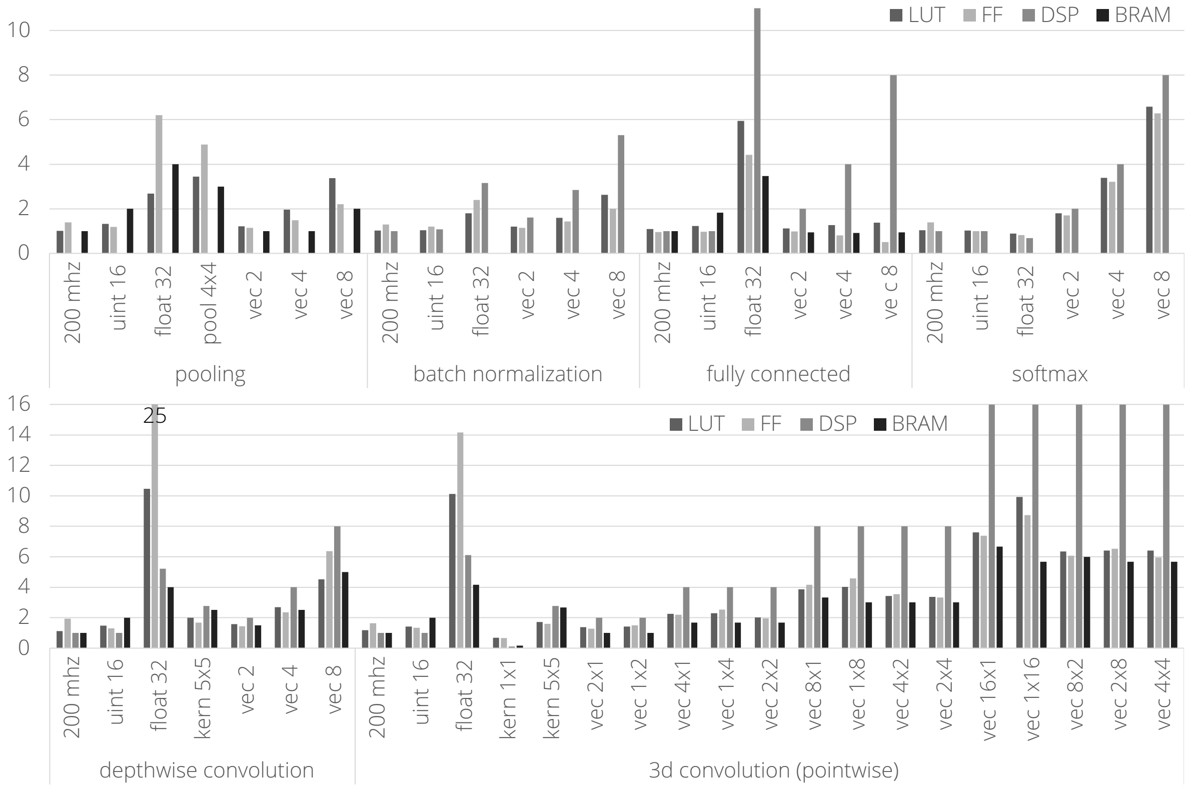


Figure 3.30: Relative resource utilization compared to the default configuration. Value is not reported if it is zero. 3D convolution has a vectorization of $V_{IFM} \times V_{OFM}$.

in release mode using the O3 optimization option. The convolution function achieved an acceleration of 260 on the real system using the default configuration with (V_{OFM} = 8), (V_{IFM} = 8) and a frequency of 200 MHz. The CPU performed the measurements without running an operating system. The consumed resources for the convolution function are: 8858 LUTs, 7679 FFs, 576 DSPs and 66 BRAM. The BRAM has increased due to fragmentation and a high demand of on-chip bandwidth. The execution time of the hardware is 873 µs, which includes the cache flushing and data movement between the FPGA and DMA.

3.3.3.2 MobileNets

Before implementing the different layers of MobileNets, the optimal parameters must be determined. When building a deep pipeline, the system is usually as fast as its slowest component. Table 3.26 shows the offline calculations for an optimal setting of the different modules containing the MobileNets layers. All parameters not listed in the table use the default configuration. The algorithm defines the parameter values for the resolution and feature maps. Section 3.1.7 already explained how to calculate the latency of the individual functions. The estimation does not consider the number of pipeline stages, as it has almost no impact. The maximum latency in the right column shows the bottleneck of the design.

The vectorization settings discussed in Section 3.2.7 are adapted to improve the maximum latency, while taking the available resources for DSPs and BRAM into account. Based on the parameters of the table, it is possible to estimate these resources, which is important

Table 3.26: Vectorization tuning for the proposed MobileNets modules. Latency is calculated for each layer separately without pipeline stages. Depthwise (dw) and pointwise (pw) latency of Batch Normalization (bn) and convolution are reported. Maximum latency of all layers within a module is shown on the right.

Module	Resolution		Feature maps		Parallelization				Estimated latency (clock cycles)				
ID	Input	Output	IFM	OFM	v_{dw}	V_{IFM}	V_{OFM}	v_{pw}	dw_{conv}	dw_{bn}	pw_{conv}	pw_{bn}	max
1	224 × 224	112 × 112	3	16		3	8	2			101 250	100 368	101 250
2	112 × 112	112 × 112	16	32	2	8	8	4	102 152	100 368	100 352	100 368	102 152
3	112 × 112	56 × 56	32	64	4	8	8	2	102 152	25 104	100 352	100 416	102 152
4	56 × 56	56 × 56	64	64	2	8	16	2	103 968	100 416	100 352	100 416	103 968
5	56 × 56	28 × 28	64	128	2	8	8	1	103 968	25 152	100 352	100 608	103 968
6	28 × 28	28 × 28	128	128	1	8	16	1	107 648	100 608	100 352	100 608	107 648
7	28 × 28	14 × 14	128	256	1	8	8	1	107 648	25 344	100 352	50 688	107 648
8	14 × 14	14 × 14	256	256	1	8	16	1	57 600	50 688	100 352	50 688	100 352
9	14 × 14	14 × 14	256	256	1	8	16	1	57 600	50 688	100 352	50 688	100 352
10	14 × 14	14 × 14	256	256	1	8	16	1	57 600	50 688	100 352	50 688	100 352
11	14 × 14	14 × 14	256	256	1	8	16	1	57 600	50 688	100 352	50 688	100 352
12	14 × 14	14 × 14	256	256	1	8	16	1	57 600	50 688	100 352	50 688	100 352
13	14 × 14	7 × 7	256	512	1	8	8	1	57 600	13 056	100 352	26 112	100 352
14	7 × 7	7 × 7	512	512	1	8	16	1	32 768	26 112	100 352	26 112	100 352
15	7 × 7	1 × 1	512	1000	1	8	1	1	25 088		64 000	2000	64 000

because in most cases they are the limiting resources in CNNs. The activation layer has the same parallelization as the Batch Normalization, but a slightly lower latency. In the table: the depthwise vectorization (v_{dw}) refers to the depthwise convolution (dw_{conv}) and its Batch Normalization (dw_{bn}); the IFM vectorization (v_{ifm}) and OFM vectorization (v_{ofm}) refer to the pointwise convolution (pw_{conv}); the pointwise vectorization (v_{pw}) refers to its Batch Normalization (pw_{bn}). For module 15, dw_{conv} refers to the Pooling layer and pw_{bn} to the Softmax layer.

Table 3.27 shows the implemented design executed on the ZCU104 without an OS. A baseline software implementation that uses 32 bit floating-point numbers runs on the ARM processor at a frequency of 1.2 GHz in release mode using the O3 optimization option. The proposed implementation uses 8 bit unsigned numbers and runs on the FPGA at a frequency of 200 MHz. The time measurements were performed with the ARM processor. A good speedup has been achieved for the single modules. Module 2 has the highest speedup, since it has the highest parallelization degree and contains most functions executed in a streaming manner. For module 1 and 2 also a frequency of 300 MHz was possible. When combining all modules to a very deep pipeline this speedup would be even higher.

When comparing the FPGA computation time with the estimated time of Table 3.26, there is some overhead for streaming multiple functions in a pipeline, for data movement to and from the DDR, and for cache flushing. This overhead is 90.8 %, 76.6 % and 52.9 % for the modules 1, 2 and 15. To verify the propagation of the error, the MAPE value was computed for 16 bit unsigned fixed-point numbers. It was 0.21 %, 0.79 % and 0.78 % for the modules 1, 2 and 15. The resources listed in the table contain only the modules and no DMA blocks. Considering the available resources on the ZCU104, these should be sufficient to place all layers on it. In this case, the URAM would be needed, and the Fully Connected layer in module 15 should not buffer its weights.

Table 3.27: Proposed MobileNets modules executed separately on the ZCU104 at 200 MHz.

	Module 1	Module 2	Module 15
ARM (ms)	34.17	53.22	9.94
FPGA (ms)	0.97	0.90	0.49
Speedup	35.38	58.99	20.32
LUT	11 881	16 914	10 579
FF	13 265	16 660	5773
DSP	237	140	27
BRAM	1	20	263.5

3.3.3.3 Comparison to Related Work

Hassan et al. [264] presented a hardware/software co-design implementation of AlexNet on an FPGA. They performed the first layer of AlexNet on hardware and achieved approximately 10.7 ms when considering a frequency of 0.2 GHz. For comparison, a similar convolution layer was implemented using `HiFlipVX` with the same frequency, same parameters and 8 bit unsigned integer data types. The implemented convolution layer had a latency of 3.31 ms, which is a speedup of 3.23 in comparison to their implementation. For the same layer, `HiFlipVX` needs 73 % less BRAM, which demonstrates the proposed library's ability to reduce the memory consumption of large neural networks on FPGAs.

Liu et al. [265] developed a CNN accelerator for the XILINX Zynq-7100 SoC. They implemented the layers of the SSD-MobileNets-V1 [266] algorithm as a test application for their proposed work. Their work is also HLS-based and uses Vivado HLS 2016.4. This thesis implemented the most computationally intensive SSD-MobileNets-V1 layers and compared them with the work of Liu et al. It evaluates and executes the proposed hardware and software implementation on the ZCU102 board (3.9) using the ARM CPU for time measurements. Table 3.28 shows the CPU and FPGA results of Liu et al. and this thesis. Both implementations run at a frequency of 100 MHz to ensure a fair comparison. However, `HiFlipVX` can achieve higher frequencies.

Table 3.28: Comparison of computationally intensive SSD-MobileNets-V1 layers with related work for a parallelization of the IFM (V_{IFM}) and OFM (V_{OFM}). Results are in ms.

	Layer 1	Layer 7	Layer 27	Layer 29
ARM Cortex A9 [265]	2000.00	9000.00	5500.00	11 000.00
Accelerator [265]	10.00	30.00	55.00	110.00
ARM Cortex A53	82.89	339.41	377.65	82.64
Proposed 1 ($V_{IFM} \times V_{OFM}$)	(3 × 2) 11.38	(8 × 8) 6.08	(8 × 8) 9.52	(2 × 4) 11.45
Proposed 2 ($V_{IFM} \times V_{OFM}$)	(3 × 4) 5.95	(8 × 16) 3.38	(8 × 16) 4.92	(4 × 4) 5.89

The table shows the execution time for different parallelization settings of the IFM and OFM parameters. The configuration for "Proposal 2" represents the maximum achievable if the entire algorithm is ported to the ZCU102 and the various layers are executed in one pipeline.

Comparing the results of layer 27 and layer 29 of the SSD-MobileNets-V1 network, the execution time of this work is 11.2 times and 18.7 times faster, respectively. When computing the entire SSD-MobileNets-V1 network, layers 1 and 27 would be the bottleneck. This is intended because layers 1, 27, 29, 31, and 33 of SSD-MobileNets-V1 are the only layers with a 3×3 convolution kernel. Therefore, these layers serve as roofline model, as they consume more DSPs than the other layers.

This work proposes to use a quantization technique, such as TensorFlow post-training quantization [175]. This allows to use smaller parameters, thus saving resources and energy. This thesis uses unsigned 8 bit integers as data types for inputs, outputs, weights, and biases, to implement MobileNets. Wu et al. [267] studied the mathematical aspect of quantization parameters for various neural networks. They also present an 8 bit quantization work flow that maintains accuracy within 1 % of the floating-point baseline. Therefore, quantized parameters with smaller bit-widths should be used instead of floating-point parameters, especially for the FPGA. This maintains reasonable accuracy, achieves higher speed, and saves resources.

3.4 Summary

`HiFlipVX` contains 66 computer vision functions with 42 based on the OpenVX standard and 24 developed within this research. The various functions of the library are in the domains of image processing, feature extraction, and neural networks. One benefit of the library is that all its computer vision functions are streaming capable, which allows building a deep pipeline. Creating streaming applications with multiple nodes or layers gives FPGAs the ability to achieve higher performance and power efficiency for computer vision algorithms compared to other architectures, such as CPUs and GPUs, as shown in this thesis [9] or by Qasaimeh et al. [10]. The library is resource and performance optimized, and highly parameterizable to improve flexibility and usability. Its various compile time parameters offer multiple opportunities for an optimized design and extensive DSE. Many functions provide more options for their parameters and additional parameters not specified in the standard. For example, the support of multiple SIMD widths, data types or kernel sizes, to increase throughput and reduce latency.

The library and its components have been used or integrated in several publications. For example, in a toolchain [268] or an OS [269]. Besides the increase in performance, vectorization has further benefits. For example, it improves the ratio of resource consumption to operations, since a vectorization of eight does not increase resources by the same factor. In [258] we were able to show that vectorization in combination with DVFS does not only improve performance, but also energy efficiency. Various vendor libraries, such as AMD's AMDOVX or NVIDIA's VisionWorks partially follow the OpenVX standard. Therefore, using C++ for HLS with OpenVX eases the cross-platform development between different architectures and vendors. `HiFlipVX` does not require any external or vendor libraries and runs on any CPU that has a C++ compiler. This eases the integration into existing projects, the usage of different devices and the portability to other vendors. While using XILINX devices to optimize the library, we could show vendor independence by porting parts of it to Intel FPGAs [15, 16]. `HiFlipVX` only contains directives for XILINX-based HLS tools like SDAccel [214], SDSoC [213] or Vivado HLS [225]. The only difference when using these tools are the interface directives, which are automatically set in the `DECISION` framework.

The image processing functions were classified into pixelwise, filter, conversion, and analysis functions, and a common structure was provided [12]. The latency analysis of these functions also helps in building the timing model for the `DECISION` framework when implementing more complex algorithms such as ORB or AKAZE. The complexity analysis of the analysis functions, which differ in their timing model from most other image processing functions, shows this. Based on a default configuration, pixelwise, filter, conversion, analysis, and most feature functions were evaluated separately. Thereby, various library parameters were adapted and resource consumption, scalability and latency were observed. Based on the results, different design choices were described. For example, where to use separable filters, as for the Sobel, based on the observation of resources. Furthermore, the advantages and disadvantages of different options were described, as shown for the derivatives in the DoH.

The library functions can also reach high frequencies. For example, all filter functions require only 43.2 % more FFs and 5.1 % more LUTs for a frequency increase from 100 MHz to 300 MHz on the ZCU104. `HiFlipVX` consumes on average only 0.39 % of FFs and 0.32 % of LUTs for a set of functions compared to the xfOpenCV library from XILINX. The difference is more pronounced for larger kernel sizes and is due to the various optimizations, such as using separable filters, exploiting kernel coefficients, and providing an integer square root. In general, the library takes care of optimizing the use of BRAM considering the available bandwidth to reduce fragmentation. On average, `HiFlipVX` consumes 1.42 times less BRAM than xfOpenCV for the selected filter functions.

A unique feature of `HiFlipVX` are the feature type functions, which were extracted from the numerous algorithms [13, 23]. Beforehand, all possible combinations of feature detection and description algorithm were compared in software regarding their performance and repeatability using optimized parameters. An implementation optimized in this work, which relies on the AKAZE detector and FREAK descriptor, achieved an inliers ratio (repeatability) of 72.57 %, while the next best combination only achieves 62.99 %, when using the geometric mean, which gives a stronger weight on the more complex cases. Compared to the original AKAZE-FREAK implementation, the proposed optimizations reduce the computation time by a factor of 2.82 while detecting 18.5 % more features. Since the input feature vectors of the AKAZE compare function arrive sorted by their coordinates, the proposed implementation reduces its complexity from $\mathcal{O}(n^2)$ to approximately $\mathcal{O}(n \cdot \log_2 b)$ for n features and a buffer size of b elements. The ORB [26], AKAZE [25] and FREAK [22] were implemented in a resource and performance optimized design using VHDL, which is ideally for embedded systems. These implementations were used to design and implement generic `HiFlipVX` functions and algorithms not included in the OpenVX standard. Comparing the repeatability of the optimized hardware designs of ORB, AKAZE and FREAK with those of the other software combinations, they are still better on average, despite the use of fixed-point numbers. Using `HiFlipVX` and the `DECISION` framework, it is possible to implement and test designs with little effort, without sacrificing repeatability or performance compared to VHDL designs.

The ORB algorithm is ideally suited for smaller embedded systems, due to its low computational overhead, low resource consumption and good repeatability. Using the FREAK and an extensive examination of various parameters, the ORB could also improve its repeatability by 7.67 % in software. The retain best function makes the computation time of the detector and descriptor predictable and reduces the WCET, while improving repeatability, especially for more complex problems. A key feature of the proposed hardware implementation of the FREAK is the pattern generator since patterns are partially precomputed and partially computed. This reduces the required bandwidth of the intensity calculation by a factor of three with minimal resource overhead. This allows it to write significantly more features per second than achieved by other researchers. The inliers ratio decreases from 78.7 % to 77.0 %, while the average inliers ratio only decreases from 91.4 % to 90.9 %, in comparison to the software implementation. The AKAZE algorithm requires more computation and FPGA resources than others but can outperform all other algorithms in terms of repeatability. The `HiFlipVX` implementation of AKAZE reaches 480 fps for a 1920 × 1080 resolution and computes between 3.56 and 4.13 times more PPS (Pixels Per Second) than achieved by other researchers on an FPGA. Thereby, the overhead for all buffers, pipelines, sliding windows and memory access for the execution of 95 compute nodes is just 13.5 % if computing four pixels per clock cycle is the baseline. At the same time, the resource consumption is comparable to that of optimized VHDL designs. This proves the resource efficiency and performance of `HiFlipVX` on the one hand, and the capabilities of HLS in general on the other.

`HiFlipVX` contains seven streaming capable neural network layers to create large designs [14]. The evaluation shows the low error rate, high performance, scalability, and resource efficiency of the library. Compute intensive layers, such as 3D convolution, get an additional parallelization option to maximize performance while optimizing bandwidth and thus using resources efficiently. Using MobileNets as an example, the evaluation showed how to tune these parameters for a larger design and how to optimally load coefficients into the design. By adding a multi-level pipelining approach, batch normalization can compute the three internal functions in almost the same time as the activation layer for the same parallelization degree. The comparison with related work shows speedups of up to 18.7 for individual layers of the MobileNets algorithm. At the same time, an AlexNet layer achieved a speedup of 3.23 in comparison to a related work, while consuming 73 % less BRAM. The next part of this thesis evaluates the `DECISION` framework, which integrates the `HiFlipVX` library and contains two toolchains.

Thanks to the high parameterization of the individual functions, optimized designs for embedded or HPC systems can be developed. To automate the design for such systems and to address different architectures, the `DECISION` framework, described in the next chapter, was developed. Its frontend is based on OpenVX and integrates the `HiFlipVX` library. It hides implementation details from the user and automatically generates IP-cores and a graph representation from the application.

4 `DECISION`: Vision Framework

The goal of this thesis is the application distribution and efficient programming of object detection algorithms on FPGA-based heterogeneous systems. Two of the core contributions are the `APARMAP` application distribution algorithm and the `HiFlipVX` object detection library. A third core contributions is the simple and efficient programming of heterogeneous FPGA-based systems. To achieve this, the `DECISION` framework was designed.

This framework consists of two toolchains that share the same frontend to implement computer vision applications. This frontend is based on the OpenVX standard and abstracts away the implementation and hardware details from the developer. The High-Performance Vision toolchain targets x86-based HPC systems, which can consist of CPUs, GPUs and FPGAs. It maps the application to devices, creates a heterogeneous schedule using profiles and estimates, creates a memory model, and handles synchronization to create a performance-optimized runtime system at design-time. The Embedded System Vision toolchain targets partition-based mesh-like FPGA topologies. It uses a NoC as the communication infrastructure to interconnect PRRs to generate a scalable, adaptable and flexible architecture. It uses the `APARMAP` algorithm, described in the next chapter, to distribute applications and create an application-specific hardware architecture. Both toolchains use the `HiFlipVX` object detection library to target FPGAs.

Section 4.1 will give an overview of the components of the `DECISION` framework and their usage in the two toolchains. In Section 4.2 the common frontend of both toolchains will be described. This work uses OpenCL to address non-FPGA devices due to its high availability. Section 4.3 will examine the architecture dependent OpenCL kernel optimization for CPU, GPU and FPGA devices. Section 4.4 will investigate the automatic OpenCL kernel code generation using source-to-source compilers. Section 4.5 will describe the High-Performance Vision toolchain and Section 4.6 the Embedded System Vision toolchain. Section 4.7 will evaluate both toolchains, and the optimization of OpenCL kernel and their automatic generation. The last Section will give a summary of this chapter.

4.1 Overview

The two toolchains of the `DECISION` framework consist of various components [27]. In addition to the various components, a robust and efficient framework requires certain characteristics. Two of these characteristics are the modularization of components and abstraction of models. The modularization of a framework leads to a better reusability and simpler exchangeability of the different components. Regardless of the type of toolchain, a first rough partitioning almost always consists of a frontend, a backend and a middleend.

Figure 4.1 shows a simplified structure of the proposed framework, which consists of different components that form the two toolchains. The models serve as interfaces between the modules and are explained in more detail together with the toolchains. The figure shows the order of execution of the different modules and by which toolchains they are used.

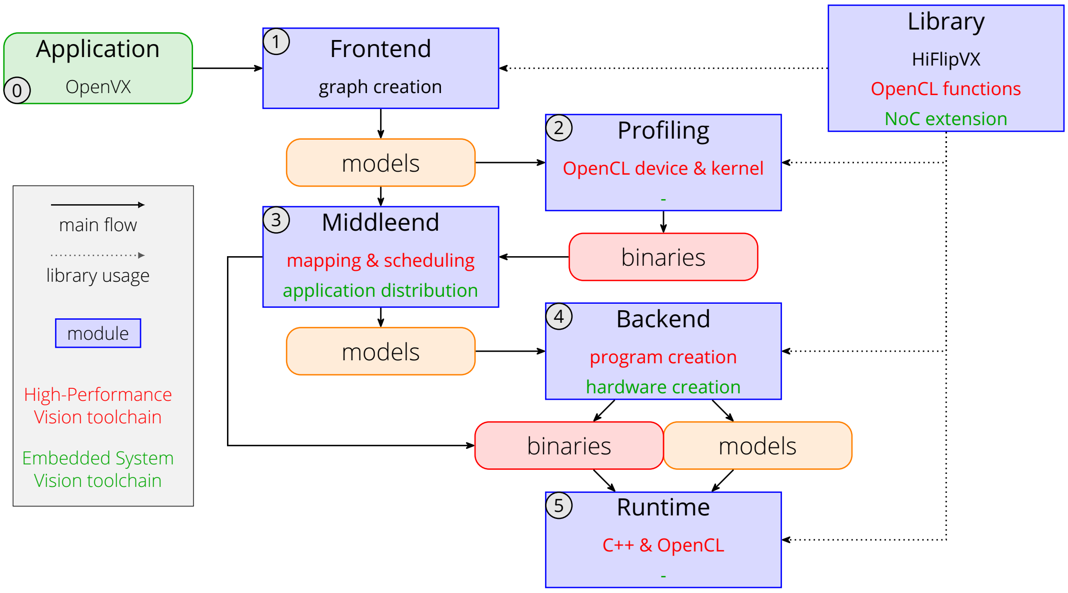


Figure 4.1: Overview and flow of the `DECISION` framework, which includes the High-Performance Vision toolchain and Embedded System Vision toolchain.

Frontend: OpenVX is used as a common frontend to implement computer vision applications and to build a model from the input application and verify that it is valid. The application model is expressed as a graph in the form of a DAG. One advantage of this abstraction is that regardless of the target platform, the user does not have to learn any new concepts or input languages. Another advantage of this approach is that the user does not have to deal with the underlying hardware architecture. The frontend is realized in the OpenVX Graph Creation module, which will be presented in Section 4.2.

Library: The main part of the Library module is `HiFlipVX`. For each of its functions there is an entry in the frontend so that the user can call it. However, the OpenVX standard makes it possible to integrate libraries from different vendors or developers, to address other architectures. The High-Performance Vision toolchain examined and added different OpenCL-based libraries, to also target GPU and CPU devices in x86 architectures. The Embedded System Vision toolchain needs additional components, to build an adaptable and flexible NoC-based architecture.

Since a library which consists of a set of functions cannot provide everything, an additional method is required to create and integrate custom accelerators. In Section 4.4 the User Function Creation module, which supports the generation of custom functions using OpenCL and C/C++, is described [28]. Implementing custom OpenCL kernels is not trivial and can differ greatly between different architectures. In Section 4.3, this difference is described for CPU, GPU, and FPGA devices, to help implementing own optimized vision functions [9].

Middleend: In the middleend various transformations are processed on the application graph. These transformations refer to the different subareas of the application distribution process, which was presented in Section 2.3.6. Which parts of the described subareas are required depends on the respective target architecture and its flexibility. The High-Performance Vision toolchain implements a heterogeneous mapping and scheduling algorithm to distribute applications to GPUs, CPUs, and FPGAs in x86-based architectures. The Embedded System Vision toolchain uses the `APARMAP` algorithm to schedule and map the application graph to physical nodes, and then cluster and place those nodes on the PRRs of the NoC-based architecture.

Profiling: In addition to the application model, a platform model is needed to describe the topology of the architecture and its components. For an optimal application distribution, the application and platform models must be annotated with information regarding resource consumption, computation time and communication costs. For the FPGA, this information is obtained from the synthesis estimates generated by the OpenVX Graph Creation module. The High-Performance Vision toolchain implements a Profiling module to profile OpenCL kernels and devices on CPU and GPU architectures.

Backend: The final program and architecture are created in the backend based on the results of the application distribution process. As intermediate language C++ and OpenCL are used to create the respective binaries with the help of the vendor tools. For the High-Performance Vision toolchain, the Program Creation module is used, to create a runtime optimized OpenCL program. For the Embedded System Vision toolchain, the Hardware Creation module is used, to create a runtime adaptive hardware architecture.

Runtime: Even though this work mainly deals with design time optimizations, a minimal runtime system is needed to run the final program. For the High-Performance Vision toolchain, an OpenCL-based runtime system was realized. The corresponding Runtime System module excels by its high parallelism and a low overhead. For the Embedded System Vision toolchain, a runtime adaptive NoC-based architecture, which is created in the Hardware Creation module, was realized.

4.2 OpenVX Graph Creation Module

This work exploits the OpenVX standard as a common frontend to address different architectures. This has resulted in the OpenVX Graph Creation module, which allows the user to implement a computer vision application without knowledge about the underlying hardware. This module is used in the High-Performance Vision toolchain to target x86-based systems consisting of CPUs, GPUs, and FPGAs. It is also used in the Embedded System Vision toolchain to address pure or ARM-based FPGA systems. To change the target architecture only a flag needs to be changed. The module integrates the `HiFlipVX` library to address FPGA-based architectures.

The main goal of the OpenVX Graph Creation module is to create an application graph and generate the required meta-data for its nodes (tasks) and edges (transactions). Furthermore, IP-cores are generated using the `HiFlipVX` library functions. The next subsection explains the OpenVX Graph Creation module and its flow, using a simple example application. The subsequent subsections deal with the generation of the IP-cores and models for both toolchains.

4.2.1 OpenVX Application Flow

Listing 4.1 shows a small code example of an edge detection algorithm implemented with the OpenVX Graph Creation module. The developer implements an algorithm in C++ and inserts images (edges) and nodes (vision functions) into the OpenVX context and graph. The `vx_context` contains the overall structure of the implemented OpenVX program. It stores all existing references to objects and is responsible for garbage collection. It can contain one or more `vx_graph` objects, each expressing the data flow of an application.

```
// Context and graphs
vx_context context = vxCreateContext();
vx_graph graph = vxCreateGraph(context);

// Images (edges). Virtual image info is set during verification
vx_image images[] = {
  vxCreateImage(context, 1920, 1080, VX_DF_IMAGE_U8),
  vxCreateImage(context, 1920, 1080, VX_DF_IMAGE_S8)};
vx_image virts[] = {
  vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
  vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
  vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
  vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT)};

// Vision functions (nodes)
vxGaussian3x3Node(graph, images[0], virts[0]);
vxSobel3x3Node(graph, virts[0], virts[1], virts[2]);
vxMagnitudeNode(graph, virts[1], virts[2], virts[3]);
vxHysteresisNode(graph, virts[3], images[1]);

// Verify and process graph
vx_status status = vxVerifyGraph(graph);
if (status == VX_SUCCESS)
  status = vxScheduleGraph(graph);
```

Listing 4.1: OpenVX example code for an edge detector.

Nodes and Images

A `vx_node` object is an instantiation of a computer vision function using its standard parameters. When creating the node, the specified parameters are automatically verified. For all other parameters default values are set. These parameters can be changed individually before the graph is executed. Additional parameters for the `HiFlipVX` functions which are not part of the standard, or additional object detection functions, have been incorporated. Apart from that, the implementation is compliant with the OpenVX standard.

The `vx_image` object stores the image information, such as the width, height, format, and vector size. The vector (SIMD) size can be seen as an additional hint for the compiler. It is not an additional dimension, as it would be the case for a tensor. When changing the vector size or format, the resolution does not need to be adjusted by the user as this is managed internally along with other properties. For example, an `RGB` image format with 24 bit data type is internally interleaved to 32 bit data types (`RGBR|GBRG|BRGB`). Virtual images must have a source and do not need to specify their parameters. They are edges between nodes and can only be written by nodes, which prevents an indeterminate order of data dependencies.

Vision functions that consist of more than one function are internally split into their subfunctions. Each of these subfunctions is added as a node to the graph. If necessary, additional (virtual) images are created. These functions are, for example, the feature detection algorithms: FAST corners, Canny edge, AKAZE and ORB. In the `HiFlipVX` library these functions also exist as single IP-cores, which use the `dataflow` and `stream` directives to connect the subfunctions.

The breakdown into separate functions has several advantages. It enables a more fine-granular application distribution within the subsequent modules of the middleend. This reduces fragmentation, when placing the IP-cores into PRRs. It also reduces the synthesis time, as the different IP-cores can be generated in parallel. Furthermore, an exponential increase in synthesis time was observed with an increasing number of functions when using the `dataflow` directive. It was also observed that in some cases the achieved latencies deteriorated when using the `dataflow` directive. On the other hand, a slight improvement in the utilized resources was observed when using the `dataflow` directive, since each IP-core requires an interface. However, this advantage is minimized by the optimization phase during bitstream generation.

Graph Verification

The `vxVerifyGraph` function creates a graph and verifies if the constructed graph satisfies the formalism defined by the OpenVX 1.2.1 standard. The graph must be a DAG and bipartite, but not all nodes must be connected. It must not contain cycles, and each virtual image used in the graph must be preceded by nodes with a real image as input. The verification process checks various attributes required by the computer vision functions. This ensures that the functions call images with correct sizes and data types, but also that the attributes of the function output are applied to virtual or real images.

When iterating over the nodes of the graph, their parameters are checked or set depending on the kernel function. First, its input parameters are verified. Certain kernels only allow a specific resolution, format, or vector size. Additionally, it is verified that the different input parameters of a kernel are compatible to each other. Next, the output parameters are either verified or set if they are virtual. Resolution, vector size or format are set based on the provided parameters. For example, the data-width converter changes the vector size based on a mandatory function parameter. Another example is that the resolution may change according to the step size of the filter functions. However, there are functions that do not allow virtual output parameters. For example, the resolution of the scale function or the format of the bit-width and color conversion functions need to be specified.

Graph Scheduling

The application of a graph can be scheduled using the `vxScheduleGraph` function after successful verification. However, due to the nature of FPGAs and the design time optimizations of this work, there are some deviations in how this function has been implemented internally. As input, a simple configuration file is required, which contains the FPGA part number and the targeted clock period to synthesize the IP-cores of the vision functions. In the `vxScheduleGraph` function an IP-core is generated for each node. Depending on the target architecture, the different models for the subsequent modules of the two frameworks are generated and stored in files. For this purpose, the information from the graph and the generated synthesis results are used. In the following, the creation of the IP-cores and extraction of the data for the models will be discussed. The models will be described along with all other models in the sections of the respective frameworks.

4.2.2 Data Extraction

Depending on the target hardware, different output information is needed. One reason for this is the use of OpenCL in the runtime system of the High-Performance Vision toolchain. On the one hand, OpenCL needs additional kernel information. On the other hand, the integration of IP-cores into the hardware design is done by the vendor toolchain, which requires less information.

Embedded System Vision toolchain

The output for the Embedded System Vision toolchain is the application model, which consists of tasks and transactions. Its graph will be scheduled and mapped to physical nodes, which are clustered and placed to a platform model, within APARMAP to create a runtime-adaptive architecture. It is possible to bypass APARMAP to directly map each task to a node and transaction to an edge and create a pure AC-based design. In this case a simplified architectural model is generated, which can be used by the Hardware Creation module. All the mentioned models will be described in more detail in Section 4.6.2.

Timing Calculation Prior to the model generation, some information needs to be extracted from the synthesis results to calculate or estimate the various parameters of the model. One of these parameters is the task latency (L_{task}), which is the sum of all non-concurrent loops of a task. The latency of each loop can be calculated by its trip count (T_C), pipeline interval (P_I) and pipeline depth (P_D). Before executing a pipelined loop, an initial clock cycle is required for control.

$$L_{task} \approx \sum_{i=0}^{i<N} \left(T_{C_i} \cdot P_{I_i} + \left(P_{D_i} - 1\right) + 1\right) \tag{4.1}$$

For the calculation of other parameters, the functions of the `HiFlipVX` library must be examined more closely. All functions of the library are streaming-capable, which leads to several simplifications. Some functions, like the histogram, consist of more than one loop, which need to be executed in a sequential manner. The trip count of a single loop can be calculated from the image columns (I_C), image rows (I_R), kernel size (K_S) and vector size (V_S).

$$T_{C_i} \approx \left(I_{R_i} + K_{R_i}\right) \cdot \left(\frac{I_{C_i}}{V_{S_i}} + \left\lceil \frac{K_{R_i}}{V_{S_i}} \right\rceil\right) \tag{4.2}$$

$$K_R = \left\lfloor \frac{(K_S - 1) \cdot \sigma + 1}{2} \right\rfloor \tag{4.3}$$

The kernel radius can be calculated from the scaled (σ) kernel size (K_S). The memory access pattern of the library functions is quite similar. Reading the input starts at the beginning of a pipeline and writing the output at the end of a pipeline. In addition, the start of the write process is delayed by the time it needs to initially fill the line buffers and sliding window. Based on Equation (4.1) and Equation (4.2), the latency of a transaction, and its offset compared to the start of the associated tasks, can be calculated.

$$WriteOff_j \approx \sum_{i=0}^{i<j} \left(T_{C_i} \cdot P_{I_i} + P_{D_i}\right) + K_{R_j} \cdot \left(\frac{I_{C_j}}{V_{S_j}} + \left\lceil \frac{K_{R_j}}{V_{S_j}} \right\rceil\right) \cdot P_{I_j} + P_{D_j} \tag{4.4}$$

$$ReadOff_j \approx \sum_{i=0}^{i<j} \left(T_{C_i} \cdot P_{I_i} + P_{D_i}\right) + 1 \tag{4.5}$$

$$L_{trans_j} \approx I_{R_i} \cdot \left(\frac{I_{C_i}}{V_{S_i}} + \left\lceil \frac{K_{R_j}}{V_{S_j}} \right\rceil\right) \cdot P_{I_i} \tag{4.6}$$

There are two types of functions that have an additional influence on transaction latency and offset. The scatter functions influence the sender, and the gather functions the receiver. In block mode the latency is divided by the scatter/gather factor (L_{trans_j}/SG). The associated offset is shifted depending on the index (sg) of the function (($L_{trans_j}/SG) \cdot sg$). In cyclic mode, the adjustments are minor. The latency is reduced by ($sg - 1$) and the offset is shifted by ($sg - 1$).

Latencies and offsets of transactions and tasks are needed by middleend for a more precise time behavior. For example, it is needed for the calculation of the bandwidth usage. The estimation of the different latencies is very accurate if the functions are not memory bound. The consideration, whether a function is memory bound, is made in the middleend, since a statement concerning this problem can only be made when considering the complete application and architecture. All further parameters do not need any further preliminary calculations and are described together with the application model.

Buffers and DMAs Additionally, it is possible to skip the middleend and create an architecture model to generate a pure AC design. However, for this the minimum size of the buffers between the ACs (nodes) must be calculated. These buffers are needed to avoid deadlocks and to achieve an optimal data throughput. For this purpose, an ASAP (As-Soon-As-Possible) schedule is created using the calculated latencies and offsets to get the time stamps of the transactions. The transactions between tasks are synchronized with each other to update their timestamps.

The buffer size results from the number of elements of a transaction, which are written by the sender before the receiver starts reading. This imbalance can occur, for example, when a node operates on two input images, but one image arrives much earlier. Figure 4.2 illustrates this problem using a small part of the AKAZE algorithm. Because of the line buffers in the filter functions, there is a delay between the inputs of the FED (Fast Explicit Diffusion) function. Therefore, a buffer is needed for the second input.

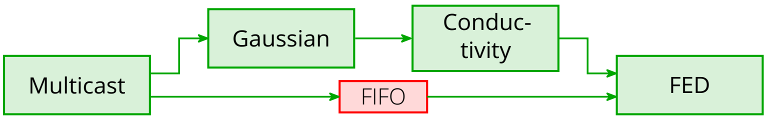


Figure 4.2: Small application graph illustrating the buffering problem.

In the described case, a deadlock would occur without a larger buffer. In many other cases, a larger buffer would not improve performance and would only consume additional resources. The synchronization between two nodes can, however, have an influence on the start times of transactions of specific tasks and thus on the buffer sizes. This includes gather and scatter tasks and tasks that consist of more than one loop. These are mainly functions of the analysis class: histogram, contrast factor, equalized histogram, table lookup, retain best, mean and standard deviation, and min-max location.

Due to the fixed buffer sizes of the XILINX IP cores, the number of calculated elements is scaled to a multiple of two. The minimum size is 16 and the maximum 32 768. For edges with no successor or predecessor, a DMA is created and connected. The associated buffer of the DMA can be set to a default size large enough for bursts transfers (e.g., 32 elements).

Synthesis For creating synthesis results, a wrapper function is created to instantiate the `HiFlipVX` function, as shown in Listing 4.2. This wrapper function sets the template parameters and adds directives to create an AXI4-stream interface (`axis`). It is a simple handshaking interface to stream data between different functions. This thesis only uses its valid (1 bit), ready (1 bit), last (1 bit) and data signals (n bit). In the final hardware design, the global memory will be accessed through DMA units. Therefore, no control port is needed (`ap_ctrl_none port = return`).

```
void Gaussian0(vx_image_data<vx_uint8, 1> p0[2073600],
               vx_image_data<vx_uint8, 1> p1[2073600]) {
#pragma HLS interface ap_ctrl_none port=return
#pragma HLS interface axis port=p0
#pragma HLS interface axis port=p1

  const vx_border_e border = static_cast<vx_border_e>(49153);
  const vx_bool separable = static_cast<vx_bool>(1);
  ImgGaussian<vx_uint8, 1, 1920, 1080, 5, border, separable>(p0, p1);
}
```

Listing 4.2: Instantiation of a `HiFlipVX`-based accelerator.

The IP-cores of the different functions are synthesized in parallel using a TCL (Tool command language) script. This script uses Vivado HLS internally and reads the configuration file to set the required parameters such as the target platform, clock frequency, and optimization levels. An IP-core is generated for each function and can be reused by the later toolchains. For example, the Hardware Creation module of the Embedded System Vision toolchain incorporates them into the final hardware design. The parallelism and fine-grained scheduling are done using OpenMP directives (`for schedule(dynamic, 1)`).

High-Performance Vision toolchain

In this work, the SDAccel tool from XILINX is used to create FPGA kernels (ACs) for the High-Performance Vision toolchain. Even though SDAccel uses OpenCL for the host code, there are different languages to implement a kernel. For example, a kernel can be written in C++, which enables the usage of template-based libraries like `HiFlipVX`.

In comparison to the Embedded System Vision toolchain, there are several changes to the wrapper function of the kernel that need to be applied. This is mainly because SDAccel automates the creation of the hardware design needed for kernel execution. The following describes the realization of the C++ wrapper for a `HiFlipVX`-based AC, to allow an optimal usage of the memory bandwidth and fully utilize the available performance of the kernel.

As the function requires direct access to memory and is not controlled via an additional DMA, AXI4 interfaces are required (`m_axi`). To increase the performance one interface per port should be used and only one input and one output port should be combined in one bundle. The drawback of this approach is an increased resource consumption of the final system.

To set the different kernel parameters by the host an additional AXI4-lite interface is needed (`s_axilite`). The same interface is used to set all parameters, by collecting them in one bundle. In contrast to AXI4, AXI4-lite is a simplified interface that cannot do burst transfers.

To utilize the full bandwidth, the `m_axi` interfaces can be up to 512 bit wide. However, this can cause a bit-width mismatch when connecting it to the vision function. Therefore, additional data-width converters are generated in the kernel for each input and output. These consist of nested loops and are also part of the `HiFlipVX` library.

Different functions are parallelized using the dataflow directive and buffers are generated out of intermediate images using the `stream` directive. Since accessing global memory from the kernel can have a high latency, it should be done in bursts. These bursts are also enabled by using the data-width converters. The additional converters and wider interfaces also lead to an increased resource consumption.

Two outputs are generated for the High-Performance Vision toolchain. These are the application model and its extended version, which already contains the mapping of tasks to compute devices. The first one is sent to the Mapping & Scheduling module and the second one is sent to the Program Creation module. The latter one can be used to skip the Mapping & Scheduling module or to change the mapping manually.

Both models will be described in more detail together with the High-Performance Vision toolchain. Most of their required data can be extracted from the OpenVX graph. Only the extended application model requires further information that needs to be extracted. For this, it is searched for all OpenCL-capable devices within the system and OpenCL/C++ based kernels within the library. Furthermore, a default device is selected, based on its setting in the configuration file. A target device of a node can also be selected in the OpenVX application.

4.3 Architecture Dependent OpenCL Kernel Optimizations

OpenCL allows the use of different architectures from different vendors using the same API. It provides code portability, but the kernel code still needs to be optimized specifically for the various architectures. The previous section presented the OpenVX-based frontend. It integrates the `HiFlipVX` library to provide optimized kernels for FPGAs. However, kernels implemented and optimized by the user can also be integrated. Thus, this section describes different optimization strategies to implement custom kernels.

This sections compares the different optimizations for different architectures, such as CPUs, integrated GPUs, GPUs, and FPGAs, using OpenCL. This thesis also investigated the capabilities of the SDAccel [214] tool for streaming applications on XILINX FPGAs. A part of the AKAZE [46, 20] algorithm has been chosen as use case, since it shows better repeatability than other algorithms. The implemented design, shown in Figure 4.3 consists of two parts: nonlinear scale-space creation (left) and feature detection (right). To have a fair comparison, different optimization strategies have been investigated and implemented for the observed OpenCL devices. More general information of the original algorithm can be found in Section 2.1. The proposed and optimized AKAZE implementation of this thesis can be found in Section 3.2.1.

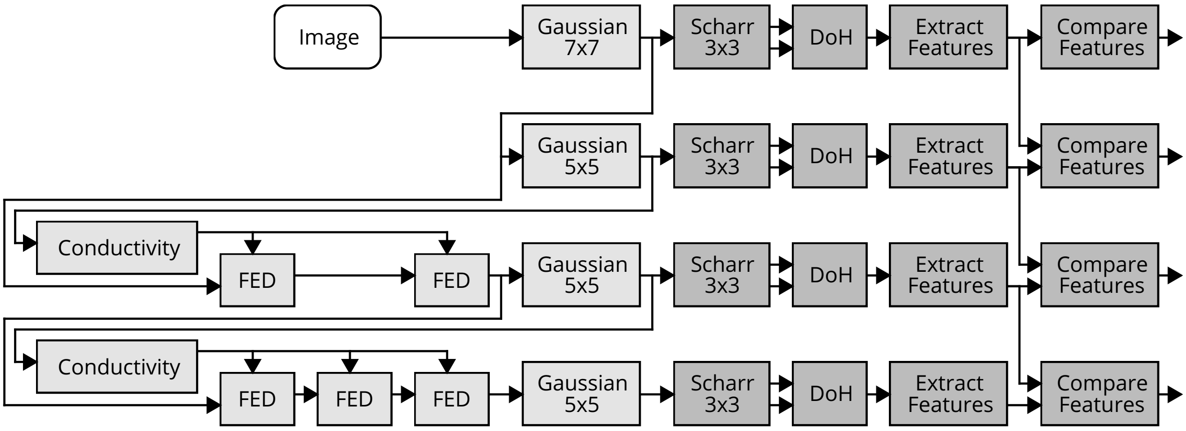


Figure 4.3: Task graph of the AKAZE feature detection implementation. FED (Fast Explicit Diffusion), DoH (Determinant of the Hessian)

The streaming capability and maximum bandwidth consumption of the FPGA were explored using the SDAccel tool. In OpenCL, streaming between kernels is realized using pipes, which is part of the 2.0 specification. The connected kernels can be executed in parallel on FPGAs, since OpenCL gives the opportunity of out-of-order execution. In comparison to other HLS models that use C/C++, SDAccel needs nearly no extensions or directives for the FPGA code, due to the rich set of instructions of OpenCL. Even loop unrolling is part of the 2.0 specification. Only one XILINX specific attribute for loop pipelining was needed in the implementation of this study.

Using the OpenCL drivers of different vendors, allows the execution of different devices within a single program using only one programming model and one toolchain. This eases the distribution of an application between several devices. The FPGA kernels are precompiled using the XILINX OpenCL compiler. This compiler can be executed using a Makefile or a TCL script. For this work the second option has been chosen and the other vendors are linked within the same TCL script. With this approach one can also integrate OpenCL devices of other vendors into the SDAccel tool. Therefore, kernels of all devices can be executed in parallel within one application.

4.3.1 FPGA Bandwidth and Kernel Optimization

Since kernels are typically memory or compute bound, it is important for the programmer to determine the limitations, when optimizing an algorithm. One approach for FPGA kernel design is to first determine the maximum achievable bandwidth by removing all code not related to memory access, and then increase the computation speed with respect to the maximum bandwidth. In the following, various optimization strategies for windowed OpenCL functions are described in chronological order.

- Baseline: The baseline OpenCL implementation is a partially optimized convolution filter, with a work-group and work-item size of one. This function has loops for *x* and *y* direction and a body for the computation of the filter.
- Loop pipelining: The first optimization step is loop pipelining, which is provided by SDAccel and applied to the inner loop. Its compiler requires that all kernel code, which is between the two loops, must be moved to the inner loop.
- Line buffers: Line buffers are used to reduce multiple global memory accesses to the same data. Using line buffers data only needs to be prefetched once into local memory (BRAM). If each line buffer row is stored in an own BRAM, a 7×7 kernel needs four clock cycles to read data, since each BRAM only has two data access ports.
- Array Partitioning: By partitioning each line buffer into multiple BRAM in a cyclic approach, the number of clock cycles for data access can be reduced with an increase in BRAM usage.
- Sliding window: A better solution is to use registers for the convolution kernel and combine them with line buffers in a sliding window as shown in Figure 4.4 for a 3×3 kernel. With this approach, it is possible that the pipeline processes one pixel per clock cycle. There is only a small overhead in computation time to fill the sliding window with data, which depends on the kernel radius. A big advantage of the sliding window approach is that the computation time hardly depends on the kernel size of the convolution filter in comparison to CPU or GPU implementations.
- Separate memory access kernels: To optimize parallelism of computation and memory access, separate kernels are used for reading, writing and computation. These kernels are connected via FIFO blocks using OpenCL pipes and executed concurrently, since OpenCL provides out-of-order execution. The computation speed of the FPGA implementation is now comparable to the speed of a single threaded and optimized CPU implementation. The difference is that the FPGA is faster for larger kernels, due to the sliding window approach. If the amount of memory buffers that are accessed in parallel increases, it is recommended to increase the burst size of the memory operations, if possible, to decrease the switching amount in the DDR memory.
- Vector operations: Vector operations can be used for computations and for memory operations, to further increase the memory throughput and the computation speed. OpenCL provides a maximum vectorization of 16. The optimal vectorization depends on the maximum achievable bandwidth or the available resources.

- **Fixed-point numbers**: While GPUs are very good for floating-point computations, FPGAs have their strengths using fixed-point numbers, due to the reduced resource utilization. Therefore, all filter calculations have been normalized and reduced from 32 bit floating-point data to 16 bit fixed-point data, which also reduces the utilized memory bandwidth. To reduce the loss of accuracy caused by using fixed-point numbers, higher bit-widths are used inside of a kernel. By halving the bit-width, the function requires only half the bandwidth. This leads to an almost twice as high computational speed if there are enough resources for a higher vectorization.
- **Data Packing**: Since the 512 bit memory port of the FPGA is not fully utilized using OpenCLs maximum vector size of 16 and 16 bit fixed-point numbers, memory is accessed with a 32 bit data type containing two 16 bit fixed-point numbers.
- **Data Streaming**: Data only needed by subsequent kernels is streamed between them to avoid global memory access and reduce latency. This streaming capability also compensates the lower global memory bandwidth of most FPGAs compared to equivalent GPUs.

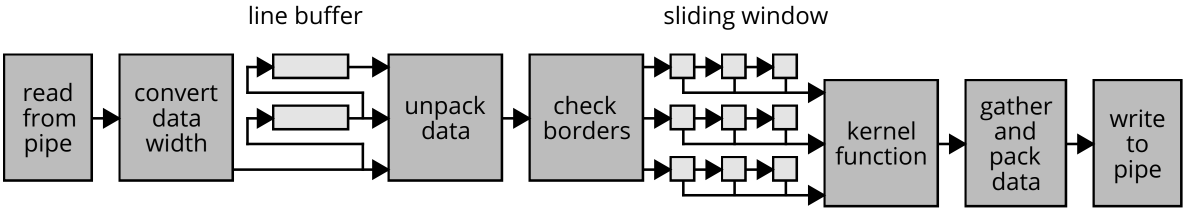


Figure 4.4: Windowed (3 × 3) function implementation on an FPGA.

4.3.2 FPGA Example Implementation

Figure 4.3 shows the implemented algorithm of this study. The input image is an 8 bit grayscale image, and each feature extraction function writes a feature vector to the output. The chosen vector size and bit-width is 4 times 16 bit for computations and 16 times 32 bit for memory access. The vector size for the computation is lower, since the implementation got memory bound. Therefore, the vector size has been reduced to lower the resource consumption. In general, if the usable memory bandwidth, the achieved frequency, and the overhead to fill the sliding window is known, it is possible to calculate the desired vector size beforehand. All interim results are streamed between the different kernels. Some functions needed to be combined in one kernel, since SDAccel 2016.1 only allows a maximum of ten CUs. Loop fusion was used to combine functions since loop-level parallelism is not possible within a single kernel in SDAccel's OpenCL implementation. There is one unit for global memory access, five for nonlinear scale-space and four for detection. The feature extraction function outputs maximum two features in one clock cycle, because there can only be a maximum of one extreme value for two neighboring pixels. This work uses a shared kernel for reading and writing, since the output is relatively small.

Figure 4.4 shows the different steps of the implemented windowed functions, which have a similar structure. First data is read vector by vector from the input pipes. If a pipe is connected to a read or write kernel, the input is one 16 times 32 bit vector and scattered into eight 2 times 32 bit vectors. These vectors are written one after the other into line buffers, where data remains packed to reduce BRAM usage. Then data is unpacked from a 2 times 32 bit to a 4 times 16 bit vector. The example implementation duplicates border pixels if the window is outside of the image boundaries. Duplicating the borders achieved the best results in terms of repeatability for the feature detection algorithm. The sliding window moves one vector in each clock cycle from one register to the other and provides parallel access to the complete window. If results are written to DDR memory, eight 4 times 16 bit vectors must be gathered and packed to a 16 times 32 bit vector. Otherwise, data is packed to a 2 times 32 bit vector to reduce the BRAM utilization of the pipes.

4.3.3 CPU and (integrated) GPU Kernel Optimization

This subsection will describe different OpenCL optimization strategies for CPUs, integrated GPUs, and GPUs. Four different strategies have been implemented for a fair comparison to the FPGA implementation.

OPT0: To create fast and simple convolution filters, different device independent optimizations have been applied, like using built-in functions and compiler optimizations. The number of computations within the OpenCL kernel has been reduced and branches are replaced by built-in functions (e.g., `min`, `max` and `clamp` operations). Kernel parameter qualifiers like `restrict`, `const`, `read_only` and `write_only` are used to optimize memory access. But no local memories, barriers or vector operations have been used.

OPT1: In this strategy, various CPU-based optimizations are applied, which make use of Intel's OpenCL optimization guide [270] as well as own methods. An advantage of the OpenCL vector instructions is the possibility of using Intels SIMD operations without using their intrinsic instructions. Since CPUs generally use caches as local memory, there is no need of exploiting optimization strategies for utilizing the OpenCL local memory. Data can be buffered directly within the registers, and in case of register spilling, the L1 cache is used automatically. Figure 4.5 illustrates the implementation using this type of buffer. For a 3 × 3 convolution filter, the first two rows are loaded into registers, taking the image boundaries into account. Then, within a loop the next row is loaded, the output vector is computed, and the observed window is updated, by moving the pointers down one row. Every work-item processes an 8 × 8 grid of pixels, using a vectorization of eight and processing eight rows in a sliding window approach, which achieved the best results. The work-group size is left to the compiler.

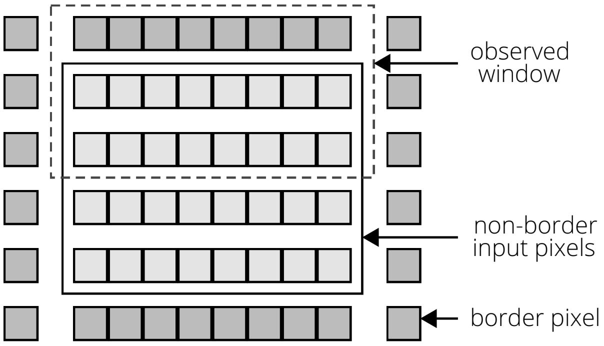


Figure 4.5: Example for the input pixels of a register buffer for an 8 × 4 grid.

OPT2: This strategy loads data into local memory before computation, to reduce global memory access. It is a typical GPU optimization method. Data is read into local memory of size ($m \times m = (n + 2 \cdot r) \times (n + 2 \cdot r)$), which depends on the work-group size $n \times n$ and the radius r of the input window. The input is split into four areas, which are read by the work-items in four separate operations, if ($2 \cdot n \geq m$), due to the image borders.

OPT3: The last strategy is like `OPT1`, but without vectorization. It loads the input window into registers, which are used as sliding window. Since input data is marked as `read_only`, `constant` and `restrict`, the read only cache of the device can be used. Due to the SIMD structure of GPU architectures, the cache creates a line buffer, and each input pixel of a work-group is only loaded once. In comparisons to `OPT2`, no extra commands are needed to prefetch data into the local memory.

Extract Features: The implementation of this function contains three stages. The first stage detects the features of the image in parallel. It writes features of each row in an own vector using a global counter for that row, since features only need to be sorted by their y coordinate. The second stage computes the parallel prefix-sum, to sum the resulting row counters, to determine the position of a feature in the resulting vector. This is computed for each work-group (e.g., three to four) independently. The third stage gets the results of all parallel prefix-sums and writes the features to the correct position in memory. The CPU version computes the prefix-sum in simple loops, due to its architecture and the small number of CUs.

4.4 Automatic OpenCL Code Generation

The previous section looked at how OpenCL kernels can be implemented and optimized for different architectures and vendors. However, architecture dependent optimization can be very time consuming. Therefore, this thesis examines how this process can be partially automated or supported. This section will investigate the automatic conversion of existing C++ code to OpenCL code to be offloaded to accelerators in heterogeneous systems. With this approach, existing program code can be reused. The OpenCL model is well suited for this task since it follows a platform-independent approach, is supported by several vendors, and can decide at runtime on which hardware code should run. In principle, the code can run on any OpenCL-capable hardware but may not achieve the same speedup. Therefore, OpenCL kernels need to be customized specifically for the different hardware architectures, as described in the previous section. A major advantage of this approach is that source code can be easily analyzed, evaluated and modified by the developer. This would not be as productive if the developer had to do this using an intermediate language, since source code is more human-readable.

Therefore, this work proposes a source-to-source compilation toolchain, which recognizes profitable program parts in C/C++ code and automatically generates OpenCL host and kernel code from it. It processes on the IR language of the LLVM toolchain. The polyhedral model, which is based on Polly [203], is used to analyze the IR code. The provided information is transferred to the PPCG [206] since Polly has been designed for CPUs. PPCG itself is a transpiler but works on the AST of the Clang frontend and can only process C-code as input.

The approach of using the IR for the code transformation enables C++ as input language. It is also possible to process other input languages by changing the frontend (Clang). PPCG is optimized for GPUs and can generate OpenCL and CUDA source code. In this work, this functionality is used to generate OpenCL code together with the generated IR code. A main contribution is the creation of valid and human readable C (OpenCL C99) code from LLVM-IR basic blocks and regions. Furthermore, an LLVM module for recognizing and transforming local variables has been created.

The next subsection will give an overview of the proposed toolchain and briefly describes the different parts that have been adapted or added. The subsequent subsections will then describe the different parts of the toolchain in more detail.

4.4.1 Overview

This work developed a tool that uses Polly [203] to create a polyhedral model [197] and PPCG [206] to optimize it for OpenCL [144]. They are based on the ISL [196] library, allowing migration of the polyhedral model. Figure 4.6 shows the combination of the mentioned tools that have already been described in Section 2.5.1. The green boxes and lines highlight the proposed flow and the red boxes highlight parts that have been added in this work. Since PPCG is a transpiler that converts from C to OpenCL, CUDA and OpenMP, it already includes a way to convert the polyhedral model to source code. The Polly-ACC project integrated PPCG into Polly, to optimize the polyhedral model for GPUs and to generate LLVM-IR code from the new schedule.

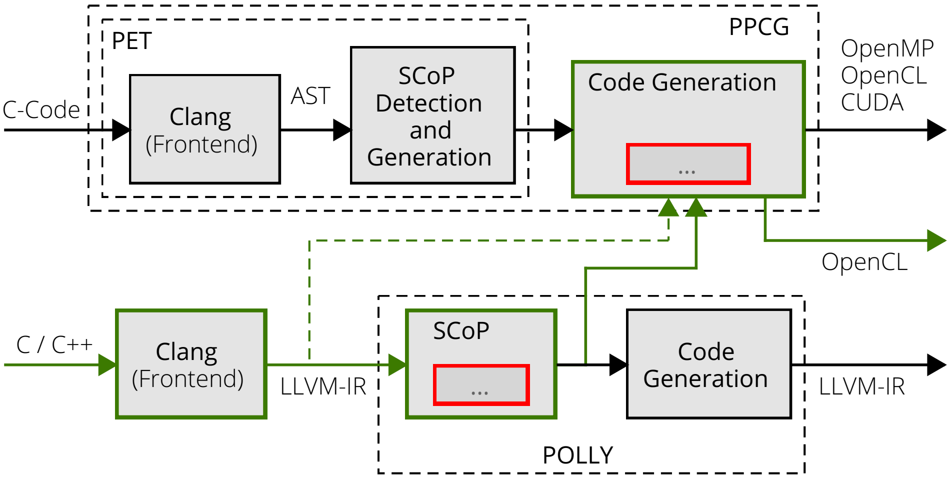


Figure 4.6: Overview of the combined Polly and PPCG tool design.

However, in this work, the source code is generated back from the LLVM-IR. PPCG can output the polyhedral model of Polly and generate valid C code for the OpenCL kernel, where statements are modeled as function calls. To generate functional code, the contents of the statements must also be converted from LLVM-IR code to C-code. However, during the optimization of the SCoP, PPCG does not analyze or touch the contents of the statements.

This work created a class called `CWriter` that can convert LLVM-IR into C code. In addition, the PPCG version 0.04 was replaced with the newer version 0.07. This offers "Live-Range Reordering" [207] to resolve dependencies with local variables (false dependencies) and assign them to a new data area. Thus, SCoPs that use local variables and have write-after-read data dependencies between loop passes can be converted. This can tell PPCG if storage area exists locally and if data is needed outside of the SCoP.

Polly creates a separate memory access with separate variables for each PHI instruction, which leads to the Loop-Closed SSA form. However, this creates unnecessary memory access to local variables that converge from one loop to another. Therefore, this work created a pass to analyze PHI instructions of loops according to the Polly Canonicalize pass and transform potential local variables into memory accesses with load and store instructions.

Figure 4.7 shows the structure of the proposed tool. The red blocks indicate the modules developed in this work. The `LocalVar2Mem` class converts relevant PHI instructions into load and store instructions, creating local variables. The `PPCGSourceCodeGeneration` class converts the SCoP information into the PPCG format, performs dependency analyzes, does the scheduling, and generates the OpenCL code using the `OpenCLWriter` class. The `OpenCLWriter` class is integrated in the `PPCGSourceCodeGeneration` class and generates the OpenCL host and device code using the new schedule. The `CWriter` class is integrated in the `OpenCLWriter` class and creates the statements. The different modules will be described in more detail in the following subsections.

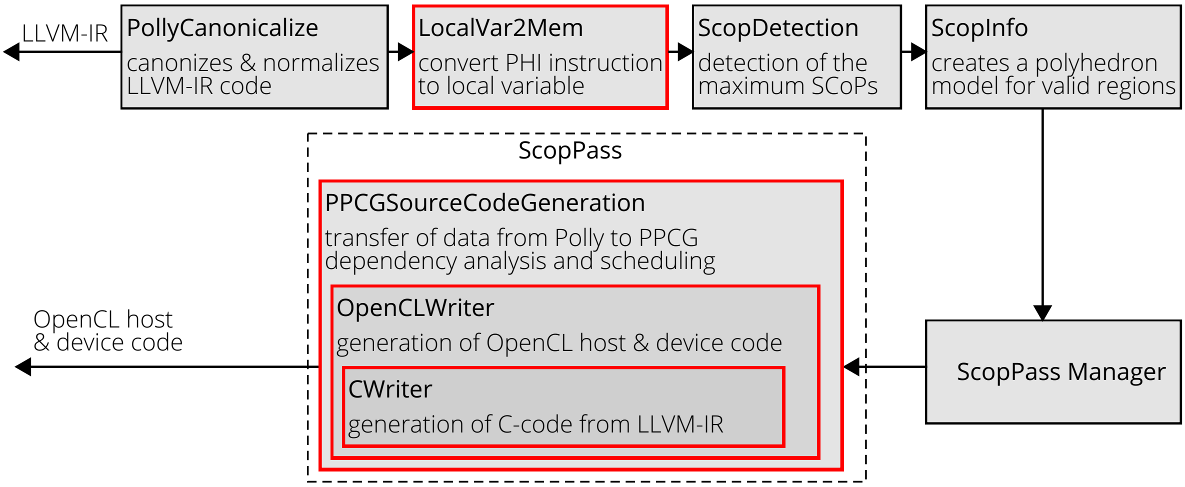

LLVM-IR
PollyCanonicalize
canonizes & normalizes LLVM-IR code
LocalVar2Mem
convert PHI instruction to local variable
ScopDetection
detection of the maximum SCoPs
ScopInfo
creates a polyhedron model for valid regions
ScopPass
PPCGSourceCodeGeneration
transfer of data from Polly to PPCG
dependency analysis and scheduling
OpenCLWriter
generation of OpenCL host & device code
CWriter
generation of C-code from LLVM-IR
OpenCL host & device code
ScopPass Manager

Figure 4.7: LLVM to OpenCL: Modifications of Polly and PPCG.

4.4.2 PPCG Source Code Generation

The `PPCGSourceCodeGeneration` class extracts needed information from the Polly SCoP. It stores all statements and their restrictions on the available parameters and their memory access relation (may-write, must-write, and may-read). It contains functionality to influence the optimization of the schedule. For example, the tile size for GPUs can be adapted or the use of the private and local memory can be activated. In addition, tagged variants of each memory access relation are created. These include a relation from a statement to a unique reference for a memory access. For example, tagged relations are used in PPCG as a reference to decide whether data areas need to be transferred to or from a device. To use the "Live Range Reordering" feature, memory access relations, in which the data of the memory location is only used within the SCoP, are collected. Additionally, all memory locations, which can be accessed outside the SCoP, are collected. Since Polly does not create kill statements for local variables and PPCG cannot detect if these variables are used outside. Using the information of the memory access relations, the dependency relations and data flow relations are created.

The class also transforms the statements and array sizes from Polly and analyzes the memory space of arrays that can be considered persistent to the SCoP. Then it creates a new schedule and marks the loops that can be parallelized. It also creates markers to initialize the device, transfer data and start kernels. Then the schedule is converted from PPCG to an AST. The function also creates new expressions (e.g., from $A[i][j]$ to $A[t_0 + b_0][t_1 + b_1]$), which are stored in a hash table and used in the `CWriter` class, of the memory accesses for the statements. Lastly, the created OpenCL host and device code is stored in files. Many of the features described for this class have been taken from the Polly-ACC project and adapted for the new PPCG version.

4.4.3 Creating OpenCL Device and Host Code

The `OpenCLWriter` class creates the OpenCL host and device code. It manipulates the AST by using functions from the ISL library and converts them into OpenCL source code. The ISL library offers the possibility to transform ISL expressions and structures into strings. It supports the C syntax and YAML, which is based on XML, as output. The different SCoPs are processed one after the other.

Functions for creating OpenCL host code have been adapted and automatic performance measurements of OpenCL kernels have been added. Additionally, function headers for the OpenCL host code are created since this did not exist in PPCG. The arguments of OpenCL host functions are created by iterating over the list of arrays of a SCoP and by considering parameters defined outside of the SCoP. Comment lines are stored for host and kernel function headers using information of the original source code. This includes the location of the original file in the file system, SCoP line numbers and original function names. If debug information is available in the LLVM-IR, the SCoPs function name is extracted from it. A nice feature that was achieved through this is that the tool can also manage C++ template functions, which is used in the `HiFlipVX` library. The square brackets of the function name must be removed beforehand.

Code for initialization, data transfer, kernels and kernel calls need to be created. The initialization generates code for the OpenCL context, program compilation and command queue. If it is not an initialization or data transfer, it is to determine if statements need to be created that PPCG has not assigned to the kernel. This can happen if the statements perform invariant calculations and do not need to be executed in the kernel. Together with the `CWriter` class, C code is generated from the LLVM-IR. Then the kernel code and calls in host code are created. Strings are set for OpenCL barriers and instructions are generated for copying data to or from local storage. Polly does not hold all variables that converge from the outside into the SCoP, since only arrays and loop limiting parameters are detected. If parameters are loop invariant, they are only cached if they occur in a domain entry. Other invariant variables used in the loop body are not captured. These parameters are created by the `CWriter` class during the analyzes of the LLVM-IR.

4.4.4 Conversion from LLVM-IR to C-Code

The LLVM-IR statements are converted into valid C-code in the `CWriter` class using the `printStmt` function. First, the hash table `LoopToSCEV`, which is for the new iteration variables of the statements, is filled. Function arguments must be cached to generate the statements in the hash table. They are needed to replace the previous iteration variables in the LLVM-IR code, since they change for the OpenCL code generation. With SCEV (SCalar EVolution) expressions, the new term can then be determined for an iteration variable. SCEV [271] is a technique to model memory accesses within loops so that they can be represented in terms of iteration variables. For each argument, a C-code term is generated using the ISL library. Then, all read operations in the statement are analyzed, to be stored in the `ValueToString` hash table. Each memory access in Polly has a unique ID and points to an `isl_ast_expr` in the hash table, which contains the new expression for the memory address. The `isl_ast_expr` is created as a string in C syntax and stored in the hash table for the corresponding read accesses in the LLVM-IR.

If the statement is a basic block, `printInst` is called for each instruction of the basic block and the write accesses to PHI instructions are processed. To prevent redundant code, `printInst` checks if an instruction is used more than once in its basic block. In this case, a local variable is created and the contents of the `ValueToString` hash table are replaced with the name of the local variable for the instruction. In addition, an investigation takes place whether the instruction represents a write access in the current statement. For a write access, the content of the variable from `ValueToString` is formed with the `isl_ast_expr` as an assignment in a new program line and converted into a string.

If the statement is a region, a function is called recursively until the own region cannot be divided any further. A region consists of a single-entry and single-exit area. It can exist of several basic blocks, which are coupled to each other with conditions. The next step is to parse the region by iterating from the inbound to the outbound basic block. At the end of a basic block, the branch instructions are checked and created.

4.4.5 Converting PHI Instructions

The Polly Canonicalize pass modifies the LLVM-IR to match the Loop-Closed SSA form. The `ScopInfo` pass generates separate memory accesses for each store, load, and PHI instruction. Thus, information about a possible local variable is lost, which might have enabled a better optimization of the SCoP. The PPCG optimizer can no longer recognize if there are multiple memory accesses to the same data because the contents of the statements are not considered. These are only used with the `printStmt` function after the scheduling. In `localvar2mem`, it is iterated over all regions of a function, and it is tried to translate PHI instructions into local variables. This is done by replacing the PHI instructions with load and store instructions.

However, PHI instructions cannot be easily changed into memory accesses. Only one memory location should be used for the representation of a local variable. Therefore, only those PHI instructions are transformed, in which the register of the PHI instruction is used only within the same basic block. To convert the relevant PHI instructions, single-entry single-exit regions are recursively searched for the beginning of a loop. After a region is found, it will be iterated over its basic blocks. Within the basic blocks, PHI instructions are examined, which do not represent iteration variables and are only used within the block.

Valid PHI instructions are stored in a list. For an observed PHI instruction, it is examined whether it is already assigned in the list to a local variable. To translate the PHI instructions into load and stores, it is iterated over each entry in the list. First, the store instructions for the PHI inputs are generated for the basic blocks. Then, the load instructions are generated and the use-define chain of the PHI instruction are replaced with these instructions. Lastly, the PHI instruction is cleared and removed from the basic block.

4.5 High-Performance Vision Toolchain

This section presents the High-Performance Vision toolchain, which is one instantiation of the `DECISION` framework. Using this toolchain, users can implement computer vision algorithms for x86-based systems consisting of CPUs, GPUs, and FPGAs without having to deal with the underlying hardware architecture. The toolchain automatically distributes the functions to the different architectures. OpenVX is used as frontend, OpenCL for the backend and runtime system, and `HiFlipVX` as library for FPGA devices. For all other devices, OpenCL libraries can be included, which will be shown using OpenCV and AMDOVX as examples. It is also possible to include self-optimized or automatically generated kernels as described in the last two sections.

Using OpenVX in the frontend has the advantage of a well-defined standard for computer vision, a good level of abstraction, and a graph based approach which is ideal for the distribution of applications. Another advantage is that there are already several OpenVX-based libraries that can be integrated. `HiFlipVX` has been integrated, which has the advantage of being performance- and resource-optimized and offers many possibilities for fine-tuning through various parameters. The middleend focuses on the mapping and scheduling problem, which is solved with the help of FPGA synthesis estimations and CPU/GPU profiling results. OpenCL is used as main low-level API, since it is very comprehensive and supports numerous vendors and devices. However, other C++ based languages can also be integrated into the runtime environment through native kernels. The backend and runtime system are designed to allow for simple and fast execution on multiple devices without the boilerplate code overhead that comes with the OpenCL API. This includes the creation of data transfers and synchronization points as well as the preparation of the final program which is executed on the OpenCL-based runtime system.

The following subsection gives an overview of the developed modules and models, their configuration, and the FPGA integration. In the later subsections, the individual modules are explained in more detail.

4.5.1 Overview

Figure 4.8 gives an overview of the modules, the models, and the tool-flow of the High-Performance Vision toolchain. It can run OpenVX graphs on multiple OpenCL-capable devices (CPUs, GPUs, and FPGAs) from different vendors. It supports and integrates XILINX FPGAs including bitstream generation. The toolchain is composed of several modules that have been separated from each other by using well-defined models. This simplifies the addition of new libraries, scheduling algorithms and APIs. They communicate with each other via files that either describe the models or contain other data, such as profiling results or binaries. The programmer can modify the models at design time, which provides an easy way of testing, debugging, or making own customizations, such as changing the mapping.

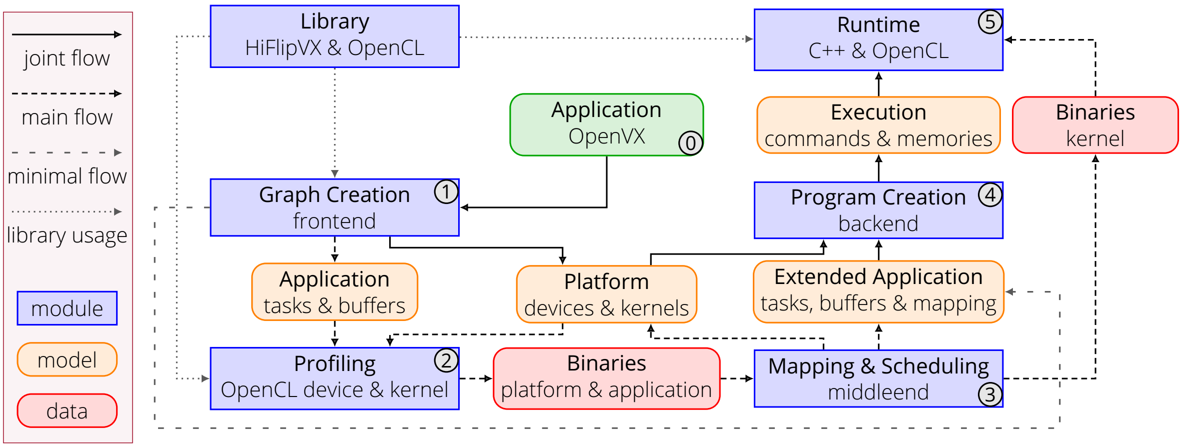


Figure 4.8: Overview of the proposed High-Performance Vision toolchain.

The dotted lines show the use of the library consisting of `HiFlipVX` for FPGAs and OpenCL-based functions for other devices. The dense dashed lines show the main tool flow, whereas the loosely dashed lines represent an alternative, where the user can decide on which device the vision functions should be executed. The solid line shows the part that is the same in the main flow and in the minimal flow. The following gives a brief overview of the different modules and models and their use within the tool flow. After that, the configuration options of the toolchain and the integration of the FPGA are described.

Module Overview

Library module: The task of this module is to integrate the `HiFlipVX` library for FPGAs, and OpenCL capable libraries for other devices. Important for the selection of the OpenCL libraries is that the contained functions are conform with the OpenVX specification. For this purpose, the extraction of OpenCL functions from different vision libraries was examined. A list of OpenCL kernels is read by the OpenVX Graph Creation module and stored for all other modules within the platform model. This approach allows an easier integration of new libraries or functions. Additionally, the OpenCL kernel source code can directly be used by the Profiling module or the Runtime System module to create the binaries at design-time or runtime, respectively. The `HiFlipVX` library is directly integrated into the OpenVX Graph Creation module to create the IP-cores.

OpenVX Graph Creation module: This module is needed by the user to implement an OpenVX application consisting of nodes and images. Using this module, the various kernels from the Library module can be utilized. It provides many functionalities of the standard and integrates the `HiFlipVX` library. Its main functionalities are, the creation and verification of the application graph, and the creation and synthesis of the IP-cores for the FPGA. It generates the application model from the application graph including synthesis results and stores the IP-cores for the Profiling module.

Additionally, the module can search for all available OpenCL platforms and devices of the system. Besides this information, the module scans the Library module for OpenCL files (programs) and lists the defined kernels. All information obtained by the module is stored in the platform model and provided to the OpenVX Graph Creation module and the Program Creation module. With the help of this information the OpenVX Graph Creation module can create an extended application model for the Program Creation module, which can be adapted by the user.

Profiling module: First, the available OpenCL devices are evaluated in terms of memory bandwidth and PCI-e transfer rates. Then, all available OpenCL kernels are executed on the CPU/GPU devices to obtain their latencies. For FPGA devices, the HLS synthesis reports forwarded by OpenVX Graph Creation module are used for the latency estimation. At the end, the obtained profiling data and extracted synthesis results are passed to the Mapping & Scheduling module including the application and platform models.

Mapping & Scheduling module: In this module, tasks are mapped to devices and a schedule is created to estimate the total execution time and achieve an optimized mapping. It is designed in such a way that new scheduling and mapping algorithms can easily be added and used. In this work, the HEFT algorithm was implemented to select devices for tasks based on their properties. Since FPGAs have special properties that are very different from CPUs and GPUs, the algorithm had to be adapted. Among them are the limited resources and TLP enabled by pipelining kernels. After scheduling and mapping, all new IP-cores and the bitstream are generated. The binaries of all devices are then forwarded to the Runtime System module.

Program Creation module: The task of this module is to take the solution of the Mapping & Scheduling module and to generate the OpenCL program needed by the Runtime System module. Its goal is to simplify and optimize the execution with regards to the utilized OpenCL API calls and required memory transactions. It also implements a memory abstraction layer including a MSI (Modified Shared Invalid) protocol to increase runtime performance and reduce code complexity. For this purpose, all data transfers between devices are calculated, synchronization points are generated, and additional memory objects are created.

Runtime System module: The task of this module is to execute the final program. It has been optimized to achieve high-performance by reducing computational overhead and parallelizing execution. It executes the OpenVX applications of the OpenVX Graph Creation module that were translated into an OpenCL program by the Program Creation module. The final program is divided into system and application functions. The system functions are executed only once to initialize or terminate the program and build all kernels. The application functions run an OpenVX graph and can be executed multiple times. In addition, the user can execute own functions or access the memory objects of the generated program.

Model Description

As shown in Figure 4.8, there are various models which are used as interfaces between the different modules. There are three important models, which are described in more detail below: (1) the platform model describes the system architecture, consisting of the various compute nodes, (2) the application model describes the application to be executed in the form of a DAG, and (3) the execution model describes the program flow including data transfers and synchronization points. In addition, there is further meta-data including the

FPGA IP-cores and their synthesis results, the OpenCL profiling results, and the generated binaries.

Platform Model: The platform model is generated by either the OpenVX Graph Creation module or the Profiling module and consists of two parts. The first part of the model includes the available OpenCL platforms (e.g., AMD, NVIDIA, XILINX, or Intel) and the usable devices (e.g., CPU, GPU, or FPGA) of each platform. To use the devices of a platform a context is created in OpenCL. All OpenCL objects belonging to a platform are stored in its context. As shown in Figure 4.9 on the left, there are three different types of devices. The advantage of the host device is that it can also execute native C/C++ functions within the OpenCL framework, which are not compiled with an OpenCL compiler. To execute native functions a specific native command queue is needed, and its arguments (buffers) need to be mapped into host space. The IACCs (Integrated Accelerators) share memory with the host, while the DACCs (Dedicated Accelerators) have their own memory. The second part of the model contains all OpenCL programs (files) and their kernels (functions) that can be executed on the devices. It also includes the native C/C++ kernels.

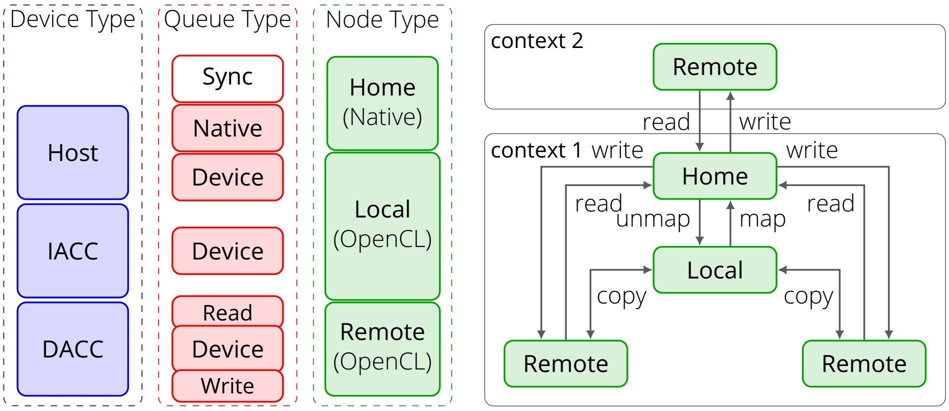


Figure 4.9: Relation between device, queue, and node types (left) and OpenCL data movement commands (right) needed to transfer data in a SVM system.

Application Model: The application model consists of two parts, which are generated by either the OpenVX Graph Creation module or the Mapping & Scheduling module. The first part of the model includes the different buffers, e.g., OpenVX (virtual) images, and their size. It is also necessary to know whether the data should be accessed before or/and after the OpenVX application. Since this information is not always apparent without the intervention of the user, default parameters are set in the OpenVX Graph Creation module, which can be modified by the middleend. Thus, all buffers that have no source are marked as writable, and all buffers that have no destination are marked as readable. One effect of this approach is that input data is automatically copied to the respective computing devices and output data is copied back to the host. Another effect is that the buffers can be reused for other purposes after an application . Furthermore, the application graph can be executed multiple times in a loop.

As shown in Figure 4.9, three different node types have been defined for a SVM (Shared Virtual Memory). Each DACC that uses a buffer has its own remote OpenCL memory object. All IACC and the host share a local OpenCL memory object. The home memory object and the local OpenCL memory object point to the same physical memory address. However, the home memory object can also be accessed from outside the OpenCL application. Due to the limitations of compute nodes in standard x86-based systems, a few restrictions can be made. Only one home node can exist per compute node. Furthermore, there can be multiple contexts within one compute node. However, only one of these contexts can have a local node. This context can have multiple IACCs and DACCs, but only one host. All other contexts can only consist of DACCs. On multisocket systems, all CPUs within one compute node will appear in OpenCL as a single CPU with multiple CUs. In the case of a cluster consisting of multiple compute nodes, an additional layer would be needed in the system, e.g., using MPI.

The second part of the model includes all tasks without data transfers and synchronization points. Tasks are instantiations of kernels with a given configuration, which are executed at runtime using command queues. As shown in Figure 4.9 on the left, each device has its own command queue to ensure that the execution is as parallel as possible. Additionally, the host has a special command queue for native C/C++ functions. This allows functions that have been parallelized using a different method, e.g., using OpenMP, to be integrated into the OpenCL runtime system. The DACC has two additional command queues to allow parallel reading and writing of data over the PCI-e bus, which is full duplex. All computation and data transfer kernels, which are executed on the same command queue will be executed in-order. The different command queues are synchronized with each other via OpenCL events. OpenCL also allows out-of-order execution within command queues, but this would require additional synchronization points, which is an unnecessary overhead. For this reason, there is one additional sync command queue, which is used to copy data between different contexts via the home node.

A task needs several parameters for its execution. On the one hand on which device and platform it is executed. On the other hand, which kernel is executed from which program. Additionally, the size of the NDRange and the argument list (function parameters) are needed. For most devices, the NDRange is equal to the resolution of the input image. However, for all FPGA functions the size is one, because the parallelization is done explicitly inside of the kernel. Each argument needs the buffer ID, size, and offset, and a flag for the access pattern of each argument (read, write or both). This access type also depends on the target device and its kernel implementation. For example, the optimized Gaussian accelerator of `HiFlipVX` only writes to its output image, whereas a simple implementation would read and write from/to the output image. The device and platform information are only included in the extended application model, as it must be generated either by the user or Mapping & Scheduling module. All transactions that occur will be generated by the Program Creation module out of the buffers, the task parameter lists and the available platforms and devices.

Execution Model: The execution model is generated by the Program Creation module and consists of two parts. In the first part, all information that is processed in the initialization phase of the Runtime System module is stored, to reduce the OpenCL-related overhead. First, it contains the platforms and their devices, which are used by the applications. Second, it contains the programs and their kernels, which are used within the applications. Third, it contains the utilized buffers and their size. Each buffer has a list of devices that accesses the buffer. This entry contains the node type, the host and device access flags to the memory object and to which device it belongs. However, there can only be maximum one entry for the local node and maximum one entry for the home node.

The second part of the model stores the application(s) including all OpenCL commands needed for execution and synchronization. It contains a list with one entry for each command queue that executes kernels for computation, data transfer or synchronization. This entry contains its context ID, device ID and queue type. For each of these command queues there is a list of commands which are necessary to execute the application. Each entry in these lists is linked to a kernel command (native or NDRange) or copy command (copy, read or write). Other command queue commands, like `map`, `unmap` or `barrier` are integrated within the corresponding entry of the list since they can only occur together with the other commands. Depending on the queue type (native, device, read or write) different data needs to be stored. These are: (1) the program and kernel, (2) the NDRange, (3) the buffer IDs and sizes (parameter list), (4) the input events and output event, and (5) the `unmap` and `map` commands. The events are used to profile each command on the command queue and to synchronize kernels between the different in-order command queues.

An additional list contains all events and their connections to each other. This is because commands of different contexts are synchronized with each other by connecting normal events with user events, and multiple user events can be triggered by one normal event. The last list of the model stores the initial state of each buffer for a particular application. Since this information is application-bound, it cannot be included in the buffer list for the initialization phase. Storing the initial state is important to allow the Runtime System module to restore this memory state, e.g., when executing the application multiple times.

FPGA Integration

OpenCL provides two API functions to program compute devices. The first one expects a string of OpenCL C code which is compiled at runtime. The second one expects precompiled binaries of device-specific code or an implementation-specific IR. FPGA devices are only capable to use precompiled binaries, since bitstream generation is very time consuming. Since the Profiling module for CPUs and GPUs must generate all binaries for profiling, the second approach can also be adopted for these architectures. In terms of performance, using precompiled binaries did not cause any disadvantage.

The generation of the bitstream for the FPGA is triggered after executing the Mapping & Scheduling module. If multiple kernels, which are consecutive in the graph, have been mapped to the same FPGA, new IP-cores are automatically generated that combine them into one kernel. To connect multiple concurrent functions in one IP-core, the `dataflow` and `stream` are needed. The rest of the IP-core generation has already been explained in the OpenVX Graph Creation module in Section 4.2. After bitstream generation, all binaries needed to run the application are prepared for the Runtime System module.

To integrate XILINX FPGAs into the rest of the OpenCL program, the vendor toolchain needs to be integrated. This toolchain is named SDAccel until version 2019.1 and Vitis thereafter. The FPGA itself can be programmed using the integrated `xocc` compiler. To do this, each kernel is compiled independently into an (`.xo`) object file, and then linked together to create the binary (`.xclbin`). One advantage of this tool flow is that besides OpenCL, also C/C++ can be used to implement an FPGA kernel.

Configuration

An incremental `Makefile` was created to combine the different modules of the High-Performance Vision toolchain. It can be used to compile and execute the different modules of the toolchain in the correct order. The `Makefile` enables linking against numerous OpenCL implementations, either directly or by using the ICD (Installable Client Driver) loader. It also integrates the XILINX FPGA toolchain (Vitis or SDAccel). In addition, there is a set of parameters which allow the user to configure the toolchain.

- `PREFERRED_OCL_PLATFORM`: Selects the desired platform from the platform model.
- `PREFERRED_OCL_DEVICE`: Selects the desired device from the platform model.
- `DIRECTLY_WRITE_SCHEDULE`: Allows to create the application graph and map the vision functions that are used in the Program Creation module, according to the preselected platform and device. The output can be used to skip the Profiling module and the Mapping & Scheduling module, or to manually create the mapping. Setting the parameter would also make sense if, e.g., there is only one accelerator, and it should execute all vision functions.
- `PROTO`: Creates the application graph as protobuf output for the Profiling module.
- `GENERATE_XO`: Switches between the IP-core generation methods of the two toolchains in the `DECISION` framework. If set, it generates the required `.xo` files for the SDAccel (Vitis) tool from XILINX.
- `USE_PRECOMPILED_BINARIES`: Required by the Runtime System module to indicate if the binaries generated by the Profiling module should be used to program the devices. Otherwise, the source files of the Library module are used to create the binaries at runtime.

4.5.2 Library Module

An important part of the High-Performance Vision toolchain are the various kernel implementations from different computer vision libraries. A major component of the Library module is the `HiFlipVX` library, which allows addressing FPGAs. Since OpenVX is an open specification, there are many vendors that provide compatible devices and compliant implementations that can be added to the Library module. The conformance of these libraries makes OpenVX applications portable and allows them to run on the devices of the same vendor. Some of these libraries use OpenCL to implement their kernels. To make use of these libraries and overcome the vendor binding, this thesis investigated the automatic extraction of OpenCL kernels for OpenVX functions. In addition to OpenVX-compliant vendor implementations, there are also numerous other vision libraries that use OpenCL and provide similar functionality.

As described in Section 2.4.2.2 there are several libraries that are either OpenVX compliant or implement a similar functionality. As a proof of concept, the AMDOVX library has been partially integrated into the Library module. Therefore, for each of the inserted libraries (AMDOVX & `HiFlipVX`), there is an entry with the functions contained in it. As a fallback solution, the Library module contains a set of default function that are not represented by one of the integrated libraries. In the current state it consists of four simple OpenCL implementations which should be executable on every OpenCL capable device.

Besides AMDOVX, AMD also offers the newer MIVisionX library, which requires PCI-e 3.0 atomics and therefore cannot be used by all available GPUs of this work. Therefore, this work use the AMDOVX library. Since no changes to the library were necessary, the packages provided by the distribution's package manager has been used. Unfortunately, the AMDOVX library implements OpenCL kernel only for GPUs, while it provides highly optimized x86 code for CPUs. To use them, the High-Performance Vision toolchain provides the possibility to run native C++ kernels.

As described in Section 2.4.2.2, AMDOVX consists of several tools. The RunVX tool generates kernels with additional control parameters, which makes its kernels incompatible with the High-Performance Vision toolchain. To create compatible kernels, the tool source code was adapted to extract the control parameters and define them as internal variables within the kernel source code. To use the AMDOVX implementation, the required functions are generated before they are used in the Profiling module. A patched fork of the repository has been created [272].

The rich feature set of OpenCV makes it a perfect candidate to develop computer vision applications. Fortunately, some of its functions support OpenCL device acceleration. Since these functions have been optimized according to their algorithm, they are perfect for kernel extraction, which can then be used within the toolchain. Unfortunately, the OpenVX specification is not fully covered by the available OpenCL kernel implementations. Before these kernels can be used in the Library module, their function headers must be adapted in a similar way to what has been done with the AMDOVX library. Since OpenCV does not provide a command line interface to easily use these kernels, this is a manual task.

4.5.3 Profiling Module

To make an accurate distribution in the Mapping & Scheduling module, the system, its devices, and the application kernels need to be evaluated. In the Profiling module this is done by profiling the devices within the system with respect to their available bandwidth, PCI-e transfer rate, and execution overhead. Additionally, the kernel execution times are determined for CPU and GPU devices via profiling and for FPGAs using estimations.

The Profiling module starts by reading the application model and the synthesis results. The available platforms and devices are then detected and stored in the OpenCL context. From this the platform model is generated. Then the devices and PCI-e are evaluated. Next, the kernels are read, compiled, and profiled. In addition, it verifies if an OpenCL source file implementing that function is available. The available kernels are stored together with the platform model. Finally, the profiling data and the models are passed to the Mapping & Scheduling module.

OpenCL Devices

One overhead of dedicated devices is that data must be transferred between host and device via the PCI-e. To calculate reliable timing data, the bandwidth and latency of each device connected via PCI-e needs to be profiled. Due to different drivers and hardware architectures, different devices can achieve different transfer rates even over the same PCI-e. There are three different types of transfers: host-to-device, device-to-host, and device-to-device.

Inside the Profiling module there is a function which measures the transfer rates between device and host using the OpenCL functions. By using prepared buffers with a size of 1 kB to 256 kB in 1 kB steps, it measures the transfer rate of the device. For the measurement on the

FPGA, the `xbutil dmatest` command line utility from XILINX was used. However, it can only transfer buffer sizes that are a multiples of two, which reduces the available measurement points. The process is repeated several times to calculate the average transfer rate. A linear function for the transfer rate, consisting of the latency and bandwidth, is derived from the measurements.

$$\text{transfer\_ time [μs]} = \text{latency [μs]} + \frac{\text{buffer\_ size [kB]}}{\text{bandwidth [GB s}^{-1}\text{]}}. \tag{4.7}$$

CPU and GPU Kernels

All kernels of in the input application graph require profiling data. Therefore, every kernel is executed on all available CPU and GPU devices to get the timing results. For this purpose, each kernel is compiled and stored as a binary file, which can be reused later in the Runtime System module to execute the function. The program code of this module is also well suited to execute the kernels on the devices. Additionally, the Profiling module checks if there is a kernel implementation in the Library module for the respective device. This can be, e.g., a generic or vendor specific implementation.

To make a reliable prediction, each kernel is executed several times. As metric two different methods can be chosen to catch the outliers. In the first method the median value is taken, so outliers in both directions can be avoided. In the second method the average value of the N percentile best results is chosen. Both methods try to filter the upper outliers, which can occur through background work of the OS.

FPGA Kernels

Since bitstream generation would be too time-consuming for the FPGA, the synthesis results from the OpenVX Graph Creation module are used to estimate the execution time. If the kernel is not memory bound, the determined execution times from the synthesis tool are very accurate. They are only missing the additional latency, needed for the data transfer to and from DDR memory of the FPGA, and for starting the OpenCL kernel. The streaming capability and similar structure of the `HiFlipVX` functions, allows a simplified estimation of the overhead. The overhead of the data transfer results from two delays: (1) Transfer time needed for the first data until the function has started ($t_{transfer_{start}}$). (2) Transfer time of the last byte after the function has finished ($t_{transfer_{end}}$). The overhead to start kernel execution also has a constant factor (t_{kernel}) that increases with the amount of kernel parameters that need to be transferred ($t_{parameter}$). The total overhead can be determined for the respective device, independently of the kernel. It only needs to be determined once per system and does not need to be recalculated for each kernel or application.

$$t_{overhead} = t_{transfer_{start}} + t_{transfer_{end}} + t_{kernel} + N \cdot t_{parameter} \tag{4.8}$$

If the function is bandwidth limited, the total latency calculated by the synthesis tool changes by a certain factor. This is a very rough estimation of the bandwidth limitation, which is accurate enough for most vision functions. Since the data transfer of a function cannot always be divided evenly over the duration of the function, a more accurate model would be needed for those cases. Therefore, the execution of a function would need to be divided into time intervals. An example of this is the histogram function, which consists of several loops. Equation (4.9) shows the effect of the bandwidth limitation and for a function that has been divided into several time intervals.

$$t_{estimated} = t_{overhead} + \sum_{i=0}^{N-1}(t_{latency_i} \cdot \min(1, \frac{band_{kernel_i}}{band_{memory}})) \tag{4.9}$$

4.5.4 Mapping and Scheduling Module

In the preceding modules, an application graph and model were created from an OpenVX application. In addition, its nodes were profiled on CPU and GPU devices, or estimated metrics were derived from FPGA synthesis results. This subsection discusses the generation of an optimized mapping and schedule using an adaptation of the HEFT [112] algorithm. While the HEFT algorithm schedules the nodes of the application graph on common OpenCL devices such as CPUs and GPUs, some adaptations and limitations must be considered for FPGA architectures.

FPGA-based Constraints and Characteristics

One major constraint that limits computational performance is the limited amount of FPGA resources. To make the best use of resources, the user can consider several factors. For example, when using `HiFlipVX`, these would be the vector size, or the number of functions being executed concurrently. Thus, the trade-off between memory and computational effort must be considered.

A major difference compared to common CPU and GPU approaches is the simultaneous execution of different kernels running in a pipelined manner, resulting in a higher utilization of hardware resources, as shown in Figure 4.10. This approach is enabled by the pipelining and streaming capability of the `HiFlipVX` library functions, where data can be streamed from one function to another and directly processed. The streaming of functions is mainly enabled by using a sliding window approach, which also reduces the utilized memory bandwidth.

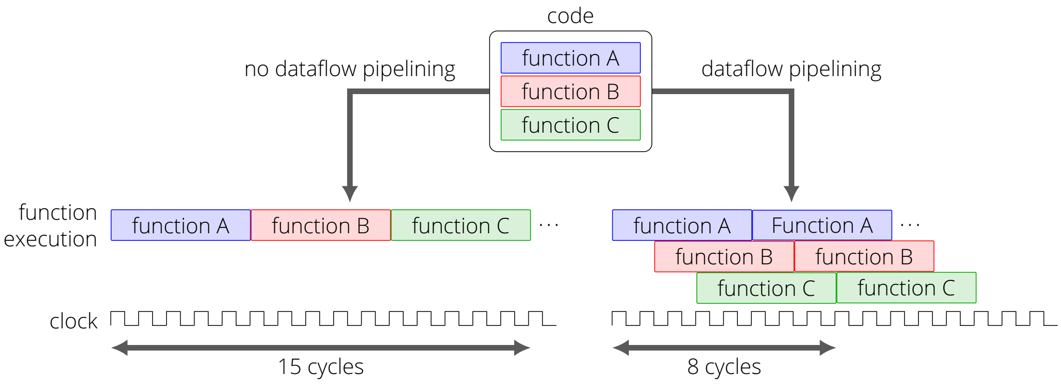


Figure 4.10: Dataflow pipelining latency effects, adapted from [214].

The estimated latency of a single function can easily be calculated from the synthesis results using the trip-count, depth, and interval of the pipeline. The trip-count can also be calculated using the vector size, image resolution, and sliding window size. The estimated execution time for executing multiple streaming-capable functions in a pipeline can be derived from aforementioned variables. The overhead of executing multiple functions results mainly from filling the line buffers and the sliding window, and from the resulting pipeline.

As shown in Equation (3.25), increasing the vectorization reduces the overall latency almost to the same extent. It can also affect the pipeline depth and the achieved frequency, based on the dependencies of the implemented algorithm. Resources are affected almost linearly. However, due to compiler optimizations and shared logic, they usually increase at a lower factor than the vectorization level. In addition, there is an overhead for interfaces and control logic.

Algorithm

The focus for the selection of nodes in the FPGA-based mapping algorithm is the pipelining of functions. The algorithm aims to create the longest possible pipeline, and thus achieve the highest possible parallelization to increase maximum performance. First, the entire DAG is placed on the FPGA, since this eliminates all additional data transfers to main memory, except for the initial input and final output. If the entire graph cannot be placed on the FPGA due to resource constraints, subgraphs are created.

If no subgraph could be created, the first node of the longest path in the DAG is chosen as starting point for the algorithm. The main reason for this approach is that starting from this point the longest pipeline can be created and thus the highest degree of parallelization can be achieved. If no subgraph meeting the resource constraints can be created, it will be iteratively split into smaller subgraphs. After each division, the resource constraints are checked again. If still enough resources are available, the subgraphs are extended by further nodes to utilize the available resources and reduce the total make span. It must be noted that a 100 % utilization of the FPGA can lead to problems with the place and routing process or minimize the maximum achievable frequency.

Subgraphs refer to a part of the original OpenVX DAG that is only connected to the rest of the graph via maximum one input and one output point. This procedure reduces the needed bandwidth and prevents possible deadlock situations. During the iterative expansion of the subgraphs, the algorithm ensures that these conditions are still met. For each of the resulting subgraphs the corresponding execution time is calculated, as shown in Equation (3.25). Using the HEFT algorithm, the remaining nodes are distributed on other available devices, and the total execution time is calculated. Finally, the distribution with the lowest execution time is selected.

For the described approach, the HEFT algorithm had to be adapted to detect concurrent data transfers and include them in its scheduling decision. While the original order of nodes was not changed, additional delays were introduced. When a data transfer occurs, a parallel transfer is executed with the rest of the available bus bandwidth if other transfers are taking place.

4.5.5 Program Creation Module

The task of the Program Creation module is to take the results of the Mapping & Scheduling module and to generate the complete information needed by the Runtime System module. The goal is it that the Runtime System module needs to perform as few as possible additional computations to manage the execution of the application. One focus is on the generation of a virtual memory system, which creates the inter-device data transfer. For this purpose, all data transfers, synchronization points (OpenCL events) and additional memory objects are generated. Thereby it is important that the path with the shortest latency is chosen for data transfer. Using synchronization points, the amount of data transfers can be reduced for multiple readers of the same buffer. Although the OpenVX Graph Creation module in the frontend and avoid multiple readers when using the FPGA, this is done to be usable on a wider scope. In general, the Program Creation module also works without the Mapping & Scheduling module. In this case, the user can either set the mapping manually or adjust it based on the application and platform models generated by the OpenVX Graph Creation module.

Shared Memory System

Table 4.1 compares the five different access patterns a kernel argument can have. In this work, a MSI protocol was implemented to ensure coherence, since there can be one memory object of the same buffer on different nodes. For this purpose, a reader-writer model was realized, where there can be either one writer or several readers of the same buffer at the same time. The buffer is in shared state if there is only one writer (owner) and no reader. It is in shared state if there is at least one reader. If an owner makes a read access to the same memory object in a subsequent kernel after a write access, the state also changes to the shared state.

Table 4.1: The different actions for the access patterns a kernel argument can have. How does the protocol change (modified, shared, invalid)? What happens to the other memory objects of the buffer (bus action)? What happens to the owner and the readers of the buffer? How is the kernel access during execution? How is the memory mapped into host space if it is a native kernel?

	read only	create write only	create read write	update write only	update read write
Modified	shared	-	-	modified	modified
Shared	shared	-	-	modified	modified
Invalid	-	modified	modified	-	-
Bus action	bus-read	bus-create	bus-create	bus-update	bus-update
Owner	-	create	create	update	update
Readers	add	-	-	clear	clear
Kernel access	read only	write only	read write	write only	read write
Host map flag	read	invalidate	invalidate	write	write

Depending on the access of a kernel to a buffer, a bus action is triggered. There are three different types of bus actions. A `bus-read` adds a new reader to the list of readers and the buffer goes into the shared state. A `bus-create` creates the new owner and can only occur if the buffer has not yet been used. A `bus-update` first reads the data, deletes all previous readers after they are finished reading, and creates a new owner. The "kernel access" represents the access from a kernel to its own memory object. When accessing the home memory object from the host side, the local memory object must be mapped into host space (read, write, or invalidate). The reason for the detailed observation is because the OpenCL runtime system can perform its own optimizations depending on the access pattern of the mapped memory object.

Program Flow

The first part of the Program Creation module generates the lists for the initialization phase of the Runtime System module. The second part generates the command queues, events, and updates the memory objects. Listing 4.3 shows a rough program flow of the second part, which takes the application model as input and creates the execution model as output. A main task of Program Creation module is to keep the buffers between the devices coherent and to create the required data transfers, synchronization points and additional memory objects. This is done by iterating over the execution list of the input (line 2). Each element in this list represents a native or NDRange kernel that should be executed by the Runtime System module via the command queues. For each argument of a kernel the required transactions, synchronization points and additional memory objects are generated in the `compute_transactions()` function (line 3-4). If data needs to be copied over the PCI-e bus, additional kernels to copy data are added to the output (line 3). Finally, the kernel, with all its synchronization points (events), and `map`, `unmap` and `barrier` commands, is added to the output (line 6). The second loop iterates over the memory list to copy all data needed after the application back to the home node (line 9). If data needs to be sent via PCI-e, copy kernels are added to the command queue (line 10-11). A final home kernel is added if data needs to be `mapped` from the local node to the home node (line 12).

```
 1 // Create commands for application
 2 loop: execution_list
 3   loop: argument_list
 4     compute_transactions()
 5     add_copy_kernels_to_queue()
 6   add_device_kernel_to_queue()
 7 
 8 // Copy needed data into host space
 9 loop: memory_list
10   compute_transactions()
11   add_copy_kernels_to_queue()
12 add_sync_kernel_to_queue()
```

Listing 4.3: Program flow of the Program Creation module to generate all commands executed on the command queues and the synchronization points between them.

The `compute_transactions()` function can be divided into several steps. In the first step, when a buffer is used for the first time, it is checked which state it had before the execution of the application. For example, if the buffer was written prior to the application, the state is set to shared state and the home node is set as owner. If the buffer has a local node, its memory object must be `mapped` into host space. In addition to this, the required memory object for the home node must be created. In the next step, the memory objects that are necessary for the memory access of the kernel are created. Then, the type of bus action, needed to access the data, is calculated (create, update, or read). In case of a `bus-read` or `bus-update`, the data must be read from the closest owner or reader. In case of a `bus-create` no data needs to be read.

Transaction Distance/Latency Calculation

The next step of the `compute_transactions()` function calculates the nearest memory object of a buffer to read from in case of a `bus-read` or `bus-update`. For this purpose, the distance to the owner, the readers and all other events that were created while copying data is calculated. As synchronization point for reading, the output event of an owner or the input event of a reader is used. The distance that the data has already traveled to a reader or copy event is added to the final distance. Distance refers to the latency required to transmit the data.

The right side of Figure 4.9 shows the commands for transferring data between the different nodes. Various operations are included in the calculation of the distance (or latency). Commands like `map` and `unmap` or synchronization events have a lower latency. However, a distinction must be made between inter and intra context events. Since there are no bindings between the platforms of different vendors, it is not possible to synchronize with normal OpenCL events between command queues of different contexts. Therefore, special mechanisms were realized in this work, which are explained in more detail within Runtime System module.

The `map` and `unmap` commands are usually zero copy commands if the buffers are configured and aligned properly. They ensure that memory access between host space and device space is coherent. The copy commands, which send data via the PCI-e bus, therefore require more time. Measurements have shown that depending on the transfer direction and OpenCL command, different bandwidths and latencies can be achieved. In some systems and configurations higher transfer rates can be achieved with the OpenCL copy command than with the OpenCL read or write commands. Therefore, this work prefers the path from remote node to home node via the local node, if it exists, compared to the direct path using the read command.

Kernel Transaction and Synchronization Creation

In the last step of `compute_transaction()` function from Listing 4.3, all commands needed for a transaction are generated. Figure 4.11 shows an example system consisting of three contexts to represent all different data transfer possibilities within one compute node. In this case, $d1$ – $d2$ could be a CPU, which can be used as a host or as a device, $d3$ could be an integrated GPU, and $d4$ – $d8$ could be dedicated GPUs and FPGAs. While $d1$ computes on the home node, $d2$ – $d3$ compute on the local node. All other devices have their own remote node. Each node represents a memory object in the same buffer. Memory objects are only created if they are accessed. In total, this work identified 18 different transfer patterns for the example system of Figure 4.11. Three further transfer patterns arise when the local node and devices *d*1 – *d*3 are omitted. A detailed examination of the transfer patterns can be found in Appendix A.1.

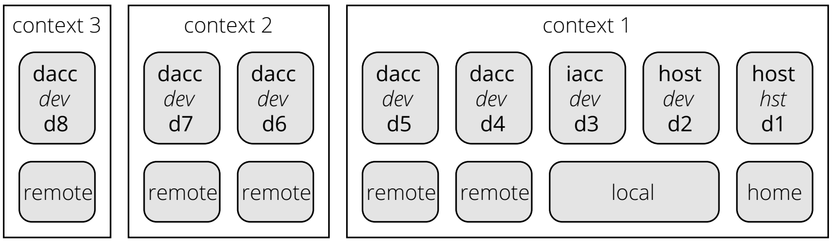


Figure 4.11: Example system to cover all possible types of data transfer.

For synchronization of data movement and kernels there is a series of events (*e*1 – *e*5), such as inter context events (*e*1) and intra context (*e*2). Barriers are used inside of device and host queues to wait until all input events of all transfers have been generated so that the kernel can be executed (*e*5). In case of a `bus-update`, the system additionally waits until all readers of the buffer are finished. After a `bus-read`, read commands can then read from these events (*e*3). If the data is copied via another node during transfer, additional synchronization points are generated from which data can be read (*e*4). A closer look at these points reveals that they occur either on the local or on the home node, which is used for further simplification.

There is a set of conditions (*c*1 – *c*6) for the execution of several commands. For a `bus-update`, the `barrier` waits until all readers have finished reading or the owner has finished writing (*c*3). Only then data can be written to the memory object. Memory objects that have been `mapped` for reading into the host space can be read from the local node without an `unmap` command. However, an `unmap` is necessary to read from the local node if the data was `mapped` for writing (*c*2). Conversely, it would also not be possible to copy data from a remote node to a local node if it was `mapped` to host space for writing. However, this case will not occur because the home node is closer to the local node as compared to the remote nodes. For a `bus-update`, an `unmap` is also needed if the data is `mapped` for reading before it can be written to the buffer (*c*1). If the host's `map` flags change, the `unmap` and `map` commands are needed to change these flags (*c*4). For example, after the flags have been changed from write to read, the local node could read from the same memory object in parallel without an `unmap`. It can happen that data was written to the home before the application was executed. In these cases, there is no input event for synchronization (*c*5). It makes a difference whether the source is the device itself or whether it is an event that occurred on the home node by copying data to it (*c*6).

For all transfers that go via the host, without it being the destination, the `map`/`unmap` commands are executed on an extra command queue. This queue is necessary so that on the one hand the host queue is not blocked and on the other hand the `map`/`unmap` commands are not blocked. If data has been copied over a node without being explicitly used by a NDRange or native kernel, an additional memory object must be created for the Runtime System module. If necessary, the flags of the other memory objects within the buffer also need to be updated. It updates the OpenCL memory access pattern, which show if a buffer is used for reading, writing or both. This information gives the OpenCL runtime system the opportunity to do some optimizations.

For read, write, or `map` commands, the memory access patterns need to be updated. For a read command from a remote to the home memory object, the home access pattern is updated for writing. Additionally, the remote access pattern is updated for reading. For a write command from the home to a remote memory object, the home access pattern is updated for reading. Additionally, the remote access pattern is updated for writing. For a `map` command, the access pattern needs to be updated depending on its flags.

At the end of the `compute transaction` function from Listing 4.3, the memory is updated depending on the bus action (read, update, or create). First, the MSI protocol of the buffer is updated, and second, the `map` flags for the home memory object are updated. A `bus-read` adds a new reader to the list if the device is not already on the list. Otherwise, the existing reader is updated. During a `bus-update`, all readers and additional synchronization points for reading are deleted. In addition, an owner is set in case of a `bus-update` or `bus-create`.

4.5.6 Runtime System Module

The generated application is executed by the Runtime System module. It is optimized to achieve high performance by reducing computational overhead and parallelizing various processes. The Runtime System module executes the OpenVX applications of the OpenVX Graph Creation module that were translated into an OpenCL program by the Program Creation module. Listing 4.4 shows an example program flow, which executes the same application several times in a row. The program can be divided into system and application functions. The system functions are executed only once for all OpenVX graphs, while the application functions can be executed multiple times. The system is like the OpenVX contexts, whereas an application executes a single OpenVX graph. In addition, the user can execute own functions or access memory objects of the OpenCL program.

System functions: First, a system object is created (Listing 4.4 line 3). Next, the system is initialized using the execution model (line 4). It creates the OpenCL contexts including all needed platforms, devices, and command queues. It builds all needed programs and their kernels. It builds all needed memory objects. In addition, it creates one thread for each command queue needed for the devices, as shown in Figure 4.9 on the left. In the termination function, all of these threads are terminated after they finished, and the profiling results are evaluated (line 13). After the initialization function and before the termination function, the created memory objects can be used in user defined code.

Application functions: The user can access certain memory objects before and after the execution of the application (line 7). For this purpose, the user had the opportunity in the OpenVX Graph Creation module to mark all memory objects which are used by the user before or after the application. To ensure that the memory objects can be used properly within the application, they must first be set to their initial state (line 9). In the `ApplicationRun()` function, the graph created by the Program Creation module is executed. For this purpose, all commands for the command queues are sent to the individual threads for execution.

During execution, the event objects are created. These events are used to synchronize the commands between different command queues. They cannot be created during initialization because they are created by the command queue commands and cannot be reused. If the appropriate flag is set, the events are also used for profiling. After all commands have been

```
 1 #include "../ocl_config/config_mapper_ovx.h"
 2 int main(void) {
 3   FrontendRunner runner; // System object
 4   runner.ApplicationInit(kOclConfigInitialization); // System Initialization
 5   // USER CODE
 6   for (int i = 0; i < N; ++i) {
 7     runner.ApplicationReset(kOclConfigApplication); // Reset application memory
 8     // USER CODE
 9     runner.ApplicationRun(kOclConfigApplication); // Run application graph
10     // USER CODE
11   }
12   // USER CODE
13   runner.ApplicationExit();// System termination
14 }
```

Listing 4.4: Complete program flow of the Runtime System module that executes one OpenVX application graph multiple times.

executed, the profiling results will be saved. Besides the profiling flag, there is a debugging flag, which gives additional information during the execution of the program. The following will explain the content of the four Runtime System module functions shown in Listing 4.4.

System Initialization

In the `ApplicationInit()` function (Listing 4.4 line 4) the execution model is used to create the majority of the needed OpenCL objects. One goal is to create all objects that can be created before the execution of the OpenCL applications to minimize the overhead of the OpenCL runtime system. Thereby, the contexts, command queues, programs, kernels, buffer objects, and platform and device IDs are stored. In addition, the required amount of memory is allocated in host space to minimize requests to the OS.

After all OpenCL objects are created, a thread is created for each command queue. Executing each command queue in a separate thread increases the performance, by reducing the waiting time in the host program. These threads wait in an infinite loop for new commands to be executed on their command queues. After all applications have been executed, the `ApplicationExit()` function (Listing 4.4 line 13) sends a termination message to all threads so that they can be terminated. If the profiling flag is set, all profiling results and timing measurements are evaluated and reported. In the following paragraphs the creation of the different OpenCL objects is described.

First, an OpenCL **contexts** must be created for each used platform. Contexts are used by the OpenCL runtime to manage command queue, buffer, program, kernel, and event objects. A context is created for all devices of a platform that will be used by the applications.

Second, for each device, a certain number of **command queues** is created. Figure 4.9 shows the number of command queues needed for the different device types. An additional sync command queue is created for the home node if there is at least one IACC or host device in the overall system. This command queue is in the same context as the IACCs and the host and is used as default command queue. A default command queue is set to reset the mapping of OpenCL buffer objects and for synchronizing data transfers between different contexts. All command queues in this work are created for in-order execution and use profiling if the corresponding flag is set. Creating multiple command queues and using in-order execution ensures an optimized performance while keeping the amount of synchronization events small.

Two pieces of information are needed, to determine the device type shown in Figure 4.9. The first is to identify if the device shares memory with the host (`CL_DEVICE_HOST_UNIFIED_MEMORY`). The second is to determine whether the device can execute native kernels (`CL_DEVICE_EXECUTION_CAPABILITIES`). Due to differences over the OpenCL versions, some functions are deprecated in newer versions or not available in older versions. These version conflicts can partially be managed by using the preprocessor macros of the host header files. Additionally, the OpenCL version of each device must be checked, as they may have a different OpenCL version, even within the same platform. A good example is the `clCreateCommandQueueWithProperties` command, which is used for all versions starting from 2.0, while the `clCreateCommandQueue` command is used for all lower versions.

Third, all OpenCL **programs** are created, built, and linked. An OpenCL program object can be created from source or from binary. For the first variant the programs are compiled at runtime and for the second variant precompiled binaries are used. For FPGAs, precompiled binaries must be used since bitstream creation is a very time consuming task. The precompiled binaries are created by the Profiling module for non-FPGA devices and after the Mapping & Scheduling module for FPGA devices. Within the execution model the program information is separated by contexts and devices. This allows programs to be built for specific devices and not for all devices of the same context. This separation is also important since it reduces the compilation time and since compiling OpenCL programs is the most computationally intensive operation in the initialization phase.

Fourth, all OpenCL **kernel** objects that are needed for the various applications are created. Separating the programs by devices has the advantage that kernels can be optimized for specific devices. For example, device specialized compiler flags can be used to optimize a kernel. Native C/C++ functions (or kernels) are not compiled using the OpenCL compiler.

Fifth, all OpenCL **buffer** objects are created, and host memory is allocated if necessary. To simplify the usage of memory and to hide data transfers, a SVM has been developed. SVM already exists in OpenCL. However, it did not exist before version 2.0. Since at the time of this work there are still some vendors, like XILINX or NVIDIA, that do not have a full OpenCL 2.0 support, it is not used. For each buffer in the execution model, there is one entry for each device that uses it. It is possible that an entry exists if the buffer is used by the home node. For example, this is the case if the buffer needs to be used by the host before or after the application. Maximum one memory object is created for all IACCs and the host, and one memory object is created for each DACCs.

When creating a memory object, the correct flags must be set. For this purpose, the device and host flags of the memory objects are read from the execution model and updated for each device that uses the same object. These flags determine whether the host or device accesses a memory object either for reading, writing, reading and writing, or not at all. An optimal configuration of the memory object allows OpenCL to make specific optimizations. A memory object can be accessed either by NDRange or native kernels, or by `map`, `unmap`, read or write commands. If a memory object is used by an IACC, the host or the home node itself, a home memory object is allocated in host space. The `CL_MEM_USE_HOST_PTR` flag must be set for the local node to use this memory object. This has several advantages. The memory object can be used before the application without any problems. The memory object will be allocated beforehand and not when it is used for the first time. When allocating the home node, the cache line size is used so that OpenCL can achieve optimal transfer rates (`CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE`).

Application Execution

The `ApplicationRun()` function (Listing 4.4 line 9) executes an application, which can be represented as an OpenVX graph. Figure 4.12 shows an example of the methodology used within this function. The goal of this methodology is to execute the OpenCL kernels with the lowest possible time overhead. In the main thread, the messages created by the Program Creation module are forwarded to the respective threads via message queues without needing to change them. A message contains a pointer that points to the command list that is targeted for a particular queue. A lock-free single-consumer, single-producer architecture is used to manage the message queues [273]. The threads iterate through the list of commands to process them one by one and to queue the necessary commands on their command queues. In addition, some synchronization mechanisms are used, which are described in the next paragraph.

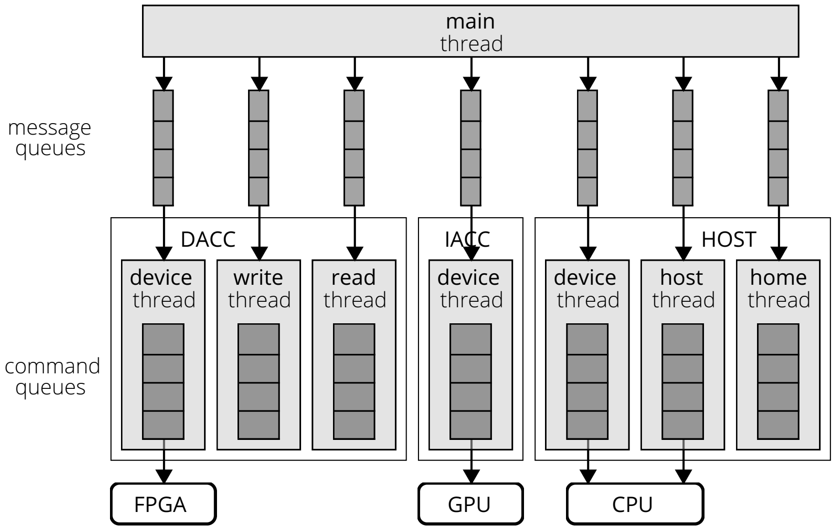


Figure 4.12: Multithreaded OpenCL runtime system to execute OpenVX graph applications.

Synchronization Mechanisms: One task of OpenCL events is the synchronization between different commands that are executed in command queues. A normal event object can have four different states: `queued`, `submitted`, `running` or `complete`. When an event is used as output of a command, it is created with state `queued` after the command is placed in the command queue. If one or more events are used as input, the command cannot be executed until all input events have reached the `complete` state. The synchronization of normal events is managed by the OpenCL runtime system.

Since each thread has its own command queue, it can enqueue its commands independently of the other threads. However, since events are not created until a command is enqueued, it can happen that a command is enqueued without having its input events being created.

Since this would lead to a runtime error, additional synchronization mechanisms are used between the threads to prevent this. These synchronization mechanisms work as simple barriers using a consumer producer principle. For this a simple `atomic` operation is used using `store` and `load` operations, which allows a lock-free synchronization. Each event has its own synchronization variable.

If the consumer thread has to wait, it is suspended by a `yield` command after each unsuccessful query so that it is returned to the OS scheduler. This is done to avoid that the thread utilizes the host unnecessarily by performing a busy-wait cycle. Measurements have shown that this method is significantly faster than releasing the host using a sleep or wait command. On the other hand, this method is less energy efficient than a sleep or wait command because the host is still fully utilized. Since performance is a higher ranked goal of this work, the `yield` method was chosen.

Since an event is bound to a context, it cannot be used for synchronization between different contexts. To solve this problem OpenCL user events are used in this work. User events are created and modified by the user, in contrast to normal events. At creation, a user event has the status `submitted`. This can be explicitly changed by the user to the state `complete`.

It needs two events to synchronize two commands running on different contexts. The producer command creates and controls a normal event. The consumer command gets a user event as input. A `callback` function responds after the normal event is set to `complete` by the producer. In this `callback` function, the user event from the consumer is set to `complete` so that the consumer can execute its command. Since `callback` functions require additional time, it must be avoided that an unnecessary amount of such functions is executed. Therefore, the `callback` function acts as a multicast that controls all consumer events that depend on a producer event.

Command Execution Order: Figure 4.13 shows the different operations and command queue commands that can occur during the execution of a message. There are five different types of command queues. Depending on the queue type, different operations can be performed. The operations with the dotted lines have suboperations, which are shown on the right side of the figure (blue boxes). All commands executed on the command queue are written in bold letters (green boxes). The producer/consumer synchronization points were explained in the previous part and are needed to synchronize events (orange boxes).

The first operation in all queues is a barrier that waits for all input events. Before an OpenCL command can be enqueued into a command queue, it must wait for the input events to be created (`sync consumer`). The OpenCL command used to enqueue the barrier is `clEnqueueWaitForEvents` (up to version 1.1) or `clEnqueueBarrierWithWaitList` (from version 1.2). The enqueue barrier command could also wait for the complete list of events and not for each event separately. However, this is not always possible since multiple barriers could be waiting for the same event and the enqueue barrier command needs a consecutive list of events.

Next, the `map`/`unmap` functions for the buffer are executed. The commands are only needed if arguments of the kernel must be switched between host and device space. In case there is an input event, it is also necessary to wait for the events to be created.

After queuing the `map`/`unmap` commands, the `set events` function first sets the synchronization point for the output event (`sync producer`). This allows waiting threads to enqueue their commands in their command queues. If the output event needs to be connected to other

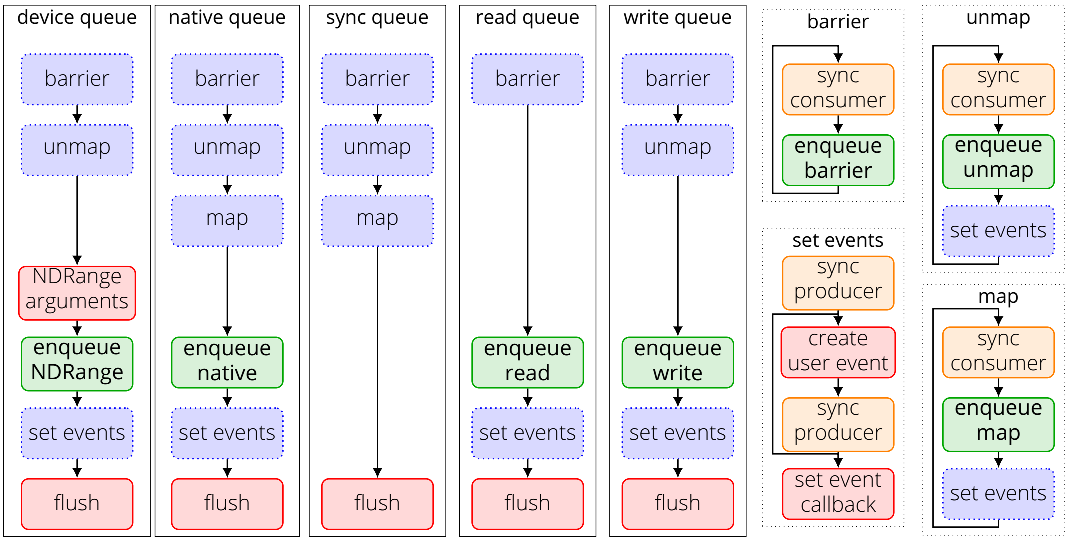


Figure 4.13: All operations processed during the execution of a message for the five queue types. Dotted lines mark that subfunctions exist (shown on the right). Bold letters mark enqueue commands for the command queue.

contexts, the required user events and their synchronization points are created. Finally, a `callback` function is created for all user events that should respond to the same event.

The `set events` function is also executed for all other enqueue commands shown in bold letters. For the NDRange command, however, the kernel arguments must be set in the OpenCL runtime (red boxes). Setting the kernel arguments cannot be done in the initialization phase, because the same kernel can be executed with different arguments. All enqueue commands are enqueued in a non-blocking mode in the command queue.

Finally, the enqueue commands are flushed to the command queue to ensure that they have been submitted (red boxes). The `set events` function is executed before flushing the command queue to allow faster synchronization between the functions. Timing measurements could not show whether it is faster to flush or to set the events first. After all commands of a message are enqueued in the command queue, the system waits for the next message. Additionally, a flag is set that indicates that the thread can be terminated.

4.6 Embedded System Vision Toolchain

The previous section introduced the High-Performance Vision toolchain. It uses OpenCL to distribute applications on x86-based heterogeneous architectures. Another part of the `DECISION` framework is the Embedded System Vision toolchain, which will be presented in this section. Both toolchains have a shared frontend, which is based on OpenVX and implemented by the OpenVX Graph Creation module from Section 4.2. One advantage is that the application implemented by the user is independent of the target platform. However, the Embedded System Vision toolchain aims to build a runtime-adaptive computer vision architecture. This architecture should be efficient, flexible, and capable of running multiple applications.

Another similarity between the two toolchains is that they use the `HiFlipVX` library to create computer vision IP-cores optimized for FPGAs. Like the High-Performance Vision toolchain, the self-optimized or auto-generated kernels that have been discussed in Section 4.3 and Section 4.4 can be integrated. The two toolchains differ in both the middleend and the backend. The Embedded System Vision toolchain uses the Application Distribution module (`APARMAP`) in the middleend. It distributes the application graph onto a partition-based mesh-like topology, thus creating a heterogeneous NoC-based architecture. This architecture is generated for the FPGA in the backend using the Hardware Creation module. It also includes the additional components, such as buffers, converters, routers, NIs, DMA controller and a MA that orchestrates everything.

The following subsection gives an overview of the toolchain and its modules. This is followed by a description of the various models, which are used as interfaces between the modules. Next, the additional hardware components are described. They are needed to build the adaptive architecture and are part of the Library module. Finally, the Hardware Creation module is described, which creates the final architecture based on the results of the application distribution process.

4.6.1 Module Overview

An overview of the Embedded System Vision toolchain is shown in Figure 4.14. The figure shows the order of execution of the different modules in the two different tool flows. The different modules can operate independently from each other to increase abstraction and interchangeability. Their inputs and outputs are represented by models and saved in files. First, the user implements an OpenVX application in C++ using the OpenVX Graph Creation module. This module takes the implemented application and creates the application model using the `HiFlipVX` library. Then the Application Distribution module maps & schedules the tasks of the application model to physical nodes, which are then clustered & placed into the platform model to create the architecture and execution models. If necessary, the user can restrict or precise the predefined platform model. Last, the Hardware Creation module creates the final system by adding all physical nodes, connecting them with each other and creating the bitstream.

The Library module uses `HiFlipVX` to implement various accelerators for object detection. It is integrated into the OpenVX Graph Creation module to create the tasks of the application model and the IP-cores that execute them. To create a runtime-adaptive, mesh-like, and partition-based system that uses a NoC as communication infrastructure, further nodes are needed.

- Routers, to set up the communication of the physical nodes between the different partitions within the NoC.
- NIs, to connect the routers with CUs (physical nodes).
- CUs, which can be ACs, PUs or I/Os.
- DMA controllers for direct access to global memory.
- MA, to orchestrate task flow and configure components, like NIs and DMA controllers.

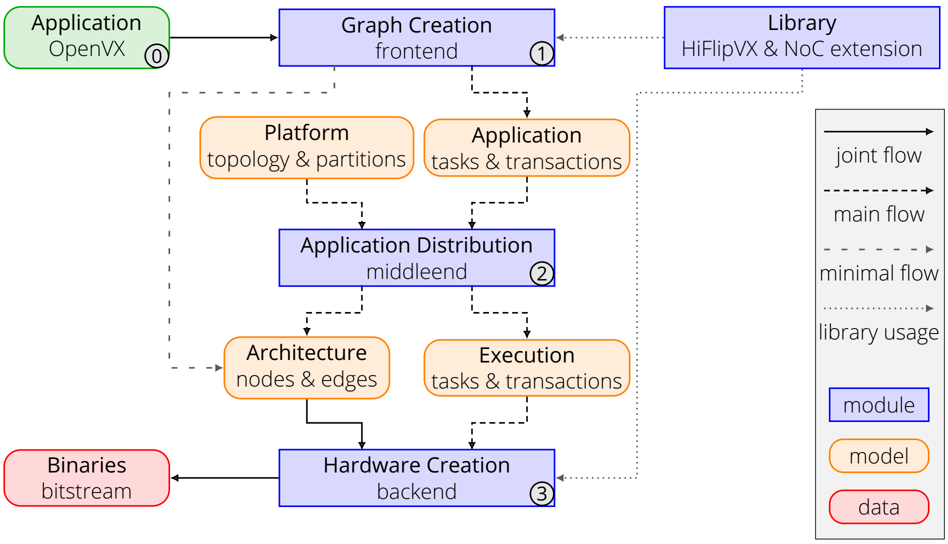


Figure 4.14: Overview and flow of the Embedded System Vision toolchain.

The OpenVX Graph Creation module abstracts away all hardware related code and gives the user an OpenVX conform interface to implement object detection applications for FPGA-based systems. First, the module creates a graph from the implemented application and verifies whether this graph is OpenVX-compatible. Then it synthesizes the utilized vision functions of the `HiFlipVX` library using wrappers that set the parameters and interfaces. The generated IP-Cores will be used by the Hardware Creation module for the final hardware architecture. At the end, the OpenVX Graph Creation module takes the task graph and calculates all parameters to create the application model for the Application Distribution module. These parameters also include the latencies and resource consumption of the synthesis results. Moreover, the module can bypass the Application Distribution module and create a simplified architecture model for the Hardware Creation module, which is a pure AC-based design.

The Application Distribution module distributes the partitioned application described by the OpenVX application model onto the platform model to generate the architecture and execution models. In the first step it maps the task graphs of the application model to physical nodes and creates a schedule. The physical nodes are clustered and placed into the partitions of the predefined platform model. The partition-based and mesh-like topology of this model can be irregular and heterogeneous. Each partition, or PRR, can have a different number of resources and can either be a region of an FPGA or an entire FPGA. Multiple physical nodes can be placed in a partition, depending on the available resources. These nodes (CUs) can be ACs, PUs, or I/Os.

Additionally, the user can add restrictions to this model if needed, e.g., place a node on a specific partition or execute a task on a specific node. The module outputs two models. The architecture model determines the placement of the various physical nodes inside the mesh-like topology and how they are interconnected. The execution model specifies the execution flow of the applications, consisting of tasks and transactions, which are stored in the MA and executed on the physical nodes.

The Hardware Creation module generates a runtime adaptive architecture, which uses a NoC as communication infrastructure, from the distribution results. To realize this architecture, several components have been designed and realized. They were implemented using HLS and will be explained in more detail later. These components include intelligent and runtime configurable NIs, MAs that configure the NIs and monitor the schedule, and DMA controller to directly access the global memory. The entire architecture is designed for computer vision applications with high data rates and low synchronization overhead. An advantage of the approach is the reusability of ACs and I/Os in different application scenarios, while preserving the performance of fully application-specific accelerator designs. In addition, the platform model opens the possibility for DPR to efficiently utilize FPGA resources. Alternatively, the Hardware Creation module can also create a pure AC design from the results of the OpenVX Graph Creation module.

First the module adds the existing repositories, consisting of routers and CUs. Then it creates the needed IP-cores for the NIs, MAs and DMA controllers. Then it generates the hardware block design using the Vivado tool from XILINX. Therein it adds all the physical nodes and connects them with each other. Additional system components, such as buffers and converters, are also added if needed.

In comparison to the overall picture of the `DECISION` framework of Figure 4.1, not all modules are needed in the Embedded System Vision toolchain. Since the current realization of the Hardware Creation module is limited to FPGAs, and the synthesis results and further estimations are generated in OpenVX Graph Creation module, no further Profiling module is required. Furthermore, no runtime module is needed to execute an application, since program flows are stored in the MA to control the runtime adaptive architecture.

4.6.2 Model Description

This subsection describes the various models needed for the Embedded System Vision toolchain and the Application Distribution module. It consists of an application model, a platform model and an architecture model as shown in Figure 4.15. Additionally, there is an execution model that describes the application running on the architecture.

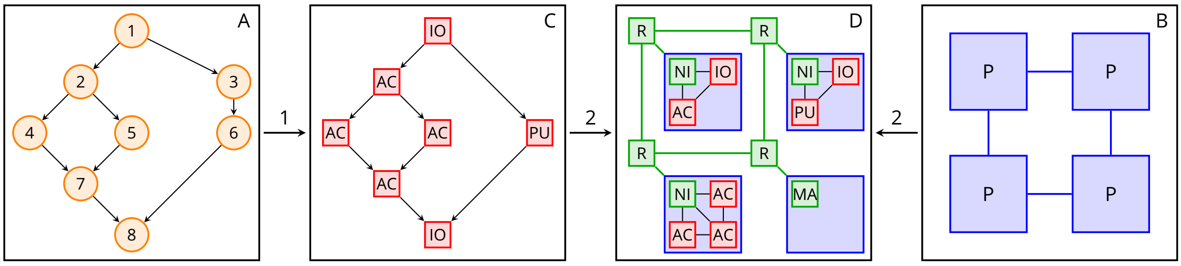


Figure 4.15: Application distribution model for mesh-like partition based architectures. Application task graph (A) is mapped & scheduled (1) to a node graph (C), which is clustered & placed (2) into platform model (B) to create final architecture (D) containing additional components. AC (Accelerator), I/O (Input/Output), PU (Processing Unit), NI (Network Interface), MA (Manager), R (Router), P (Partition).

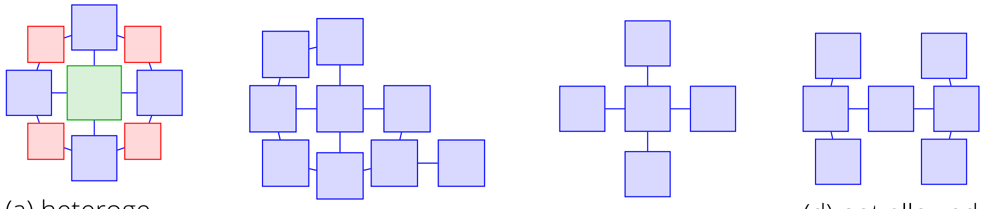
Application Model: The application model is described as a task graph, which is a data flow graph, containing tasks and transactions. The required parameters of the application model

(Figure 4.15.A) are generated in the OpenVX Graph Creation module and are partly based on the synthesis results.

- *Tasks* are functions of the input application that are executed on a node (CU) and can have input and output streams of data. As information they require:
  - The resources for each resource type. The number of resource types is determined in the platform model. For the application distribution algorithm, it does not matter whether these are FPGA resources, such as LUTs or FFs, or a different type of resource.
  - The latency in clock cycles for each CU type on which the task can be executed. The CU types are determined in the platform model. One of these CU types is the generated AC from the OpenVX Graph Creation module. Another CU type could be a PU or an I/O.
  - The IP name for the Hardware Creation module.
- *Transactions* are data messages which are sent between two tasks and can contain a single variable or a complete image. As information they require:
  - The IDs and ports of the sender and receiver tasks, to connect them with each other in the final hardware design.
  - The total latency of the transaction.
  - The size of an element (bit-width) and the total amount of elements, to calculate the average bandwidth consumption.
  - The latency offsets in relation to the start of sender and receiver tasks, to estimate the buffer sizes and average bandwidth consumption at a given time.

Platform Model: In addition, each task and transaction requires a unique ID. The platform model (Figure 4.15.B) describes of a 2D mesh-like topology consisting of partitions.

- The *topology* can be homogeneous or heterogeneous since different types of links (bandwidth) and partitions (resources) are possible (Figure 4.16). Additionally, not only regular (rectangular) 2D mesh-like topologies are supported. The only limitation is that no empty spaces are allowed within a single row or column of partitions, due to the adapted XY-routing algorithm.
- The *partitions* are PRRs and can be complete FPGAs or regions on FPGAs, which can differ in the number of available resources.



(a) heterogeneous

(b) irregular

(c) irregular

(d) not allowed

Figure 4.16: Example platform model topologies. A row or column is not allowed to have an empty space.

Architecture Model: Task graphs are first mapped and scheduled to a set of CUs, which create the node graph (Figure 4.15.C). This graph is then clustered and placed into the platform model to create the architecture model (Figure 4.15.D). This model mainly consists of physical nodes and edges.

Nodes are IP-cores that can be placed on the FPGA. Therefore, it stores its unique IP-name and the partition it should be placed on. There are different types of nodes:

- CUs can be placed on partitions and run multiple tasks in sequential order. The resource consumption of a CU is either fix for PUs and I/Os, or the maximum of all ACs mapped to that node. In addition to the resource consumption, the maximum available bandwidth must be defined.
  - PUs can be a processing architecture, like RISC-V [274] or MicroBlaze [275]. They can execute different tasks and, unlike ACs, delay the start time of a task execution.
  - ACs are custom IP-cores, like in `HiFlipVX`. They can either perform the same task multiple times or perform different tasks multiple times by using DPR. Compared to PUs, they can only react to their input.
  - I/Os can be, for example, a DMA or an HDMI that can be used to communicate externally.
- NIs can connect multiple CUs to a router and schedule transactions between the CUs and the router. They synchronize ACs with each other, avoid deadlocks and are configurable at runtime.
- MAs control the program flow and configure DMA controller and NI nodes, to start tasks including their transactions. It is possible to have multiple MAs in one system to counteract possible bottlenecks.
- DMA controllers are configured by the MAs to give CUs direct access to main memory.
- Routers can transfer data between different partitions.

The NI, MA and DMA controller nodes are part of the NoC extension of the Library module shown in Figure 4.14 and have been implemented using HLS.

Edges are connections between nodes and transfer transactions. They are full duplex when connected to a router. An edge can be off-chip when connected to a router. It needs the IDs and ports of the sender and receiver node. Additionally, its bit-width and buffer (FIFO) size, if connected to a CU. Buffers are needed to increase the throughput and prevent from deadlocks. The needed amount of buffer elements is calculated in the OpenVX Graph Creation module or Application Distribution module.

The architecture has a static and a dynamic part. The static part contains routers and their interconnections, as well as controllers and decouplers needed for DPR. All other units can be placed on a static or dynamic region (partition). This includes CUs, NIs, MAs, DMA controllers, buffers, and converters.

To provide the user with a more granular control over the final architecture, there are a few additional configuration options, which are not mandatory. They can be useful, for example, if the user wants I/Os to be placed on certain partitions. Or whether a fixed architecture, such as a GPU, should be connected to the remaining architecture via the NoC.

- A maximum buffer size can be defined for all FIFO components used to connect CUs.

- CU types can be predefined for PUs and I/Os.
- CUs can be predefined, and certain tasks can be bound to them.
- CUs can be bound to specific partitions.
- Partitions can be blocked for CUs that are not bound to them.

Execution Model: The execution model describes the flow of the applications (OpenVX graphs), with respect to the final architecture, including their tasks and transactions. It is stored on the MA node and used to configure the NIs and DMA controllers to control the application flow. More details are described in the next subsection together with the MA node.

4.6.3 Library Module

The final hardware architecture requires several components to create a runtime-adaptive system in addition to the CUs. For the communication between different partitions, the routers of RAR-NOC [65] are used . Additionally, three different components have been developed for the final architecture.

- Programmable intelligent NIs that can connect multiple components to a router to process or create the routing information.
- DMA controller, which provide direct access to main memory by the various CUs.
- MAs to control and configure the mentioned components, and to orchestrate and schedule the execution of the applications.

Message Types and Interfaces

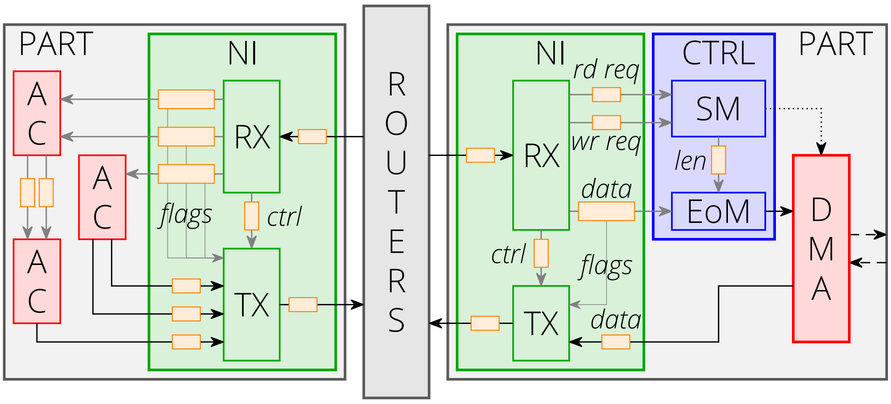


Figure 4.17: Example architecture, which shows the use of the NIs (Network Interfaces) and the DMA controller (CTRL) on the basis of two partitions (PART). Further partitions and the router connections are omitted for simplicity. AC (Accelerator), EoM (End of Message), SM (State Machine).

These components communicate with each other via control messages: to synchronize, to maintain the schedule and to avoid deadlocks. Each message begins with a header flit containing the *x* and *y* coordinates of the router and the output port of the RX unit, which is part of the NI, as shown in Figure 4.17. The remaining data flits of the message contain the control data, as shown in Figure 4.18. The bit-widths of the different fields in the flit are parameterizable. They depend on the bit-width of the NoC, its maximum size in *x* and *y* direction, and the maximum number of ports of the RX and TX units.

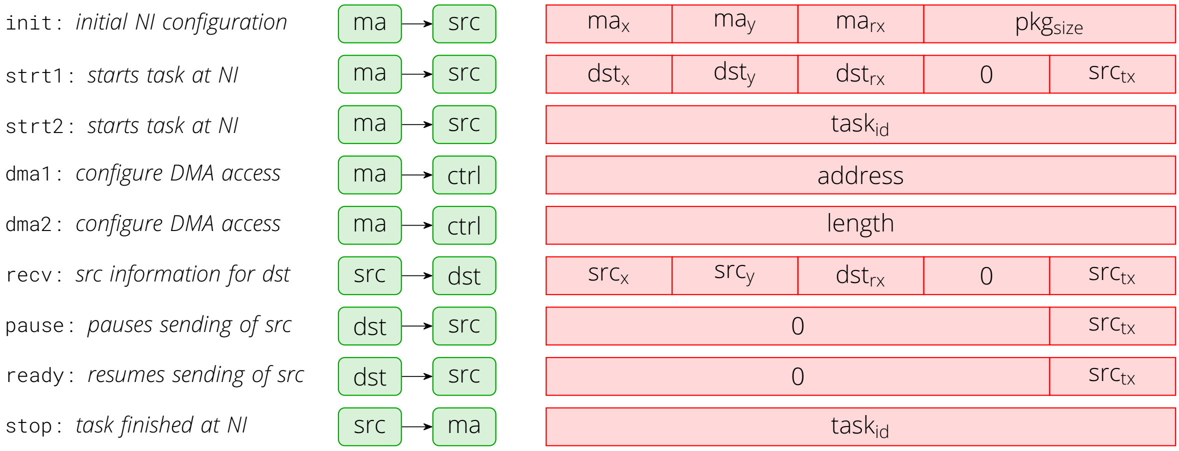

init: initial NI configuration | ma → src | ma_x | ma_y | ma_{rx} | pkg_{size}
strt1: starts task at NI | ma → src | dst_x | dst_y | dst_{rx} | 0 | src_{tx}
strt2: starts task at NI | ma → src | $task_{id}$
dma1: configure DMA access | ma → ctrl | address
dma2: configure DMA access | ma → ctrl | length
recv: src information for dst | src → dst | src_x | src_y | dst_{rx} | 0 | src_{tx}
pause: pauses sending of src | dst → src | 0 | src_{tx}
ready: resumes sending of src | dst → src | 0 | src_{tx}
stop: task finished at NI | src → ma | $task_{id}$

Figure 4.18: Control flits transferred between MA, TX and DMA control units. src/dst (source/destination of transaction), x/y (coordinates of router), rx/tx (receive/-transmit port number of NI), ctrl (DMA controller), pkg_{size} (maximum package size transferred over the NoC).

Most units in the architecture send data using the AXI4-stream protocol, which is represented by the solid lines in Figure 4.17. This is a simple protocol that includes the `valid` and `ready` bits for handshaking, as well as the data bits. A `last` bit, which is optional, is used to determine the end of a package or the end of a message. For all signals where it is not necessary, it is not included to save resources (gray arrows). Between routers, or routers and NIs the `last` signal is used for the end of a package. Between CUs, or CUs and NIs the `last` signal is used for the end of a message. A message is split into multiple packages within the wormhole NoC to avoid high latency. There are other optional signals in the AXI4-stream protocol, but they are neither needed nor used in this work. The AXI4-lite protocol, represented by the dotted lines of Figure 4.17, is needed to configure registers within the IP-cores. The MA is configured by the ARM and the DMA by its controller. The DMA is connected to the DDR controller via the AXI4-full protocol, which enables burst reads and writes (dashed lines).

Manager

On system setup the MA sends an `init` message to all NIs (Figure 4.15). It contains the coordinates of the MA and the maximum $package_{size}$ for data transfer, which can be send over the NoC. All messages that are larger are automatically split into multiple packages by the NIs.

The MA contains lists for the coordinates of all partitions and the available applications including their tasks, transactions and memory accesses. Each application points to its first and last task in a task list. The tasks store the number of transactions, their NI-ID, and the direct predecessor and successor tasks. Each task points to its first element in a transaction list. The transactions store their receiver coordinates and transmitter port. If a transaction includes a memory access it points to its entry in the memory access list. This entry stores the address and length of the transaction, the DMA controller port for the request, and if it is a read or write transfer.

The user can start an application by sending the application ID and a `start` bit to the MA over its AXI4-lite port. After an application is started, the MA iterates through the list of tasks that belong to the application until it reaches a barrier. A barrier can be reached if a task needs to wait until its predecessor task, which is executed on the same CU, is finished. Tasks mapped to different CUs are executed in parallel if possible. Since all CUs are streaming capable, they start processing a task as soon as data is available.

For each new task, the MA sends a message to the corresponding NI, containing the `strt1` and `strt2` flits for each transaction contained in this task. `strt1` contains the destination coordinates (x, y, rx_{port}) of a transaction, which is send from the tx_{port}. `strt2` contains the $task_{id}$ of that transaction. If the MA reaches a barrier, it waits for the `stop` messages coming from the NI containing this $task_{id}$. A `stop` message is sent by the NI after a transaction has finished. If all transactions of a task have finished, the successor task of the same node can be executed, and the MA continues iterating over the task list. A `done` bit is set in the MA when the application is finished, to inform the user.

Network Interface

The NI consists of an RX and a TX unit, as shown in Figure 4.17. If the RX unit receives a package, it first takes the header flit to configure the output port and then sends all following flits of the same package to this port. All control messages are sent to the TX unit. The RX unit is connected to its CUs via BRAM-based FIFO units to maximize the throughput. All other buffers in the NI only need small LUT-based FIFO units, to maintain the data-flow. The output ports of the RX units or the input ports of the TX units can have different bit-widths, which need to be a multiple of two. Therefore, data-width converter units are used for down-conversion after a FIFO and for up-conversion before a FIFO connected to an RX unit. For the TX units it is exactly the opposite, since it receives the data. This approach was the most efficient one, which did not create an additional bottleneck for data-width conversion. Packing multiple data (e.g. pixels) into one flit, utilizes the NoC more efficiently. In most cases the NoC should have a higher data-width than the CUs, to not limit the performance of the application.

The TX unit transmits the transaction data and sends & receives the control messages. To transmit a transaction, it generates the header-flits and sets the `last` bit for the tail of a package. It stores the destination address of each input port to which a job was assigned via the control message from MAs. After receiving the `strt` flits for a transaction, it sends a `recv` message to its receiver. This message contains the sender coordinates and the receivers rx_{port}. The TX unit iterates over all input ports, which are not connected to the RX unit, in a round robin manner. It selects a port if a job was assigned to it and the input FIFO is not empty. Then it sends a package of maximum $package_{size}$ data flits to the receiver. The package can be smaller if it receives a `last` signal at its input port indicating the end of a message (transaction).

Additionally, the TX unit monitors the programmable almost empty ($empty_a$) and almost full ($full_a$) flags of the FIFO units, which are located between the RX unit and the CUs. If a $full_a$ flag is set, a `pause` message will be sent to the sender of the transaction. The sender will then pause the corresponding job until it gets a `ready` message, which is send after the FIFO $empty_a$ flag is set. To maintain a high performance, these flags could be set, for example, to 75 % of the FIFO elements for $full_a$ and 25 % for $empty_a$. The `pause` and `ready` messages are used to prevent deadlocks, which can occur due to a high congestion of the NoC. During evaluation, the flag signals of the XILINX FIFO units were not stable in the first clock cycles after a reset signal. Therefore, its port must be ignored for the first few clock cycles after a system reset.

DMA Controller

If a transaction requires memory access, a message is sent by the MA to the DMA controller, in addition to the configuration message for the TX unit. Its two data flits contain the start address of a memory access and its length in bytes. The controller can configure the DMA for both read and write accesses. As shown in Figure 4.17, it has separate ports for read and write requests, since the DMA can send and receive data in parallel. With only one request port, it could happen that a longer read request blocks several write requests or the other way around.

For a read request, the data is sent directly from the main memory to the receiver via the DMA and NoC. For a write request, the data is sent directly from the sender via the NoC and DMA to the main memory. Based on the length of a write request, the DMA controller sets the `last` signal so that the DMA can accept a new request. This additional block is used, because the `last` signal is already used for the individual packages of a message. An alternative would be to send an additional flit via the NoC, which marks the end of a message. However, data flits would need to be buffered, which increases resources and latency, since they would arrive after the last data flit of a message. Additionally, it would increase the number of flits sent. There is no additional FIFO after the EoM (End of Message) unit since data is already buffered before this unit and within the DMA.

The internal state machine of the controller can accept new read and write requests in parallel but can only configure one at a time. After getting a new requests it reads the corresponding control registers of the DMA. When the DMA is ready for a new request and the previous request is done, its control registers are written. In addition, the length flit is sent to the EoM unit if it is a write request.

Unlike the other data ports of the RX unit, the flags of the request FIFO units are not monitored by the TX unit. This is because in the configuration of the proposed architecture, only a limited number of requests can be sent, which is also related to the maximum number of executable tasks within the TX unit. A higher number of buffered requests would only be possible if the TX unit could cache more requests. This could lead to a slight reduction in latency, but would increase the required resources, and increase the complexity of the scheduler inside of the MA. Therefore, only small FIFO units are needed.

Discussion

The proposed architecture focuses on image processing applications with large data transfers and heavy loads. With the multiport NI approach a high data throughput can be achieved. Due to the buffers and `pause` & `ready` messages, the overhead for synchronization can be reduced. The MA overhead is low, since it only maintains the schedule, by starting task transactions including memory accesses, and being notified when they finish. Another advantage is that ACs can easily be reused by different applications and send data to different locations.

A notable feature of the individual units described above is their modularity and their potential use in different architectures. Since the responsible MA of a NI can be updated at runtime, there are several possibilities for redundancy. In larger networks it would be possible to have several MAs for multiple applications. Depending on the load of the NoC, the MA can change the maximum package size send via the NIs. The DMA controller was designed in a generic way so that it can also be used in other types of architectures. The independent queue for read and write requests enables a high throughput. The controller does not have to be used only for external DDR memory but could also be used for a shared on-chip BRAM.

4.6.4 Hardware Creation Module

This module creates the final architecture and bitstream for the FPGA. The execution model and the architecture model generated in the Application Distribution module serve as input. In addition to the two models, the system configuration, which was also used in the OpenVX Graph Creation module, is imported. It contains the maximum package size that can be sent via the NoC, the target clock period and the FPGA part number.

The module currently supports two types of XILINX platforms (Zynq-7000 SoC and Zynq Ultrascale+ MPSoC). The various identifiers and interface names for the individual platforms are each contained in a separate `struct`. This simplifies the incorporation of further platforms. Due to the programming of the DMA controller by the MA, the architecture itself is not bound to SoC designs. There is only one AXI4-lite connection between the ARM and the MA. This can be exchanged by three signals: (1) application ID, (2) start bit and (3) ready bit. The only other external links are the signals for the clock, the reset, and the DDR memory controller.

In the first step, the additional components needed for the NoC-based architecture are created using Vivado HLS. This step is skipped if the Application Distribution module was bypassed, and the OpenVX Graph Creation module created a simple AC-based design. The required components are the RX and TX units of the NIs, the MAs, and the state machine and EoM units of the DMA controller. When creating the wrapper for the MA, its program memory is created, which contains the flow of the applications (execution model). This memory contains the MA and NI coordinates, and the applications with its tasks, transactions, and memory accesses. Depending on the size of each list, either LUT memory or BRAM memory is used for them separately. To generate the IP cores, C++ wrappers are created in which the parameters and interfaces are specified. A TCL script takes the file containing the wrappers and creates a solution for each component. The different solutions are synthesized in parallel and exported as IP-core.

In the second step, the architecture is constructed. Therefore, a TCL script is generated, which creates a Vivado project for the selected FPGA. It loads all repositories and adds all IP-cores, which were created in OpenVX Graph Creation module and Hardware Creation module. If necessary, it adds data-width converters and buffers when connecting the components. It also sets buffer sizes, buffer types (BRAM, LUT memory), flags (programmable empty & full), packet mode, and a `last` signal if needed.

To connect the DMA IPs with the memory controller of the ARM, `smartconnects` are used. One `smartconnect` for each memory port. Due to the number of limited high-performance ports to the DDR memory and the maximum number of `smartconnect` ports, the number of DMA IPs is limited. Since each `smartconnect` has a maximum of 16 single direction ports, a maximum of 32 DMA blocks, which can both read and write, are possible for four DDR memory ports. The DMA IPs are distributed evenly over the `smartconnects`. Names and version numbers of all IP-cores from the vendor are stored in a configuration file. This makes it easier to adapt the design to new tool and IP-core versions. At the end of the TCL script, the address space is set and the individual steps to create the bitstream are performed.

To debug the architecture, a simulation design can be created automatically by setting a flag. It replaces the connection between the ARM and MA by the three signals described above. It relocates the clock and reset signals and adds a simple VHDL testbench. Additionally, a BRAM is used instead of the connection to the DDR memory and the `smartconnect` is replaced by a BRAM controller. However, care should be taken in the design that the BRAM is significantly smaller than the DDR.

4.7 Evaluation

Using OpenCL, developers can program different architectures, such as CPUs, GPUs and FPGAs using the same API. Section 4.7.1 compares the implementations of different optimization strategies on various architectures in terms of their performance and energy consumption [9]. For FPGAs, detailed measurements have been performed to create a roofline model and make accurate estimates of its performance to balance between memory and compute bound kernels. Section 4.7.2 evaluates the automatic detection of parallelizable regions, the generation of human-readable C code from LLVM-IR and the optimization of this for GPUs, which are done to simplify the programming of OpenCL kernel [28]. Thereby, the generated code has been examined and the performance gain has been analyzes for different devices.

This thesis proposes a toolchain for x86-based systems that can integrate the user implemented or automatically generated kernels for the different architectures [27]. This toolchain performs heterogeneous scheduling and program generation, implements a performance optimized OpenCL runtime system, and integrates vendor libraries. Section 4.7.3 examines this toolchain and its individual modules with respect to their performance and overhead. The toolchain is part of a larger framework, called `DECISION`, which uses OpenVX as a common frontend and integrates `HiFlipVX` as a library for XILINX FPGAs. The second toolchain of this framework aims at designing an application specific architecture on FPGA-based systems. It automatically creates an adaptive NoC-based architecture developed from a dataflow-based application graph. Section 4.7.4 evaluates this system and its HLS-based components in terms of its overhead, resource consumption and performance. Both toolchains have an OpenVX-based frontend that integrates the `HiFlipVX` library evaluated previously. The second toolchain uses `APARMAP`, which will described in the next chapter, to distribute the application graph on a mesh-like and FPGA-based topology.

4.7.1 Architecture Dependent OpenCL Kernel Optimizations

Section 4.3 described the optimization and implementation of vision kernels for different OpenCL-capable devices. On the one hand to include other architectures and on the other hand to give developers the possibility to implement their own OpenCL-based kernels. This subsection compares and evaluates these optimizations in terms of performance, resource utilization and energy consumption.

Table 4.2 shows the devices of the test system, including a CPU with integrated GPU, a dedicated GPU, and an FPGA. This evaluation includes devices with a similar technology process to have a fair comparison for the energy measurements. It uses the SDAccel 2016.1 toolchain from XILINX for the FPGA. A TCL script integrates the compilation of the OpenCL kernels for the different devices.

Table 4.2: Test system components.

Device	Vendor	Model	Technology [nm]	Bandwidth [GB s^{-1}]
FPGA	XILINX	Virtex-7 XC7VX690T	28	10.7
GPU	NVIDIA	GTX 780	28	288.0
CPU	Intel	Core-i7 4770k	22	25.6
iGPU	Intel	HD Graphics 4600	22	25.6

Table 4.3 shows the estimated FPGA resource utilization and the achieved memory bandwidth for an application that copies data between two locations in global memory. The implementation contains two kernels, one for reading and one for writing. Kernels stream data between each other using a FIFO and the `async_work_group_copy` command to maximize bandwidth usage. The input and output images consist of 2048 × 2048 pixels with a data width of 32 bit. The maximum bandwidth achieved is 83 % of the theoretical possible bandwidth from Table 4.2, which is very high.

Table 4.3: Resource usage and memory bandwidth for 1 read and 1 write DMA kernel.

	32 bit	64 bit	128 bit	256 bit	512 bit
FF	2095	2205	2357	3141	4713
LUT	2537	3128	3122	3700	4850
BRAM	12	16	24	48	92
Bandwidth [MB s^{-1}]	740	1450	3170	6290	8899

Table 4.4 shows the computation time of a Gaussian filter for different port widths and vectorization degrees. The resolution of the input and output images is 1280 × 720 with a data width of 16 bit. The estimated computation time of the kernel shown in Equation (4.10)

and a similar memory copy kernel used in Table 4.3 serve as roofline model. It includes the number of columns (I_C) and rows (I_R) of the image, the kernel radius (K_R) of the Gaussian filter, the vector size (V_S) and the pipeline depth (P_D).

Table 4.4: Compute time [ms] for different memory port widths of a 5 × 5 Gaussian kernel.

	No compute kernel	Gaussian	
		Vector 4	Vector 8
128 bit	0.829	1.686	1.688
256 bit	1.107	1.101	1.175
512 bit	0.829	0.836	1.171
Estimated	-	0.851	1.159

$$\text{Estimated computation time} = \frac{(I_R + K_R) \cdot \left\lceil \frac{I_C + K_R}{V_S} \right\rceil + P_D}{frequency} \tag{4.10}$$

The measured time approaches the roofline as expected, which makes the computation time of the functions predictable. The resource utilization increases with the bit-width because the compute kernel, and not the copy kernel, does the unpacking and packing of the data to maximize memory throughput. It takes about 15 µs to offload a kernel without any functionality and with only one kernel parameter to the device. Each additional kernel parameter increases the time by about 15 %.

Fusing multiple kernels (loops) into one, not only reduces the number of kernels, which the tool limits to ten, but also reduces the utilized resources. However, it is more challenging for the tool to meet the timing constraints when fusing loops. Most likely, this is due to the larger region that needs to be controlled by the FSM of the loop. SDAccel also provides the Vivado HLS tool flow that uses C++ instead of OpenCL for its kernels. This allows parallel execution of multiple functions in one kernel by enabling loop-level parallelism using their `dataflow` directive. Section 4.7.3 investigates this method when integrating the `HiFlipVX` library into the High-Performance Vision toolchain.

Figure 4.19 shows the computation time of the best optimization strategy for each device. The computation time is based on the average of 3072 runs, and the image resolution is 1280 × 720. The labels in the figure show the selected optimization strategy. The impact of an increasing window size is less than for the single-threaded CPU implementation. For the CPU, `opt1` achieves the best performance for larger kernels. This is due to the optimal utilization of the registers as sliding windows and the caches as line buffers. For small kernels like the FED or kernels with sparse input like the DoH and Scharr filters, `opt0`, which benefits greatly from auto-vectorization and compiler optimizations, achieves the best results. This shows the strength of the Intel compiler, which defeats the strategy of its own optimization guide [270].

For the GPU and integrated GPU, `opt3` shows the best results. This is because caching data in the read-only memory saves computational overhead compared to `opt2`, which explicitly loads data into local memory. `opt3` achieves a bandwidth of 409 GB s^{-1} on the GPU for a 7 × 7 kernel. This proves that the GPU caches the data, since the bandwidth is higher than the

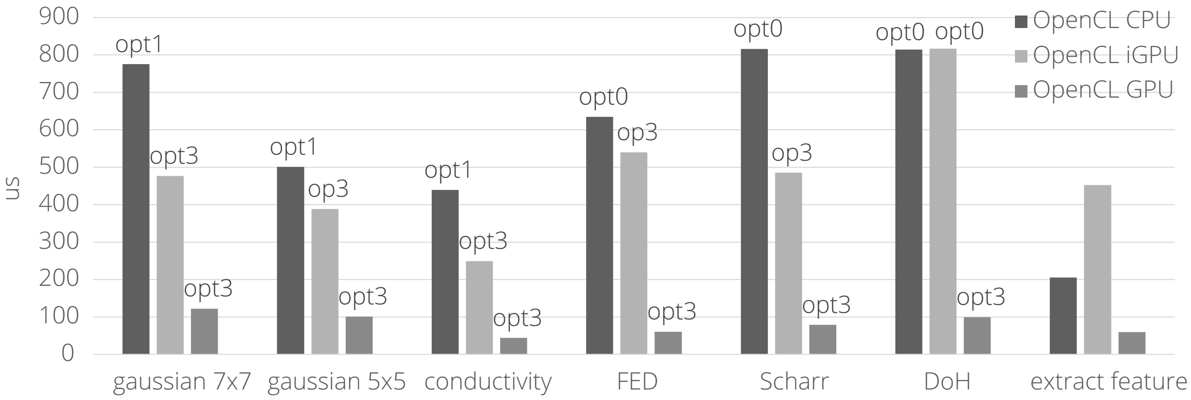


Figure 4.19: Computation time for different devices for a resolution of 1280 × 720.

theoretical one shown in Table 4.2. The sliding window of `opt3` or `opt1` additionally reduces memory access. The feature extraction function achieves a speedup of 3.88 on the CPU, 1.76 on the integrated GPU and 13.32 on the GPU, in comparison to the single thread CPU implementation.

Table 4.5 shows the results of the computer vision application chosen for evaluation. It is a partial OpenCL implementation of the AKAZE algorithm containing 27 vision functions. Figure 4.19 has already evaluated the functions individually for the CPU and GPU devices. The image resolution of the algorithm is 1280 × 720 (1920 × 1080). The feature comparison function remains on the host and needs 130 µs (255 µs) using OpenMP. The CPU can process this function in parallel to the execution on a dedicated device, such as a GPU or FPGA. The feature extraction and feature comparison functions are simplified implementations of the ones of the `HiFlipVX` library. The power consumption was measured for images with a resolution of 1280 × 720.

Table 4.5: Comparison of the implemented algorithm for different devices and resolutions.

Device	API	Resolution 16:9	Computation time [ms]	Speedup	Power [W]	Energy [mJ]
CPU	none	720p	80.22	1.00	23.77	1906.9
		1080p	183.48	1.00	-	-
CPU	OpenMP	720p	23.60	3.40	65.74	1527.9
		1080p	53.60	3.42	-	-
CPU	OpenCL	720p	15.57	5.15	66.86	1040.7
		1080p	37.69	4.87	-	-
iGPU	OpenCL	720p	13.26	6.05	29.31	388.6
		1080p	30.92	5.93	-	-
GPU	OpenCL	720p	2.135	37.58	222.69	475.4
		1080p	4.290	42.77	-	-
FPGA	OpenCL	720p	1.467	55.14	30.45	44.67
		1080p	2.923	62.77	-	- 1

The average time it takes to send an image to the devices and read back the approximate 4464 features, is 430 µs (770 µs) for the dedicated devices. The FPGA achieves the highest speedup and is 1.46 times faster than the GPU, due to the efficient use of its memory bandwidth enabled by streaming. The evaluation measures the entire system power with all its components since host and components like the DDR memory are also part of the execution. It measures power by executing the application for ten minutes, calculating the average power consumption, and subtracting the idle from the total consumption. The integrated GPU is only 1.17 times faster than the CPU, but 2.68 times more energy efficient, due to its parallel structure and lower frequency. The speedup of the FPGA increases with the resolution, showing its good scalability.

To check whether the application is memory bound or not, this work estimates the memory overhead of the application. One assumption here is that starting a new kernel takes longer than the overhead for filling the internal line buffers of the preceding kernel in the pipeline. The estimate is the sum of the ten kernels (165.5 µs) and the execution time of the last kernel. For a frequency of 200 MHz the application would need 1343 µs (2780 µs). The discrepancy to the values in the table is caused by the overhead for writing back the results. An own copy kernel for the output would optimize this but exceed the limit of the ten CUs the tool can manage.

The final FPGA design consumes 46.1 % LUTs, 19.6 % FFs, 8.1 % DSPs and 44.8 % BRAM. It achieves 342 fps for a resolution of 1920 × 1080. If copying data between host and device is not parallelized to the computation it would still achieve 271 fps. This measurement uses an accelerated OpenMP version of the FREAK descriptor. Section 3.3.2 shows the evaluation of this descriptor. The host executes the feature comparison function and the FREAK descriptor with 324 fps. A heterogeneous design could execute them on the CPU in parallel to the proposed implementation on the FPGA or GPU. The implementation thus achieves a higher performance than the related work in Table 3.16. However, most of the related work implement eight sublevels instead of four.

The following subsection will discuss the automatic generation of kernels using a source-to-source compiler. With the support of this compiler, it should be possible to reuse existing code without having to implement it from scratch, when using a different architecture. The generated code should be as human-readable as possible, to apply own optimizations, as described in this subsection.

4.7.2 Automatic OpenCL Code Generation

The last subsection evaluated the hand optimized implementations of OpenCL kernel for different devices and vendors. This should not only show the differences of the architectures in terms of optimization strategy, performance, and energy consumption, but also assist with the implementation of own kernels. This subsection will evaluate the source-to-source compilation toolchain explained in Section 4.4, to automatize a part of the OpenCL kernel creation. The compiler toolchain contains Polly and PPCG to generate source code from LLVM-IR using polyhedral optimization. Polly detects suitable code sections in the LLVM-IR. PPCG uses this information and optimizes it for the target architecture.

This work evaluated the source-to-source compilation toolchain using a variety of examples. One of these examples is the matrix multiplication function of Listing 4.5. The source-to-source compiler successfully built the valid OpenCL kernel shown in Listing 4.6. Additionally,

```
void matmul(int n, int m, int o, float A[n][o], float B[o][m], float C[n][m]) {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      float sum = 0;
      for(int k = 0; k < o; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
```

Listing 4.5: Example C++ matrix multiplication function.

it inserts comments, which contain information about the origin of the source code, above the header. The parameters are the same as in the original code. It marks memory pointers with `MemRef`. The example does not use a tiling strategy to avoid overloading the code in the listing. However, the use of tiles is possible, as the performance results will show.

PPCG adds defines, barriers, and additional iteration variables, and gets the thread IDs to generate the OpenCL kernels. The proposed `CWriter` class created the statements, which are the lines of code inside the for and if statements, from the LLVM-IR code. The variables of `private_MemRef_C` show that the tool has successfully transformed the `PHI` statements into local variables. Therefore, it does not require additional access to global memory or variables. In addition, the evaluation verified that the compiler processes C++ templates or regions consisting of multiple basic blocks.

One contribution of this work is the conversion of the existing PPCG integration in Polly to a newer version that uses the "Live Range Reordering" feature. Another contribution is the generation of valid and human-readable C source code from LLVM-IR and the correct generation of OpenCL host and device code. This approach can transform code from LLVM-IR, which provides the possibility to use different programming languages. PPCG, on the other hand, uses the AST of the Clang frontend and can only manage C code. This makes some analyzes easier to implement because the LLVM toolchain provides several capabilities, such as pointer alias analyzes, that are difficult to implement on an AST. The approach allows for better debugging, as it is easier to detect compiler optimizations in source code than in LLVM-IR. In addition, the developer can use any available OpenCL compiler if it supports the kernel version. Whereby the supported versions of kernel and host code in OpenCL can differ. Otherwise, the developer would need a backend for each architecture from each vendor. Furthermore, some analysis is easier with Polly, such as detecting loops that do not have a standard structure.

Figure 4.20 shows the performance capability of the generated matrix multiplication of Listing 4.5 for different matrix sizes. The test system consists of two CPUs (Xeon E5-2690 v2) with ten cores (20 threads) each and one GPU (Tesla K20c). For evaluation, the compiler generated OpenCL kernel with and without a local memory approach for both GPU and CPU. An implementation which uses OpenMP directives serves as a comparison. The figure shows the speedup for the different settings in comparison to the sequential execution of the matrix multiplication. The measurement only considers kernel execution times and no overhead for initialization or data transfer. However, the effort is less worthwhile for smaller matrices when considering the overhead. Unlike the CPU, the GPU can benefit from local memory usage for any test size. This is because GPUs have a much simpler structure and do not implement multilevel caches. For large amounts of data, the CPU also benefits from the local memory approach due to the limited cache memory.

```
// Function: matmul(), File: /home/test_polly_ppcg/matmul.cpp
// Line start: 56 ,end: 65
__kernel void kernel0(__global float *MemRef_C, __global float *MemRef_A,
  __global float *MemRef_B, int n, int m, int o) {

  int b0 = get_group_id(0), b1 = get_group_id(1);
  int t0 = get_local_id(0), t1 = get_local_id(1);
  float private_MemRef_C[1][2];

  #define ppcg_min(x,y)    ((x) < (y) ? (x) : (y))
  #define ppcg_fdiv_q(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
  for (int c0 = 0; c0 <= (n - 32*b0 - 1) / 8192; c0 += 1) {
    for (int c1 = 0; c1 <= (m - 32*b1 - 1) / 8192; c1 += 1) {
      if (n >= 32*b0 + t0 + 8192*c0 + 1 && m >= 32*b1 + t1 + 8192*c1 + 1) {
        private_MemRef_C[0][0] = 0;
        if (m >= 32*b1 + t1 + 8192*c1 + 17)
          private_MemRef_C[0][1] = 0;
        for (int c2 = 0; c2 <= ppcg_fdiv_q(o - 1, 32); c2 += 1) {
          for (int c3 = 0; c3 <= ppcg_min(31, o - 32*c2 - 1); c3 += 1) {
            private_MemRef_C[0][0] = private_MemRef_C[0][0] + (
              MemRef_A[(32*b0 + t0 + 8192*c0) * o + (32*c2 + c3)] *
                MemRef_B[(32*c2 + c3) * m + (32*b1 + t1 + 8192*c1)]);
            if (m >= 32*b1 + t1 + 8192*c1 + 17)
              private_MemRef_C[0][1] = private_MemRef_C[0][1] + (
                MemRef_A[(32*b0 + t0 + 8192*c0) * o + (32*c2 + c3)] *
                  MemRef_B[(32*c2 + c3) * m + (32*b1 + t1 + 8192*c1 + 16)]);
        } }
        MemRef_C[(32*b0 + t0 + 8192*c0) * m + (32*b1 + t1 + 8192*c1)] =
          private_MemRef_C[0][0];
        if (m >= 32*b1 + t1 + 8192*c1 + 17)
          MemRef_C[(32*b0 + t0 + 8192*c0) * m + (32*b1 + t1 + 8192*c1 + 16)] =
            private_MemRef_C[0][1];
      }
      barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
} } }
```

Listing 4.6: Generated OpenCL matrix multiplication kernel.

The next subsection will evaluate the proposed OpenCL-based toolchain for distributing OpenVX-based graphs on heterogeneous and x86-based systems consisting of GPUs, CPUs, and FPGAs. The toolchain can integrate both the automatically generated kernels from this subsection and the self-optimized kernels from the previous one to increase the number of available vision functions. In addition, the next subsection looks at the extraction of OpenCL kernels from existing libraries, such as OpenCV and AMDOVX.

4.7.3 High-Performance Vision Toolchain

The last two subsections focused on the evaluation of the OpenCL kernel generation. The first part evaluated different optimization strategies on different architectures and compared

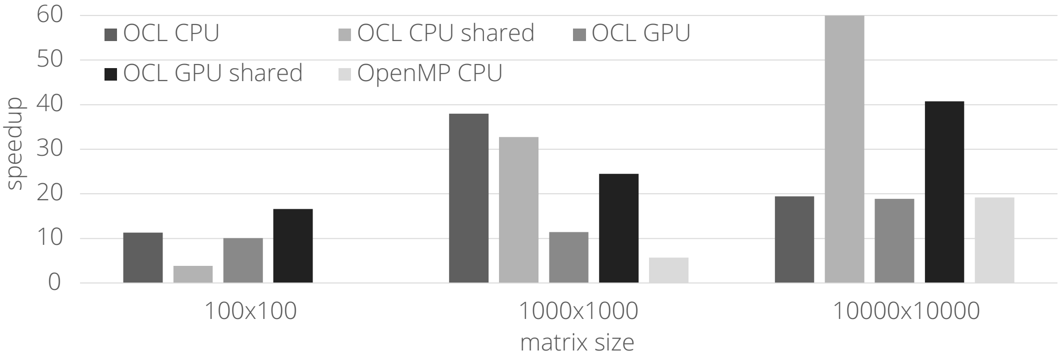


Figure 4.20: Matrix multiplication speedup compared to a single-threaded CPU execution.

them with each other. The second part evaluated the automatic generation of OpenCL kernels in terms of their performance and readability. This subsection evaluates the High-Performance Vision toolchain from Section 4.5, which further simplifies the implementation of heterogeneous architectures and enables the automatic distribution of applications on them.

The toolchain allows the user to implement OpenVX compliant applications for heterogeneous x86-based systems without knowledge about the underlying hardware. It distributes the application to the available devices, creates a runtime-optimized program including data transfers and synchronization mechanisms, and executes it. Its design is modular to allow adaptions to different architectures, programming models or applications. This simplifies the addition of new libraries, own kernel implementations, scheduling algorithms and APIs. The programmer can modify the models at design time, which provides an easy way to test, debug, or customize the application.

Setup: The evaluation verifies the efficiency of the framework and the individual modules, which operate in an automated manner and interact with the user only when explicitly wanted. Table 4.6 shows the devices used in the different measurements. It contains one GPU from the upper and one from the middle segment from each vendor. `GPU1` and `FPGA1` are part of one system, while the other devices are part of another one. The TDP (Thermal Design Power) and the technology process help to better interpret the results. The FPGA designs run at a clock frequency of 300 MHz. Due to limited support, the toolchain uses the Intel OpenCL driver to program the AMD CPU, since both are x86-based architectures. For all other devices, the toolchain uses the OpenCL drivers of the respective vendors. The toolchain runs the application 2048 times and takes the average of the 50th percentile to filter out variations caused by the operating system. The image resolution of the applications is 1920×1080, and the data width is 8 bit.

Profiling Module: Table 4.7 shows the calculated bandwidths and latencies of the Profiling module. It is important to emphasize that different vendors prefer different methods for memory transfer and memory allocation to achieve the maximum bandwidth. With the various OpenCL versions available, it is a complex task to choose the optimal method for each vendor. The device profiling in the table illustrates the meaning of this measurement, which can change depending on the method, driver, and tool version. To measure actual bandwidth and latency of FPGAs, the toolchain would need to generate a bitstream containing a bandwidth measurement kernel for each device. To prevent this, the toolchain integrates the xbutil program from XILINX, which contains pregenerated kernels and methods for

Table 4.6: Devices used in the different measurements.

ID	Device name	TDP [W]	Technology [nm]
`CPU1`	AMD 3900X	105	7
`FPGA1`	XILINX Alveo U50	75	16
`GPU1`	NVIDIA 1080TI	250	16
`GPU2`	AMD 5700XT	225	7
`GPU3`	AMD 560RX	60	14
`GPU4`	NVIDIA GTX 1650	75	12

measuring bandwidth. CPUs and GPUs can also compile their kernels at runtime, unlike FPGAs. Measurements have shown that the selected method (runtime or design time) has no influence on the execution time of the kernel. The toolchain already creates most binaries in the Profiling module to determine their execution time. It creates the FPGA binaries based on the mapping, as it may need to create a dataflow region with multiple image processing functions in one kernel.

Table 4.7: Device profiling results calculated with Equation (4.7).

		Host to device		Device to host	
Device	Theoretical bandwidth [GB s^{-1}]	Measured bandwidth [GB s^{-1}]	Measured latency [μs]	Measured bandwidth [GB s^{-1}]	Measured latency [μs]
GPU1	15.754	13.036	4	5.412	4
FPGA1	15.754	11.384	11	6.892	12

Library Module: Table 4.8 compares the execution time of the integrated libraries on different devices. A single function without vectorization requires only a few resources on the FPGA, which is why it has a comparatively low performance here. The executed function has only a small influence on its execution time compared to other devices. The reason for this is that the execution time depends mainly on the image size and frequency ($1920 \cdot 1080 \cdot 3.33\text{ns} \cdot 10^{-3} \approx 6905\mu\text{s}$) and increases only slightly with the kernel size of the windowed functions. Measurements show an additional dependency on the number of inputs and outputs, without being memory bound. Most likely, the AXI4 interfaces used, and the data movement system generated by Vitis cause this behavior, as it is a different approach than the one used by the Embedded System Vision toolchain. Reading data seems to have a slightly higher impact than writing data to memory.

The advantage of the base implementation is that it is portable and serves as a fallback solution. It also serves as a basis to evaluate the performance of the individual libraries. A comparison between the GPUs shows that the acceleration of the kernels can be quite different. This shows that the performance of a kernel depends not only on the respective implementation, but also on the architecture and compiler. AMDOVX achieves, on average, 2.93 times the performance on the GPUs compared to the base implementation. More

Table 4.8: Comparison of devices and kernels without data transfer in [µs].

Device Library	`CPU2` Base	`GPU4` Base	`GPU1` Base	`GPU2` Base	`GPU2` AMDOVX	`GPU3` Base	`GPU3` AMDOVX	`FPGA1` `HiFlipVX`
Gaussian	719	128	49	97	46	225	116	7172
Median	2218	573	205	148	47	1495	140	7139
Sobel	932	131	56	107	51	251	173	7358
Magnitude	142	104	37	31	30	105	138	7454

complex functions like the median filter show an even better optimization, when using this library. OpenCV is 2.04 times faster than AMDOVX on the AMD GPUs for a Gaussian filter. This demonstrates the strength of OpenCV, as it does not limit its kernels to AMD GPUs. On `CPU2`, the OpenCV Gaussian filter is 4.11 times faster compared to the base implementation. The base implementation achieves similar performance as a simple OpenMP implementation on the same CPU.

FPGA Optimizations: Due to the unused resources on the FPGA, various optimizations can be applied to reduce its execution time.

- Pipelining: Since the FPGA can overlap consecutive functions, its runtime does not increase as fast as the GPU runtime for multiple consecutive tasks. There is at most a deviation of about 5 % to the latency estimation of the scheduling algorithm, including data transfers.
- Vectorization: The FPGA can also increase its SIMD-size at the cost of increased resource utilization. The vectorization factors of 2, 4 and 8 are available in `HiFlipVX`. Depending on vector size, data width, number of inputs and outputs, and available bandwidth, the function can become memory bound.
- Kernel size: Even though the OpenVX standard just defines 3×3 kernel sizes, larger ones are frequently used. For some windowed functions, two consecutive 3×3 kernels do the same as one 5×5 kernel. However, this can lead to a slightly lower accuracy. While the FPGA mainly requires more resources, the kernel size has a strong influence on the runtime of other devices.

Figure 4.21 shows the total execution time for several consecutive functions on `GPU1` and `FPGA1` to illustrate the mentioned optimizations. It does not include CPU measurements because its performance was not comparable. With a filter size of 5×5, GPU performance decreases. Kernel pipelining allows parallelization on FPGAs with only a small overhead for filling buffers. To pipeline multiple functions within a kernel, the toolchain uses HLS directives and automatically builds a single kernel according to the output of the scheduling algorithm. This is different to the Embedded System Vision toolchain, which creates an IP-core for each function of the dataflow graph and connects them via AXI4-stream ports. Each additional node executed sequentially in a pipeline increases the runtime of the FPGA by about 9µs. On the GPU, it needs the entire runtime per additional task since it already uses its full computing capacities for a single function. For a vectorization factor of 8, resource utilization only increases by about 1 % per kernel. When combining all three optimizations, the FPGA can outperform the GPU. While the quality of the GPU OpenCL kernel leaves room for optimizations, it highlights the FPGA as accelerator for image processing pipelines.

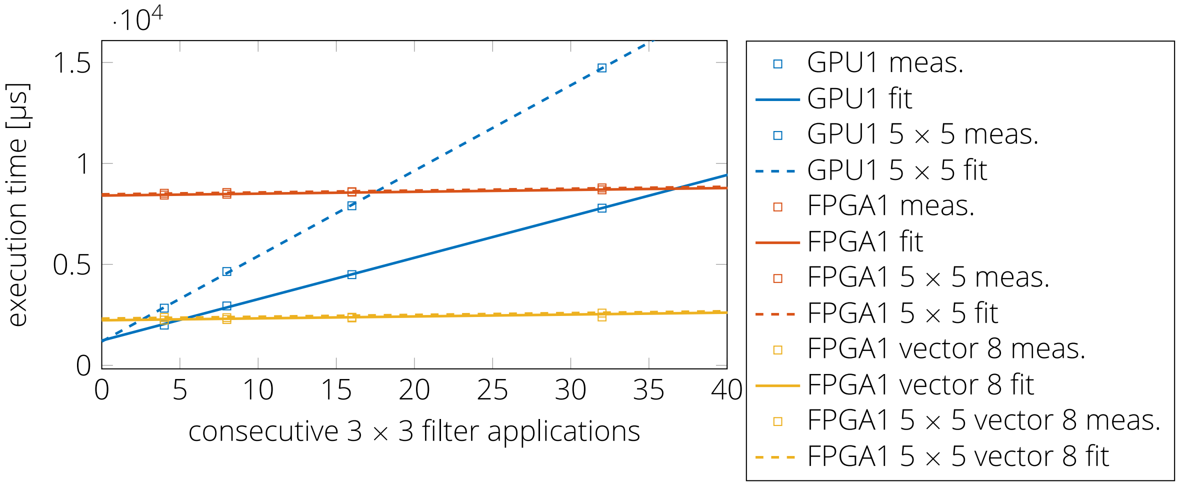


Figure 4.21: Median kernel pipelined and vectorized execution time for different devices including data transfer time. One 5 × 5 kernel equals two 3 × 3 kernels.

Program Creation and Runtime Modules: The Program Creation module precomputes all OpenCL command lists. The Runtime System module already includes them during compilation to minimize the additional overhead. Thus, OpenCL almost exclusively determines the overhead of the runtime system. Only the consumer/producer commands needed to wait for the creation of an event can in worst case produce an overhead of 100 ns to 200 ns by the `yield` command, according to measurements on CPU2. However, this is low compared to the OpenCL overhead. In all other cases, the program can enqueue the commands in the OpenCL command queues in advance.

Table 4.9 shows the average duration between the execution of two dependent functions, depending on whether they are on the same device or in the same context. It shows the overhead caused by the OpenCL runtime, including synchronization with (user) events and callback functions. The program takes the kernel start and end times of OpenCL events when profiling is enabled. The results show a non-negligible overhead between different devices, which complicates the scheduling of dependent tasks. It also takes some time to offload a kernel without considering its computation time, as shown by the overhead between two kernels on the same device. A copy between two GPUs of the same context has similar latency and performance as a read command. However, it has the advantage that it copies directly between two devices of the same context without going via the host memory. Transferring via host memory would require a read and a write command, which would approximately double the total latency. The `map` and `unmap` commands require at most 1 µs.

Table 4.9: OpenCL synchronization and kernel offloading overhead.

Device	Context	Time [µs]
Same	Same	3.7
Different	Same	56.9
Different	Different	128.2

Scheduling Module: This work evaluates the Mapping & Scheduling module using an ORB-like application, similar to Figure 3.11, which consists of 22 tasks. The application replaces some functions with similar ones, since some kernels are not available in any of the OpenCL libraries. The resulting graph contains several parallel and sequential parts of varying sizes. The baseline maps all tasks to `GPU1`. This work artificially reduced FPGA resources to force a heterogeneous schedule when including `FPGA1`. Using 3 × 3 kernels and a vectorization factor of 8 on the FPGA, the speedup is only marginal (1.01) when using `GPU1` and `FPGA1`. Without resource constraints, the toolchain schedules the entire graph on the FPGA, resulting in a speedup of 2.38. Using 7 × 7 kernels and a vectorization factor of 8 instead, the speedup of the heterogeneous schedule is 1.63. Without resource constraints the speedup is 13.39, executing the entire graph on the FPGA. Executing the graph with one CPU and one GPU merely reduces the total runtime by about 4 %, since it uses the CPU for only one task due to the performance differences.

The focus of the scheduling algorithm is on application performance and full utilization of FPGA resources. To create a representative schedule: the device profiles from Table 4.7, the kernel profiles from Table 4.8, the FPGA dataflow characteristics of Figure 4.21, and the OpenCL synchronization overhead of Table 4.9 need to be considered. Heterogeneous schedules can be seldom due to the widely varying execution times of different devices. High data transfer latencies compared to short function execution times can amplify this circumstance. In terms of energy consumption, FPGAs and integrated GPUs can further exploit their advantages, as shown in Section 4.7.1.

OpenCL: OpenCL provides a good base and low-level API to address as many devices as possible. Using native kernels, it can incorporate other C++-based parallelization methods for CPUs. This work could show with the help of `HiFlipVX` that it is possible to integrate kernels implemented in C++ for XILINX devices with SDAccel (Vitis). While the OpenCL overhead is not negligible, it allows the use of different vendor-optimized drivers in one program without having to use multiple programming languages or APIs. A driver update or using different OpenCL versions did not cause any problems. To further reduce synchronization overhead, the toolchain would have to replace the OpenCL environment with a custom implementation.

The next subsection evaluates the Embedded System Vision toolchain and discusses the advantages of the OpenVX Graph Creation module. It shares the frontend and `HiFlipVX` library with the High-Performance Vision toolchain, which is another advantage of the modular design. On the other hand, it uses a different architecture and topology, and therefore needs a different middleend and backend.

4.7.4 Embedded System Vision Toolchain

This subsection evaluates the Embedded System Vision toolchain from Section 4.6 and the generated architecture. This includes the OpenVX Graph Creation module, the Hardware Creation module and the additional HLS-based NoC components from the Library module. The OpenVX Graph Creation module allows the user to implement an application without having to deal with the underlying hardware or implementation details of the algorithm. It builds the application graph and uses `HiFlipVX` to build IP-cores and extract the necessary information. `APARMAP` creates an application-specific and adaptive NoC-based architecture from this graph. The Hardware Creation module creates and configures the components required for the NoC design, which include the TX and RX units of the NIs, the DMA controllers, and the MA. Alternatively, it can take the application graph and create a pure AC design.

Based on the final design, it connects all components and adds buffers, converters, DMA blocks, etc. and generates the bitstream.

The design of the toolchain and the modular structure of the framework have several advantages. The well-defined OpenVX interface, graph verification, default parameter setting, parameter propagation, and IP-core creation significantly speeds up the implementation of object detection algorithms. Thanks to the modular structure and well-defined models, customizations are easier to manage for both the user and the developer. This can be, for example, the addition of new library components, the development of new application distribution methods or the support of new architectures and platforms.

The total computation time of the toolchain mainly depends on the synthesis time of the different IP-cores and the final bitstream generation. Therefore, both modules use OpenMP with dynamic scheduling to generate all HLS IP-cores in parallel, which significantly speeds up the design process. For example, the tool needed only one minute to synthesize the 19 IP-cores of the ORB on an AMD 3900X CPU with 24 threads. The same design took four minutes when the tool synthesized a single IP-core containing all vision functions in a large dataflow region. For larger designs, the difference becomes more apparent. For the parallel synthesis of the IP-cores and the creation of the entire block design, including the automatic connection of the cores based on the graph description, the design of AKAZE needed about 32 minutes for 95 IP-cores and 43 minutes for 163 IP-cores. Packing all vision functions in one IP-core using the HLS `dataflow` directive had to be aborted.

Figure 4.22 shows a small architecture that demonstrates the benefits of the proposed toolchain and architecture. This architecture contains a 5 × 5 Gaussian (*G*), a 3 × 3 Sobel (*S*), a magnitude (*M*), a 7 × 7 segment test detector (*F*) and a 3 × 3 hysteresis (*H*) function. The segment test detector detects the corners of the FAST detector and outputs an image of response values. The hysteresis function, which is part of the Canny edge detector, has a lower and an upper threshold. The output value of an input pixel is true if the value is above the upper threshold (strong), or above the lower threshold (weak) and a strong pixel is in the neighborhood. The architecture in Figure 4.22 implements five different applications that share the mentioned functions:

- smoothing filter (G)
- segment test detector (F)
- smoothed gradient magnitude ($G + S + M$)
- edge detector ($S + M + H$)
- corner detector ($F + H$)

The Hardware Creation module allows developers to easily add new development boards and exchange or upgrade XILINX-specific IP-cores. It currently supports the PYNQ-Z1 and ZCU104 (3.9). However, designs should run without adjustments on any device of the Zynq SoC and Ultrascale+ MPSoC family. The test configuration for the following evaluation uses the ZCU104 and Vivado 2019.1.

- FPGA frequency: 100 MHz
- image dimensions: 1920 × 1080 × 8 bit
- node SIMD width: 1

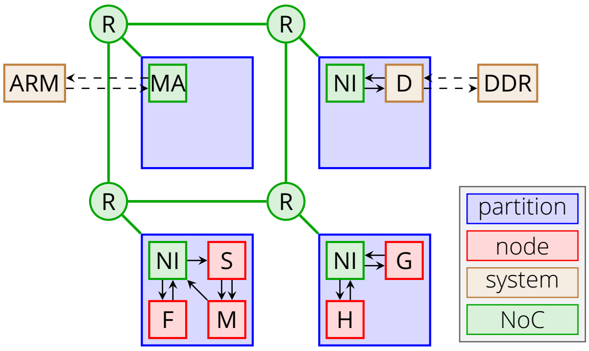


Figure 4.22: Embedded vision example architecture. R (Router), MA (Manager), NI (Network Interface), D (DMA), G (Gaussian), S (Sobel), M (Magnitude), F (Fast), H (Hysteresis).

- maximum number of data flits per package: 8
- NoC bit-width: 32

The first test evaluates the latency overhead of the proposed architecture against a base design that includes the five example applications. This base design does not have a NoC as a communication infrastructure, nor do the applications share functions or DMA blocks. The first test runs both architectures in the Vivado simulator without DMA or the proposed DMA controller. For a resolution of 1920 × 1080, the proposed architecture increases the latency between 107 and 168 clock cycles (133 on average), which corresponds to an overhead of 0.006 %. Time measurement starts when the MA receives the start signal to execute an application and ends when it receives the end signal. There are two reasons for the small increase in latency. The first reason is the initial configuration of the NIs by the MA. The second reason is the increased pipeline depth, since a pixel must additionally pass through the various routers and NIs.

The second test executes the proposed architecture in a real system setup. In this setup, the ARM CPU controls the MA and DMA and measures the time. Time measurement starts before the CPU sets the MA start signal and configures the DMA blocks. It ends after the CPU has detected the stop signal using a busy wait routine. The system runs all applications 100 times and selects the fastest execution of each application to minimize the impact of the CPU. The goal of this measurement is to determine the DMA and memory overhead. The latency of the real system increases by an additional 280 to 304 clock cycles (293 on average) compared to the simulated architecture. This is an additional latency overhead of 0.014 %. The result shows that there is no additional interference to the architecture when used outside of the simulation. The additional latency is due to the access time of the CPU, the access time to the main memory and the configuration time of the DMA blocks.

The third test adds the DMA controller to the design to evaluate its usefulness. This allows the MA to control the DMA directly and can thus save time. Compared to the system without a controller, it saves an average of 106 clock cycles. Thus, the additional overhead caused by CPU, DMA and memory access is only 187 clock cycles compared to the simulation. Another advantage of the DMA controller is that the system becomes more independent of the CPU, which also makes pure FPGA designs possible.

The previous examples did not use the caches. However, if the CPU executes an application in a hardware/software co-design, it may need to flush the caches. This process increases the latency by an average of 41 869 clock cycles, which means an increase in execution time by 2.026 % compared to the third test.

Table 4.10 divides the resource usage of the example architecture from Figure 4.22 into three parts. The NoC includes the NIs, routers, MA, DMA controllers, and all buffers and converters. The application nodes are the OpenVX-based vision functions, which have a very low resource consumption due to the efficiency of `HiFlipVX`. When designing bigger applications, the benefits of this architecture get more visible. The system part contains the interconnection networks required for the DMA and the peripherals, the DMA, and the reset signal. Since the proposed architecture reuses functions for different applications, it saves about 50 % of the resources for the application nodes. In addition, the NoC gives the opportunity to reuse the same DMA block for the different applications, which saves a lot of resources. One reason for the higher resource consumption of the DMA controller is the AXI4 interface generated with the HLS tool. The IP-core for the interface consumes 554 FFs, 628 LUTs, and 1 BRAM.

Table 4.10: Resource utilization of the implemented design for the embedded vision example application using the NoC-based architecture.

	LUT	FF	BRAM	DSP
Routers	1205	626	0	0
Manager	263	275	0	0
RX/TX units	1029	1794	0	0
DMA controller	803	1788	1	0
Buffers/converters	1972	3305	4	0
NoC	5272	7788	5	0
Gaussian	227	206	2	0
Sobel	237	219	1	0
Magnitude	230	183	0	1
Segment detector	1835	922	3	0
Hysteresis	120	159	1	0
Application nodes	2649	1689	7	1
System	5542	8215	5	0
Total	13 463	17 692	17	1

The next test increased the frequency to 300 MHz to evaluate the impact on execution time and resource consumption. For three times the frequency, the execution time increased by a factor of 2.9998. The reason for the small difference is that the frequency of the CPU and the DDR memory have not changed. The entire NoC architecture, including all buffers, converters, routers, NIs, MA, and DMA controllers, needed only two additional LUTs compared to the 100 MHz design.

The next test evaluates the maximum throughput of the router within the NoC by setting the data width of the application to 16 bit, which increases the amount of data transmitted in the NoC by a factor of two. The execution time of the applications increased by at least 50 % compared to the 8 bit application and by 64 % on average. This is because the router can send data at most every two clock cycles and has an additional overhead for sending the header flit and creating the route. Changing the data width of the application had no significant impact on the resource utilization of the NoC and the system-specific components. Increasing the package size minimizes the overhead by increasing the maximum throughput but decreasing the response time. Increasing the package size from 8 to 16 (32) increased the execution time by 32 % (16 %) on average and 25 % (12 %) on minimum. The minimum occurs when using an application with only one function, since the NoC utilization is lowest here.

The next test creates a design from the example application using the Embedded System Vision toolchain that contains only ACs. In this design, applications do not share nodes or DMA blocks. Table 4.11 shows the resource consumption for this system. The high additional resource consumption of the system is mainly due to the many interconnection networks and additional DMA blocks. There is one peripheral interconnect to control all the DMA blocks, four smart interconnects to connect to the memory controller, and five DMA blocks for the ten application nodes. Due to the 128 bit memory interface of the ZCU104, the DMA blocks use a relatively large amount of BRAM.

Table 4.11: Resource utilization of the implemented design for the embedded vision example application as pure AC design.

	LUT	FF	BRAM	DSP
Nodes	5279	3358	14	2
Buffers/converters	2295	5273	0	0
DMA blocks	10 970	16 290	25	0
Smart interconnects	10 388	17 240	0	0
Peripheral interconnect	1444	1477	0	0
Total	30 380	43 671	39	2

Both systems have their advantages and disadvantages. On the one hand, the pure AC design can run all applications in parallel. On the other hand, the NoC design requires significantly fewer resources. The MA and DMA controllers simplify the start of individual applications considerably, since the user only needs send the application ID and a start bit. A general advantage of OpenVX Graph Creation module is that it requires only 25 lines of code to create the AC design, including the five different applications. This is twelve lines of code for the nodes and their configuration, seven lines for the edges, and six lines for the rest of the OpenVX-based system.

4.8 Summary

This chapter presented a framework that consists of two toolchains that share a common frontend and `HiFlipVX` as library. Its design is modular to allow adaptions to different architectures, programming models or applications. This simplifies adding new libraries, scheduling algorithms and APIs. The programmer can modify the models at design time, which provides an easy way to evaluate, debug, or customize the application.

The joint frontend provides users with an OpenVX-compliant interface to implement computer vision applications. It is used for different platforms without needing to learn new concepts and languages or understand the underlying hardware and implementation. It checks the application graph and its parameters to detect possible user errors at an early stage. Default parameters are automatically set, and image properties are propagated through the graph to ease application development. Larger vision functions are divided into their sub-functions to achieve less fragmentation in the distribution of the application. It generates IP-cores and application model for the selected toolchain from synthesis and analytical estimates. To create synthesis results, wrapper functions are generated that instantiate `HiFlipVX` functions. The synthesis of individual IP-cores is parallelized to achieve a significantly faster flow compared to the advanced XILINX tools. A schedule is created to determine exact buffer sizes between nodes to avoid deadlocks and increase throughput.

Besides the integration of `HiFlipVX` for FPGAs, this work investigated the integration of existing libraries to support other devices. OpenCL has been used because it allows to address most architectures. To overcome vendor bindings, this work looked at the automatic extraction of OpenCL kernels for OpenVX functions. The AMDOVX library implements OpenCL kernels for their GPUs, while providing optimized x86 code for CPUs. To use the CPU code, the High-Performance Vision toolchain provides the possibility to run native C++ kernels. To create compatible GPU kernels, the AMDOVX code was adapted to extract control parameters and define them as internal variables within the kernel code. A similar approach was used to integrate functions of the OpenCV library, which is a good candidate due to its rich feature set, OpenCL acceleration, and vendor independence. One advantage of OpenCL is that it has a rich set of instructions, and thus requires only few vendor-specific constructs, such as XILINX directives or Intel intrinsic operations. However, C++ is better suited for libraries because of templates, as shown with `HiFlipVX`.

Since it is not possible to cover everything with libraries, this work explored the manual optimization of OpenCL kernel for various architectures [9]. For this purpose, four different optimization strategies for CPU and GPU devices have been implemented. Furthermore, this thesis provides a step-by-step optimization of windowed functions for the FPGA to maximize performance, minimize global memory access and eliminate bottlenecks. It uses loop pipelining, line buffers, sliding windows, array partitioning, vector operations, fixed-point numbers, data packing, streaming between functions, and separate memory access and computation in different kernel. In the evaluation, a roof line model was created for the FPGA to make the computation time of a kernel more predictable. Therefore, the achievable memory bandwidth was measured when using separate memory access kernels. In addition, the time required to offload a kernel to a device was measured, as well as the time required for each additional kernel parameter. The evaluation also shows the best optimization strategy of each vision function for each OpenCL device to make general assumptions about the best strategy. The FPGA achieves the highest performance for the test application and is 1.46 times faster than the GPU, which is due to the efficient use of its memory bandwidth by streaming data. Due to its low power consumption, it needs 10.6 times less energy than the GPU. The integrated GPU is only 1.17 times faster than the CPU, but 2.68 times more energy efficient, which is due to its parallel structure and lower frequency. The speedup of the FPGA increases with the resolution, which shows its good scalability compared to other devices.

This thesis developed a source-to-source compilation tool, which recognizes profitable program parts in C/C++ code and automatically generates OpenCL host and kernel code from it, to assist the implementation of custom kernels or acceleration of existing code [28]. The detection and transformation of suitable code is implemented in Polly using polyhedral optimization. This information is then transferred to PPCG and optimized for the target architecture. One contribution is the conversion of the existing PPCG integration in Polly to a newer version that uses the "Live Range Reordering" feature. In addition, a module to recognize and convert local variables has been created. Furthermore, the generation of valid and human-readable C code from LLVM-IR. The proposed approach opens the possibility to easier debug optimization passes or to use existing OpenCL drivers and toolchains without creating and updating a backend. It also provides the possibility to use other programming languages than C as input. In addition, some analysis is easier, such as detecting loops that do not have a standard structure, and own optimizations can be made to the code. The proposed source-to-source compiler successfully built an OpenCL kernel and transforms `PHI` statements into local variables. The generated code achieves a speedup of up to 60 on a dual CPU system with 20 cores (40 threads) and the tiling optimization strategy enabled.

The High-Performance Vision toolchain uses OpenCL as a low-level API to distribute and execute an application implemented in OpenVX on heterogeneous x86-based systems consisting of CPU, GPUs, and FPGAs [27]. The modules of the toolchain are independent from each other and can be used in a larger scope. For example, the Program Creation module and Runtime System module also work for non-streaming applications. In addition to the integration of `HiFlipVX`, OpenCL functions are automatically scanned to be accessible to the user. Thus libraries, self-optimized kernels, or even automatically generated kernels can be integrated more easily. To build a heterogeneous schedule from an application graph, the device and kernel profiles/estimates, the FPGA function-level parallelism characteristics, and the OpenCL overhead must be considered. Therefore, in addition to the FPGA synthesis results, all devices are profiled in terms of their bandwidth and all CPU/GPU kernels are profiled in terms of their execution time. From this schedule, a program is created at design time that takes care of finding the shortest transfer paths, maintaining data coherence, and setting up synchronization mechanisms, even between different vendors. Due to these precomputations and the parallelization of the different device queues in the runtime system, it can execute the applications with the lowest possible overhead. For an application similar to the ORB algorithm, the toolchain was able to achieve speedups of up to 13.39 using multiple optimizations on the FPGA in comparison to a GPU. In a heterogeneous schedule with constrained FPGA resources, it achieved a speedup of 1.63 when using both devices.

The Embedded System Vision toolchain allows easy programming of object detection algorithms and the creation of a runtime-adaptive NoC-based architecture or a pure AC design based on an application. Its well-defined OpenVX interface, graph verification, default parameter setting, parameter propagation, and automatic creation of a complete hardware design significantly improve the design process. Its modules use OpenMP with dynamic scheduling to generate all HLS IP-cores in parallel to further improve the design process. The generated architecture with all its components (CUs, NIs, MA, DMA units, DMA controller, routers, converters, and buffers) is automatically created for the FPGA in the backend. To design a flexible architecture, the following generic HLS-based components have been developed: (1) Configurable NIs, which connect CUs to the NoC, receive, send, or create packages, and integrate mechanisms to prevent deadlocks that occur when buffers overflow. (2) DMA controller that allow shared DMA units to be configured directly over the NoC. (3) A MA, which orchestrates the application and configures the mentioned NIs and DMA controller for a specific application flow. The `APARMAP` algorithm, which will be presented in the next chapter, distributes the application graph onto a partition-based mesh-like topology, thus creating the heterogeneous NoC-based architecture. It is optimized for streaming data, efficient in terms of resource consumption, flexible, and capable of running different vision applications. The evaluation showed that the generated architecture has a high potential to reduce the resource utilization. For an example application and a resolution of 1920 × 1080, the overhead of the NoC-based architecture in terms of performance is just 0.006 % in simulation. In the real system, the overhead increases to 0.015 %. The small increase is due to the latency from the ARM to the MA, and the latency from the DMA to the main memory using the DMA controller. The example design also runs at a high frequency of 300 MHz on the ZCU104 without performance loss. With the help of the generated MA and the DMA controller, the user can easily run whole applications.

5 `APARMAP`: Application Distribution Algorithm

This chapter describes the proposed application distribution algorithm [29, 30]. Its task is to distribute application graphs on a partition-based and mesh-like FPGA topology. It is also part of the Embedded System Vision toolchain, which has been described in Section 4.6. Figure 4.14 shows the usage of the algorithm within this toolchain including the interfaces.

Figure 5.1 shows all steps of the application distribution algorithm. The different steps have been described in Section 2.3.6 together with the related work. The partitioning of an application into a task graph and the tuning of its parameters is done in the OpenVX Graph Creation module of the Embedded System Vision toolchain by utilizing the `HiFlipVX` library. The proposed algorithm uses multiple heuristics and load balancing techniques in a multi-threaded and grid-based approach to achieve scalability for both the application and the architecture. It balances between exploitation and exploration to find a near-optimal solution in a reasonable and scalable amount of time and not get stuck in local minima.

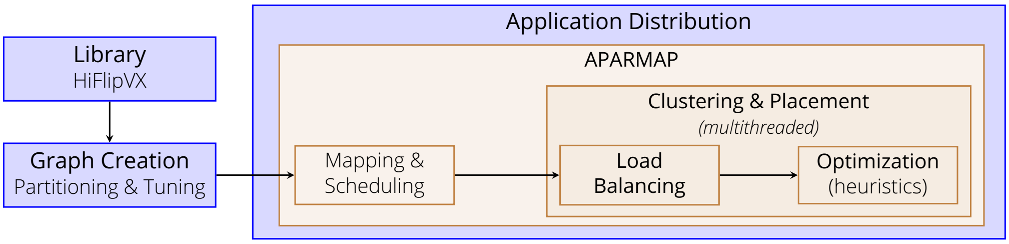


Figure 5.1: Overview of the `APARMAP` algorithm within the `DECISION` framework.

This thesis keeps the developed models and methods as general as possible, to be applicable to a wide range of use cases, applications, architectures, and topologies. This makes the algorithm independent of the toolchain and vendor. The used models and allowed topologies have been described in more detail in Section 4.6.2. Both analytical approaches as well as synthesis or simulation results can be used to calculate the data of these models. This work uses a composite of synthesis results complemented by analytical models based on the implemented library (`HiFlipVX`) as one possible instantiation. Figure 4.15 describes the algorithm using these models, as an application distribution of a task graph (*A*) into a partitioned and meshed topology (*B*) to create an application-specific hardware architecture at design time (*D*).

5.1 Overview

The application model is a data flow graph consisting of tasks and transactions (*A*). First, these tasks are mapped and scheduled to a flexible set of physical nodes (2). This schedule contains timing behavior including the information needed for DPR. The physical nodes span the node graph consisting of CUs, which can be dedicated ACs, general purpose PUs or I/Os (*C*). The platform model allows heterogeneous architectures with an irregular structure, for a high flexibility (*B*), as shown in Figure 4.16c. A NoC can be used for inter- and intra-chip communication. Its routers are connected via NIs to PRRs, which can be an entire FPGA or a region on an FPGA. This shows the applicability of the proposed algorithm for a single-chip system as well as for a large FPGA cluster.

The focus of the algorithm is the clustering and placement of the node graph (*C*) into the platform model (*D*) in a multithreaded approach. It is an NP-hard problem [11] if one of the two graphs is an irregular graph. Therefore, this thesis makes use of load balancing and different heuristics like TS and SA, in a multithreaded grid-based approach to find a near-optimal solution. To not lose optimality, clustering and placement is performed in one process. The first phase maps the node graph into the platform graph within a two-dimensional Euclidean vector space using load balancing techniques. Load balancing is a good technique to find a reasonable solution within a predictable and scalable amount of time.

The second phase optimizes the solution using various heuristics. The algorithm balances between exploitation and exploration, while iterating through the search space. This work uses a gradient descent method as search function to find possible solutions. A multi-objective function evaluates the quality of the solutions by calculating a fitness value. The main constraints and objectives are the FPGA resource utilization, the NoC bandwidth consumption, the NoC hop count and the computational speed of the toolchain. The solution space is divided into a grid to prevent threads from calculating the same solution.

SA allows solutions to get worse up to a certain value, which decreases from time to time. TS uses a smart and multithreading capable history to prevent solutions from being repeated. Additionally, it is also possible to preconfigure parts of the architecture by:

- binding tasks to specific nodes
- binding nodes to partitions
- excluding all nodes from partitions other than those bound to them

Section 5.2 describes the proposed concepts and heuristics and Section 5.3 explains the complete algorithm. The variables and constants used in the equations of this chapter are summarized in Table 5.1 and Table 5.2. The assigned values of the constants will be discussed in more detail in the evaluation in Section 5.4. Section 5.5 gives a summary of this chapter.

Table 5.1: Variables of the application distribution algorithm.

P	partition amount	$Upart$	max partition utilization
N	node amount	Rad	radius of resource
$Pgrid$	partition groups in grid	B_i	fitness value of single link
$Ngrid$	node groups in grid	Bt	utilized bandwidth of time stamp
$Pres$	partition resource	Bs	bandwidth usage of edge
$Nres$	node resource	Bs	available bandwidth of link
A, B	coordinates	$Fedge$	edge force
$norm$	normalization factor	$Frange$	range of force
$act(x, c)$	activation function	$Fstrength$	strength of force
O	fitness value	$Fnode$	node force
U	utilization	$Fpart$	partition force

Table 5.2: The different parameters and their default values of the proposed algorithm. constraints (upper); load balancing phase (middle); optimization phase (lower).

$C_{00} = 0.8$	max allowed resource utilization per partition
$C_{01} = 0.8$	max allowed bandwidth utilization per link
$C_{02} = 0$	max allowed number of hops per transaction
$C_{03} = 8$	number of parallel working threads
$C_{04} = 64$	number of load balancing iterations
$C_{05} = 16$	load balancing rotation quantization (step 1)
$C_{06} = 0.35$	load balancing edge Force *Fedge* (step 3)
$C_{07} = 0.65$	load balancing edge Force *Fedge* (step 3)
$C_{08} = 0.8$	load balancing node force *Fnode* (step 4)
$C_{09} = 0.2$	load balancing partition force *Fpart* (step 5)
$C_{10} = 1.0$	load balancing partition force *Fpart* (step 5)
$C_{11} = 16$	number of best solutions stored during load balancing (step 6)
$C_{12} = 1.2$	initial simulated annealing factor after load balancing
$C_{13} = [4..16]$	max number of STM elements per LTM element
$C_{14} = [32..255]$	max number of MTM elements per LTM element
$C_{15} = [4..16]$	max number of MTM elements allowed to deteriorate

5.2 Heuristics and Concepts

The proposed algorithm contains three phases to find a near-optimal solution: Scheduling & Mapping, Load Balancing and Optimization. This section describes the proposed heuristics and concepts that will be used. To distinguish the different solutions in terms of quality, they are rated by a multi-objective function described in Section 5.2.1. The main objectives are: resource usage, bandwidth usage and hop count. Various strategies address the dilemma between exploration and exploitation. A gradient descent method searches for new solutions in the neighborhood of a solution using the objective function, as described in Section 5.2.2. The solution space is divided into a grid, as described in Section 5.2.3. This allows several threads to work in parallel in the same solution space without collision. The SA approach described in Section 5.2.4 allows solutions to get worse up to a certain value, which decreases from time to time. Section 5.2.6 describes the TS algorithm, which uses a history (Section 5.2.5), to prevent solutions from being repeated. Additionally, solutions are only allowed to get worse for a certain number of steps. Then a different path needs to be chosen in solution space.

5.2.1 Objective Function

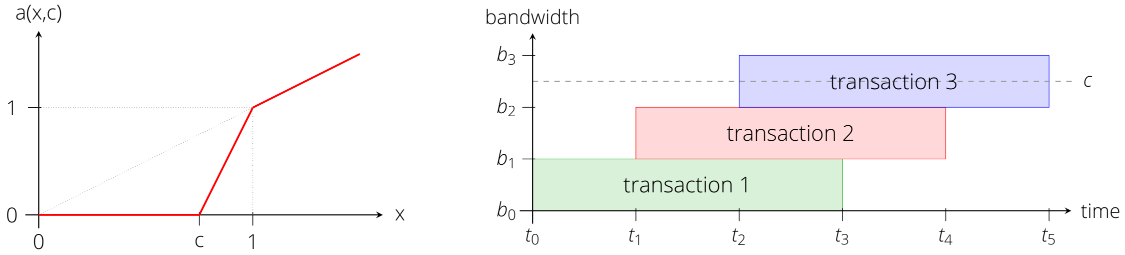
To evaluate a solution, the fitness value (O) is calculated, as shown in Equation (5.1). This thesis considers three main objectives, which are considered with descending importance. First, the fitness value (O_{part}) is calculated, which includes the maximum and average resource usage of all partitions. If the constraints of O_{part} are not met, the resource usage is too high, or the place and routing might fail. If they are met, O_{part} is zero and the fitness value O_{band}, which depends on the maximum and average bandwidth usage of all links, is calculated. If the constraints of O_{band} are not met, this would affect the execution time of the distributed application. If they are met, O_{band} is zero and the fitness value O_{hops}, which depends on the maximum and average number of hops of all transactions, is taken. O_{hops} mainly have an impact on the latency. However, for data flow applications with large packages, e.g., in image processing, an additional latency of, e.g., three, can be negligible. Additionally, a reduced average hop count decreases the dynamic energy consumption of the communication [123].

$$O = \begin{cases} O_{part} + 1 & O_{part} > 0 \\ O_{band} & O_{part} = 0 \wedge O_{band} > 0 \\ O_{hops} - 1 & otherwise \end{cases} \tag{5.1}$$

As shown in Equation (5.2), this work uses the maximum and average fitness values to calculate the fitness values for the partition (O_{part}), bandwidth (O_{band}), or hop count (O_{hops}). These values are normalized by ($norm_i$) to lie within the value range between 1 and 0. To calculate a single fitness value, an activation function is used ($act(x, c)$), as shown in Equation (5.3) and in Figure 5.2a. This function gets a lower (c) and an upper (1) limit and the input value (x). If the value is below the limit, all the requirements of the objective are met. If the value lies between the limits, the result may be of reduced quality. For O_{part} either the place and routing step of the CAD tool can fail, or a high clock frequency cannot be reached. For O_{band} the overhead for the header of a package and the setting of routes in the NoC must be considered. In addition, the package size and number of packages from different sources can have an impact on the limits (c).

$$O_i = \left(\max_{1 \leq j \leq J} act(x_{i,j}, C_i) + \frac{1}{J} \cdot \sum_{j=1}^{J} act(x_{i,j}, C_i) \right) \cdot norm_i \tag{5.2}$$

$$act(x,c) = \begin{cases} 0 & x < c \\ \frac{x-c}{1-c} & c \leq x < 1 \\ x & 1 \leq x \end{cases} \tag{5.3}$$



(a) activation function

(b) bandwidth consumption

Figure 5.2: Activation function getting a fitness value (x), a lower (c) and upper (1) boundary (left). Consumed bandwidth of a link with three transactions mapped to it and a lower boundary (c) (right).

The fitness value for O_{part} is calculated in two loops. In the first loop, the node placement of a solution calculates the resource usage of each resource type of each partition. For all subsequent solutions, only the resource usage of the migrated nodes needs to be updated. In the second loop, the maximum resource usage (U) of all partition types is calculated for each partition. For all subsequent solutions, only the maximum resource usage of partitions with a different node placement is updated. The activation function ($act(x,c)$) is applied on U and uses C_{00} as the lower limit for the desired maximum resource usage of all partitions. The normalization value ($norm_i$) is $\frac{1}{2}$.

The fitness value for O_{band} is calculated from a precomputed list of events, which contains the time stamps for the start and end of all transactions. The list is created while computing the schedule using the synthesis results. The list, which is sorted by time stamps, is traversed in ascending order. Each event belongs to a transaction that has a path from its sender to its receiver node. The fitness value of each link to which the path of the transaction is mapped will be updated. First, the utilized bandwidth (Bt_i) of the current time stamp (i) is updated by determining the available link bandwidth ($Bs(l)$) and the requested transaction bandwidth ($Bs(e)$), as shown in Equation (5.4). Then the fitness value (B_i) of the link is updated by calculating the time product, as shown in Equation (5.5).

$$Bt_i(e,l) = \begin{cases} Bt_{i-1} + \frac{Bs(e)}{Bs(l)} & StartOfTransaction \\ Bt_{i-1} - \frac{Bs(e)}{Bs(l)} & EndOfTransaction \end{cases} \tag{5.4}$$

$$B_i = B_{i-1} + (t_i - t_{i-1}) \cdot act(Bt_{i-1}, C_{01}) \tag{5.5}$$

An example of three transactions mapped to one link including the time stamps t_0 to t_5 and the bandwidth of the link, is shown in Figure 5.2b on the right. C_{01} is the lower limit for the desired maximum bandwidth usage of all links. After traversing over the list of events, there is a fitness value for each link. The final fitness value (O_{band}) is calculated using the maximum and average fitness values of all links (Equation (5.2)). In parallel to O_{band}, the hops fitness value (O_{hops}) is calculated using the same equation. Therefore, the number of hops of each event is counted and the average and maximum are calculated. O_{band} and O_{hops} are both normalized ($norm_i$) to ensure a value below 1.

In this work, an XY-routing algorithm was implemented, which had to be adapted for irregular mesh-like topologies. Thereby, routing is still done in X direction first. If this is not possible, it is routed by 1 position in Y-direction, then again in X-direction. This is repeated until the X-coordinate is correct. Then it is routed to the Y-direction. This works since irregular mesh-like topologies are not allowed to have gaps between two partitions within a row or column.

5.2.2 Local Search Function

The search function calculates new possible solutions during the optimization phase. Starting from the fitness value of the current solution, a gradient descent method calculates new possible solutions. Depending on the fitness value (Equation (5.1)) of the current solution, one of three search functions is selected. They try to improve the maximum and average: partition resource usage (O_{part}), bandwidth usage (O_{band}) or hop count (O_{hops}). During the optimization phase it is also possible to choose between two modes.

In the first mode all observed nodes (i) can be moved by exactly one hop to a neighboring partition. This creates multiple solutions. However, each of these solutions can have only one node, which has been migrated. To optimize O_{part}, all nodes placed to the partition with the highest resource usage are observed. To optimize O_{band}, all nodes connected to an edge (transaction) that is routed via the link with the highest bandwidth usage are observed. If this link connects a router with a partition, nodes communicating over this link are moved away from the partition or towards it if the number of hops is one. If this link connects two routers, there are two possible movements for the connected nodes since this work uses an XY-routing algorithm. If it communicates via a horizontal axis, the sender node is moved up or down by one. If it communicates via a vertical axis, the receiver node is moved one to the left or right. To optimize O_{hops}, all nodes connected to an edge whose hop count is equal to the maximum hop count of all edges are observed.

The second mode increases the radius of the gradient descent method if needed. In this mode all nodes j connected to i are also moved. For each combination of i and j, a solution is created that moves two nodes. The number of hops between these nodes may increase by a maximum of one for the search functions of O_{part} and O_{band}. The number of hops is not allowed to increase for the search function of O_{hops}, since it should reduce the number of hops. For both modes, the new fitness value is calculated if the migrated nodes are not bound to a partition. A solution is stored in a list if the fitness value is below the current SA. The list has a maximum of C_{13} solutions and is sorted by its fitness values. Moving two nodes instead of one increases the probability of finding a solution that meets the SA.

5.2.3 Grid-Based Solution Space

The solution space consists of all possible distributions of the nodes on the partitions. For P partitions and N nodes there are P^N possible solutions. The flits (flow control units) are always routed over the same path through the NoC, because an XY-routing algorithm is used. Depending on the number of partitions and nodes, the solution space is divided into a grid.

The partitions are grouped into *Pgrid* elements by numbering them line by line. Each group has its own ID (*PgridID*). The first (*P*/*Pgrid*) partitions belong to element one, the next (*P*/*Pgrid*) partitions to two and so on. This approach simplifies clustering partitions for irregular graphs. The grid ID (*GridID*) is calculated for a subset of nodes (*Ngrid*) by determining on which cluster of partitions each node is placed to. This leads to a grid with $Pgrid^{Ngrid}$ elements, as shown in Equation (5.6).

$$GridID = \sum_{n=1}^{Ngrid} PgridID_n \cdot (Pgrid)^{n-1} \tag{5.6}$$

A subset of nodes (*Ngrid*) could also be selected by choosing nodes with a large distance to each other, e.g., by using the Gray code. The solution space of each grid element can only be held by one thread. If a thread finds a solution for an element that is not owned by another element, it gets that grid element. If it finds a solution for an element that is already owned, it passes that solution to the owner. This approach allows exploration, prevents duplicate solutions from different threads and is used in the optimization phase.

5.2.4 Simulated Annealing

The SA approach allows to deteriorate new solutions until the fitness value has reached a certain threshold. This threshold decreases when the TS memory is full, which limits the amount of data stored by the TS algorithm. This approach is used to increase the exploration of the search space and reduce the overall computation time. The initial threshold is C_{12} times the fitness value of the initial node placement stored after the load balancing phase. Each thread has its own SA value. A fitness value is stored for each solution of the optimization phase. After C_{14} solutions the median value of all solutions is taken as new value for the threshold. All solutions with a higher fitness value are deleted from the TS and SA history as they can no longer occur. A global SA value that is the maximum of all SA values is used in the local search function, since it is also possible to find solutions for other threads.

5.2.5 History

The TS algorithm needs a history to prevent solutions from being recalculated or from ending in circles in the solution space. To manage multiple threads working on the same history, this thesis proposes a smart memory, as shown in Figure 5.3. Each thread has an LTM element that stores the history of its solutions. It stores the complete initial and last (current) solution (distribution) of the nodes. The MTM stores the intermediate solutions. The elements contain the changes between the different solutions. The STM stores the C_{13} best solutions that have been calculated by the search function but have not been taken so far. The elements are sorted by their fitness value and contain the change to the corresponding MTM element or to the initial distribution.

Additionally, the best achieved fitness value of all MTM elements is stored. The deterioration counter counts the last consecutive solutions that did not lead to an improvement. If this value exceeds C_{15}, the best element from the STM is taken as the next solution to balance exploitation and exploration. The migration counter counts the number of moved nodes

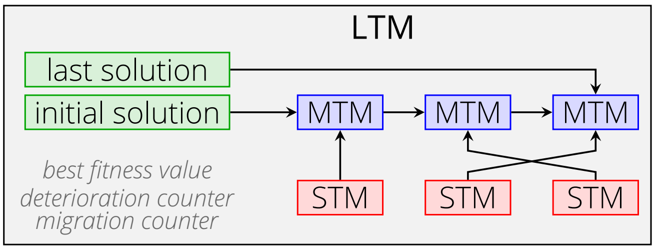


Figure 5.3: Structure of the LTM (Long-Term Memory) in the history. Each thread has one LTM (Long-Term Memory) for the TS algorithm. The MTM (Medium-Term Memory) elements are the taken solutions. The STM (Short-Term Memory) elements are the N best calculated but not taken solutions.

between the initial and final node distribution. It is needed to determine whether a solution of an MTM/STM element must be created from the initial or the last distribution. There is a `mutex` for the STM because other threads can insert STM elements into this memory. The MTM and STM elements store the fitness value of the solution, which nodes moved between which partitions, the corresponding grid ID and their migration counter. If the SA changes, all MTM/STM elements below the new threshold are deleted.

5.2.6 Tabu Search

Each thread executes the TS algorithm independently. It stores the solutions in its history to prevent solutions from being taken more than once (Section 5.2.5). It works on a grid-based solution space to prevent other threads from working on the same solution space (Section 5.2.3). It uses SA to limit its solution space and memory consumption (Section 5.2.4). It takes the possible solutions calculated by the local search function as input (Section 5.2.2).

Main Algorithm

The candidate solutions have been presorted according to their fitness value. They are traversed by the TS algorithm in ascending order, to find the next solution (MTM) and buffer further valuable solutions (STM). Listing 5.1 gives an overview of the algorithm. First the grid element and the owning thread is calculated (line 1). If no thread owns this grid element, it is obtained by the observing thread. The next step is to check whether the solution satisfies the SA of the owning thread (line 2). If the grid element belongs to another thread, an attempt is made to store the solution in the STM of the other thread (line 3-4). If it belongs to the own thread, the deterioration counter did not reach the threshold and no new MTM element has been added, the history of the MTM is checked (line 5-6). If this solution does not yet exist in history, a new MTM element is added (line 7).

Otherwise, if the deterioration counter is reached or a new MTM element has already been added, it will try to add the solution to its own STM if this is not already full (line 8-9). If no new STM element can be added, the STM will be marked as full for the following solutions in the candidate list, since all of them will have a higher fitness value. The grid element can be released if the observed solution could not be added to an STM or MTM (line 10). After all solutions have been traversed, it is checked whether an MTM element has been added

```
1  Obtain grid element if not owned by other
2  if (SA of thread holding grid element is met)
3    if (Grid element belongs other thread)
4      Try to add element to remote STM
5    else if (No MTM element has been added) & (deterioration count not reached)
6      if (Check MTM history (hamming distance))
7        Add element to local MTM
8    else if (STM not full)
9      Try to add element to local STM
10 Release grid element if no own STM/MTM added
11 if (No MTM element has been added)
12   Try to add MTM from STM elements
13 if Maximum number of MTM reached
14   Update SA and cleanup LTM
```

Listing 5.1: Main structure of the TS (Tabu Search) algorithm.

(line 11). If no MTM element has been added so far, it must be created from the STM (line 12). When a new MTM element has been added, the LTM and SA are updated. The last step checks whether the maximum number of MTM elements has been reached (C_{14}) (line 12). If so, the SA is updated and the LTM is cleaned up accordingly (line 14).

Add MTM Element (line 7): When adding a new MTM element, it is added at the end of the MTM containing its data, and the last (current) solution of the LTM is updated. All threads get their memory elements for the STMs and MTMs from a shared pool to reduce the memory usage.

Add STM Element (line 4 & 9): An STM is always locked by a mutex during access. The STM has a maximum number of elements (C_{13}) and is sorted by its fitness values. Based on the fitness value of a solution, an attempt is made to insert it as an STM element into the STM at the correct position. If a new STM element is added to an STM that already contains its maximum elements, the element with the highest fitness value is exchanged for the new element. The grid element of a deleted STM is released if it is free. If the STM is not full, the new STM element is taken from the pool. When adding an STM element to the STM of the own thread, the element is created by the observed solution. When adding an STM element to the STM of another thread, the element is created from the difference of the initial solution of the other thread and of the observed solution.

Add MTM Element from STM (line 12): If no new MTM element has been found in the candidate solution list, it is tried to create an MTM element using the STM. Therefore, it is traversed through the STM in ascending order. First, the solution of an STM element needs to be created. It starts from the nearest solution, which is the initial or last solution from the LTM and then create the STM by updating it using the MTM elements. After the solution has been created, the history is checked if it already exists. If it exists, the STM element is transferred to the end of the MTM, and its information is update. Otherwise, the STM element is given back to the pool and the grid element is released if it is free.

Check History (line 6): Whether the current solution already exists in history is checked by traversing through all MTM elements of the LTM and comparing them with the current solution. The grid-based solution space has the advantage that a thread only needs to check the MTM elements of its own LTM. For comparison, this work uses a Hamming distance bit vector. Each bit indicates whether the distribution of a node is the same (0) or different (1) between the current solution and the MTM element. The total Hamming distance is calculated using Equation (5.7). It checks whether the old solution ($M\_old_i(n)$) of a migrated node (n) in the observed MTM differs from the current one ($M\_new(n)$). This value is subtracted by the current Hamming distance ($M\_ham_i(n)$) of the node to update the total Hamming distance (H_{i+1}). All migrated nodes of an MTM element are traversed before checking if the current solution is a duplicated one, which is the case if the Hamming distance is zero.

$$\begin{aligned} H_{i+1} &= H_i + (M\_old_i(n) \neq M\_new(n)) - M\_ham_i(n) \\ M\_ham_{i+1}(n) &= M\_old_i(n) \neq M\_new(n) \end{aligned} \tag{5.7}$$

Cleanup History (line 14): The history must be cleaned up when the maximum number of MTM elements per LTM is reached (C_{14}). First, the SA of an LTM is updated using the median of the fitness values of all MTM elements as described in Section 5.2.4. Then all STM elements with a fitness value higher than the SA are deleted and free grid elements are released. All LTM data, except the MTM and STM elements, is reset. The LTM is updated while traversing through the MTM elements starting from the initial solution. If an MTM element does not meet the SA, all STM elements that point to it shall point to the predecessor MTM element. Additionally, these STM elements and the successor MTM element are updated, the observed MTM element is deleted, and its grid element is released if it is free.

5.3 Algorithm

The proposed algorithm optimizes the application distribution for three different objectives. It attempts to reduce the maximum and average: resource usage of all partitions, bandwidth usage of all links, and number of hops of all transactions. The algorithm contains three phases that are executed one after the other.

1. The **Scheduling & Mapping Phase** takes the input application graph and maps its tasks to nodes. It also creates the schedule that is needed to determine the bandwidth usage of the links for a given time period.
2. In the **Load Balancing Phase**, this node graph is distributed to the host graph within the two-dimensional Euclidean vector space using load balancing techniques. The phase computes several initial node placements in parallel.
3. In the **Optimization phase**, the final clustering and placement is created by optimizing the initial placement using the heuristics described in Section 5.2 in a multithreaded algorithm.

5.3.1 Scheduling & Mapping Phase

Before nodes can be placed to partitions, the schedule must be created. One job is to map tasks to nodes. Tasks can be bound to specific nodes in the configuration file. In addition, the schedule for the timing behavior of transactions is required. The timing of nodes is not important for their assignment to partitions since a node always exists and only the tasks on this node will change. However, mapping transactions to links requires timing behavior to see which bandwidth is needed for which time period.

The schedule is calculated in two steps. The first step is to calculate the ALAP (As-Late-As-Possible) schedule where nodes are executed as late as possible. Then all source nodes (CUs) that are either I/O or PU are fixed to this calculated time. A source node is a node that only transmits data and does not receive any. After that, the ASAP Schedule is calculated in which the precalculated time of the fixed nodes is maintained. The reason for this type of scheduling is that the latencies between tasks are reduced and thus smaller buffers (FIFOs) can be used. This is possible because nodes like PUs can have wait states, whereas ACs can only react to their input.

The ASAP Schedule is calculated by processing a list of tasks ordered in the sequential order of their execution. When this list of tasks (send) is processed, the start time (strt) of each task (recv) which receives data from the sender is updated as shown in Equation (5.8). The latency time between the start of a task and the start of a transaction is represented by the offsets (off). If two tasks run on the same node, they are executed one after the other. An additional delay can be added between these tasks if DPR needs to be performed.

$$strt_{recv} = max(strt_{recv}, strt_{send} + off_{send} - off_{recv}) \tag{5.8}$$

When DPR is used, an entire partition, including all its nodes, will be reconfigured. This happens because this work uses a common interface to the router, which makes the DPR easier. Therefore, only reconfigurable nodes can be assigned to a partition that should be reconfigured. In addition, the scheduler must take the reconfiguration time into account. Furthermore, if the maximum possible buffer size is used, the scheduler can recognize whether deadlocks may occur due to too small buffers. This is done by calculating the delay time of a transmission between a sender task and a receiver task. From this information the number of required buffer elements can be calculated.

5.3.2 Load Balancing Phase

The second phase of the algorithm does the load balancing to create an initial clustering and placement. It is a deterministic process that will output the same result for the same input. It takes the node (guest) graph, which consists of nodes and edges (transactions), as input. In the initial phase, the guest graph is mapped into a two-dimensional Euclidean vector space. This is done by calculating the distances between one node and all other nodes using the Dijkstra algorithm. The distance between two nodes is the minimum number of edges between them. If another node is selected as the starting node, a different initial graph is created. This way several initial graphs are created, as described in Section 5.3.2.1.

Each thread applies load balancing on a different initial graph independently from each other and in parallel to each other. During this process, the guest graph is expanded within the boundaries of the host graph in the Euclidean vector space. Figure 5.4 shows a guest graph that was mapped to a host graph using load balancing. The main goal of the load balancing process is to reduce the maximum resource usage of each partition in the host graph or to get it below a certain value. In addition, the maximum length of the edges in the guest graph is kept small to reduce the maximum number of hops required for the communication between different nodes. The load balancing algorithm iteratively optimizes the solution in

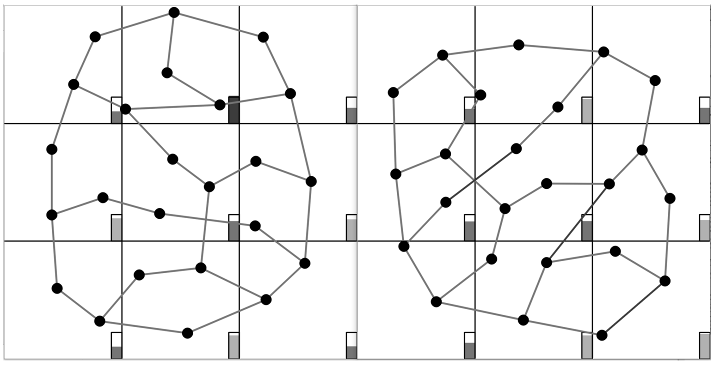


Figure 5.4: Plot of the node graph after load balancing it into a 3 × 3 host graph for two independent threads. Bars show maximum resource usage of partitions.

six steps, which will be described in more detail later. The different steps are only active for a certain period of iterations, as shown in Figure 5.5.

- **Step 1**: The guest graph is inserted into the host graph by scaling and rotating.
- **Step 2**: Nodes that are only connected to two others are centered between them.
- **Step 3**: Nodes that are connected to each other are forced towards each other.
- **Step 4**: Nearby nodes are forced apart from each other according to their resource usage.
- **Step 5**: Nodes are forced to adjacent partitions depending on their resource usage.
- **Step 6**: The current solution is stored depending on the fitness values.

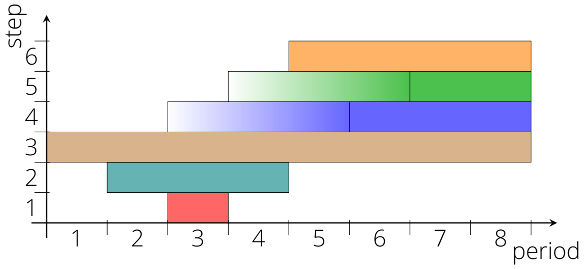


Figure 5.5: Shows which steps of the load balancing process are executed in which iteration periods. Steps 4 and 5 increase their strength until they are applied at full strength.

5.3.2.1 Initial Guest Graphs

Initial guest graphs need to be created in a two dimensional Euclidean vector space using the node graph, before load balancing can take place. To determine the coordinates of the different nodes in the guest graph, the distances between these nodes are calculated using the Dijkstra algorithm. The distance between two nodes is the minimum number of edges that lie between them. To obtain different starting points in the solution space, each thread (*t*) needs its own guest graph.

To achieve a distinction between the different guest graphs, different nodes are chosen as **1. reference point** (*A*). This reference point is placed in the center of the coordinate system, as shown in Figure 5.6. To determine coordinates for the other nodes that are as accurate as possible, three reference points are required.

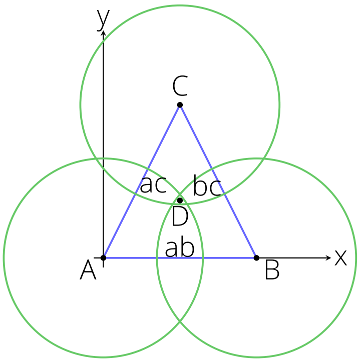


Figure 5.6: Computation for the initial node coordinates (A, B, C and D) of a guest graph. The distances ab, ac, bc are the minimum amount of edges between these nodes.

The **2. reference point** (*B*) is the node with the furthest distance to the first one. As shown in Figure 5.6 this point is located on the x-axis of the coordinate system. To avoid duplications, such as *ab*, *ba*, twice as many reference pairs as threads ($t \cdot 2$) are calculated. Duplicated pairs are deleted and the best pairs in terms of maximum distance are selected for the different threads. Each thread computes its guest graph based on its reference pair (*A*, *B*).

The **3. reference point** (*C*) is selected based on its distances to *A* and *B*. The node with the highest result for ($x_i = ac_i \cdot bc_i$) is chosen as the reference point. The coordinate of the third reference point is calculated using the following equations:

$$\begin{aligned} C_x &= \cos(\pm\alpha) \cdot ac \\ C_y &= \sin(\pm\alpha) \cdot ac \\ \alpha &= \cos^{-1}\left(\frac{ac^2 + ab^2 - bc^2}{2 \cdot ac + ab}\right) \end{aligned} \tag{5.9}$$

The same equation is used to calculate the coordinates of all other nodes (*D*). As shown in Figure 5.6, it is possible that there are multiple intersections when the coordinate of *D* is calculated. The final coordinate of *D* is the center of the smallest possible triangle that can be formed from these intersections. To increase the accuracy of the guest graph, the coordinates of all nodes are computed several times, using reference points *A* and *B* with a different *C*. Therefore, the results of the equation ($x_i = ac_i \cdot bc_i$) for all possible reference points *C* are stored in a vector and then partially sorted. The best ($t \cdot 2 - 2$) results are used to determine the different nodes for *C*. The average value of the calculated coordinates is taken as the final coordinate.

Load Balancing Steps

After the initial guest graphs have been computed, they can be unfolded in the various threads in the host graph using load balancing. In **step 1**, the guest graph is rotated and scaled to fit best into the host graph. For this purpose, the guest graph is rotated in a certain number of steps (C_{05}) by a total of 180°. For each rotation, the area of the smallest rectangle that fits around the guest graph is calculated. The rotation where the rectangle covers the largest area is chosen as the best solution. If the host graph is a rectangle, the longer side of the guest graph rectangle is scaled to the longer side of the host graph rectangle. The guest graph is then rotated according to this solution and scaled to fit within the limits of the host graph. If the host graph is an irregular graph, it is extended to a minimally large rectangular graph that fits around the irregular host graph. The new area (partitions) in this extended host graph is marked as not valid. This simplified process is done to reduce the computation time.

In **step 2**, nodes that are only connected to two other nodes are centered between them to achieve equally long edges. In **step 3**, all interconnected nodes are pulled towards each other. Both nodes are pulled with the same force ($\overrightarrow{Fedge_j}$), which depends on the distance ($dist(\overrightarrow{a}, \overrightarrow{b})$) between the node coordinates ($\overrightarrow{A_j}$ & $\overrightarrow{B_j}$), as shown in Equation (5.10). All distances and the maximum distance shown in the equation are computed first. Because nodes can have multiple forces, all forces of a node are summed before its coordinates are updated. The coefficients C_{06} and C_{07} are used for fine-tuning.

$$\overrightarrow{Fedge_j} = \frac{(\overrightarrow{A_j} - \overrightarrow{B_j})}{2} \cdot min(\frac{dist(\overrightarrow{A_j}, \overrightarrow{B_j}) \cdot C_{06}}{max_{0 \leq k < K}(dist(\overrightarrow{A_k}, \overrightarrow{B_k}))}, C_{07}) \tag{5.10}$$

In **step 4**, nearby nodes are forced apart from each other according to their resource usage. The algorithm uses a grid-based approach to detect collisions between two nodes, which reduces the complexity of computations from $O(n^2)$ to approximately $O(n \cdot log(n))$. The grid size is the maximum radius of all nodes for each resource type (e.g., LUT, FF, DSP, BRAM) on each partition type. Equation (5.11) calculates the radius for a resource type of a node on a partition type, where the resource usage of a node is *Nres*, and the available resource of a partition is *Pres*. The coefficient C_{00} defines the maximum resource usage a partition should have after the entire distribution process.

$$Rad_k(l) = \sqrt{\frac{1}{\pi} \cdot \frac{Nres(l)}{Pres(l)} \cdot \frac{1}{C_{00}}} \tag{5.11}$$

The node coordinates are stored in a sorted sparse matrix to reduce memory usage and ease the collision detection by using adaptive search pointers. For nodes that are in adjacent grid coordinates it needs to be checked if they collide. They collide if the total range of the force ($Frange_j$) is smaller than the distance ($dist(\overrightarrow{A_j}, \overrightarrow{B_j})$) between them. This range is calculated by the maximum of the ranges of all resource types of a node, as shown in Equation (5.13). The strength ($Fstrength_i$) of this force depends on the iteration (i) of the load balancing process and increases between iteration t_0 and t_1, as shown in Equation (5.12) and in Figure 5.5.

$$Fstrength_i = \begin{cases} \frac{i - t_0}{t_1 - t_0} & t_0 \leq i < t_1 \\ 1 & i \geq t_1 \end{cases} \tag{5.12}$$

$$Frange_j = max_{0 \leq k < K}(Rad_k(A_j) + Rad_k(B_j)) \cdot Fstrength_i \tag{5.13}$$

The total force acting on a node ($Fnode_j$) in a collision depends on the distances between both nodes and the range of their forces, as shown in Equation (5.14). The coefficient C_{08} is the tolerance factor of this force to prevent the effect of hysteresis between two nodes. Because nodes can have multiple forces, all forces are summed for each node before the node coordinates are updated.

$$\overrightarrow{Fnode_j} = \frac{(\overrightarrow{A_j} - \overrightarrow{B_j})}{2} \cdot \frac{(Frange_j - dist(\overrightarrow{A_j}, \overrightarrow{B_j})) \cdot C_{08}}{dist(\overrightarrow{A_j}, \overrightarrow{B_j})} \tag{5.14}$$

In **step 5**, nodes are forced to adjacent partitions depending on their resource usage. The first part of this algorithm moves all nodes inside of the extended host graph. Additionally, it precomputes the resource usage of each partition using the new node coordinates. The second part of the algorithm calculates the total force applied to the node by the adjacent partitions. If the node is bound to a certain partition, it is only pulled by the center of that partition. Otherwise, the node is pulled to the center of its current partition and to the 4 adjacent partitions, independently of each other. Of course, a partition can only be pulled by a valid partition. If there is no partition that pulls the node, it will be pulled to the center of the host graph. Because nodes can have multiple forces, all forces are summed for each node before the node coordinates are updated. Equation (5.15) shows the force by which each separate partition acts on the node ($\overrightarrow{A_j}$). In this function, $\overrightarrow{B_j}$ is the center of the partition or host graph. $Upart_j$ is the maximum resource usage of all resource types in case $\overrightarrow{B_j}$ would be mapped to the pulling partition. A single partition force ($\overrightarrow{Fpart_j}$) is only computed if ($Upart_j < 1$), to let only partitions force a node, which have available resources. The strength $Fstrength_i$ has different values for t_0 and t_1, as shown in Figure 5.5, where the lighter box marks the period between t_0 and t_1. The coefficients C_{09} and C_{10} are used for fine-tuning.

$$\overrightarrow{Fpart_j} = \frac{\overrightarrow{B_j} - \overrightarrow{A_j}}{2} \cdot min(\frac{C_{09}}{dist(\overrightarrow{A_j}, \overrightarrow{B_j}) \cdot (Upart_j)^2}, C_{10}) \cdot Fstrength_i \tag{5.15}$$

In **step 6**, the current solution is stored depending on its fitness values. The fitness value is obtained by computing the objective function on the current solution (Section 5.2.1). The distribution is obtained by converting back from the Euclidean vector space and placing the nodes to the nearest valid partitions. The best solutions of all threads are stored into a list of solutions. To achieve a better exploration the entire solution space is divided into a grid (Section 5.2.3). Therefore, if multiple solutions are in the same grid of the solution space, only the best of them is stored into the list. Storing a solution in each iteration of the load balancing process from multiple threads has several benefits. The threads will find different valuable solutions, depending on their start in the solution space. Solutions can also get worse during load balancing process. Multiple starting points can be created in solution space for a better exploration in the optimization process.

5.3.3 Optimization Phase

After all threads have completed the load balancing phase, the optimization phase begins. In this phase a multithreaded heuristic approach finds a near-optimal solution depending on the objectives. The different heuristics and concepts used are described in Section 5.2. In the first step, initial solutions are assigned to each thread. The intermediate solutions from the load balancing phase are used for this purpose. The C_{11} best solutions were stored in the previous phase, whereby only one solution per grid element was allowed. The solutions are sorted by their fitness value and distributed in ascending order to the threads as initial solution. After each thread has received a solution, the process starts again. However, all further solutions are stored in the STM of the respective thread. Furthermore, a thread owns the grid elements of the solutions assigned to it. If the load balancing process could not find a solution for each thread due to the grid fragmentation, random solutions are assigned to the threads that did not receive a solution.

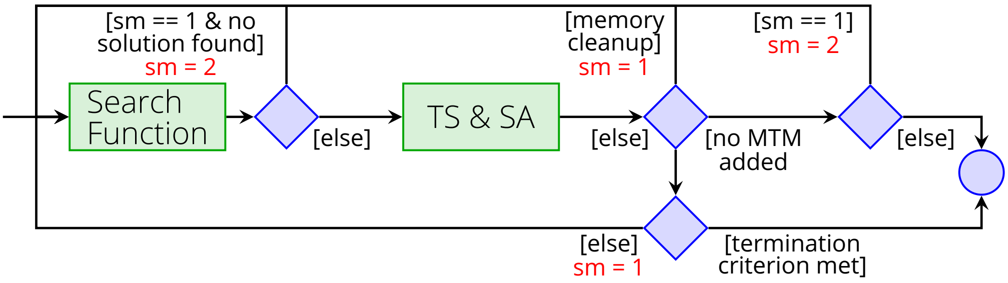


Figure 5.7: Optimization algorithm for clustering and placement process. sm = search mode

After the initial solutions of all threads are stored, the threads calculate new ones iteratively and in parallel, as shown in Figure 5.7. In the first step, the search function calculates possible new solutions, as described in Section 5.2.2. If no solution was found and search mode1 was selected, the next iteration starts with search mode2, which increases the radius of the gradient descent method. Then the TS algorithm takes the candidate solutions to find a new solution, as described in Section 5.2.6. If the limit of MTM elements is reached, the history is cleaned up and the next iteration starts with search mode1. If no new solution was found and search mode1 is selected, search mode2 is selected and the next iteration begins. If search mode2 is selected and no solution was found, the thread is terminated. If a new solution was found and all constraints are met, the thread is terminated. If a new solution was found and the constraints are not met, search mode1 is selected and the next iteration is started. The constraints depend on the three objectives: maximum resource usage (C_{00}), maximum bandwidth usage (C_{01}) and maximum number of hops (C_{02}). Due to the SA the algorithm will always terminate. After all threads have terminated, the clustering and placement of the best fitness value of all threads is created and saved in the output file. The algorithm in the optimization phase is not deterministic, because the collaborative work between the threads can lead to a different final solution.

5.4 Evaluation

This section evaluates the C++ implementation of APARMAP. It is an application distribution algorithm for partition-based and mesh-like FPGA topologies. It is part of the Embedded System Vision toolchain evaluated in the previous section, but it can cover a much wider scope, which is why this section examines it separately. It uses several heuristics and load balancing techniques in a multithreaded and grid-based approach to find a near-optimal solution.

This work has created several application graphs to evaluate the distribution of applications on different topologies for various parameter settings. The key performance metrics are the fitness values of the final solution and computation time of the algorithm. Other goals of the algorithm are its scalability, optimality, and multithreading capability. The computation time of the algorithm does not play a role in the execution time of the application since it distributes the application at design time. However, the computation time of the algorithm is not negligible and should scale with the problem size. This work uses an exhaustive search method to determine the values of the algorithm parameters under the constraint of their reasonability and by using visualization methods.

5.4.1 Default Parameters

Table 5.2 shows the default values of the algorithm parameters. The first three parameters in the table are the constraints and objectives of the algorithm. A resource utilization of less than 100 % is beneficial to meet the timing constraints of the routed design. The maximum bandwidth utilization depends on the overhead of a package sent via the NoC. This overhead can be the header information of a package. The algorithm starts reducing the maximum and average hop count when the distribution satisfies the bandwidth and resource utilization constraints. Decreasing the number of hops improves the computation time of the application and reduces the energy consumption of the communication infrastructure. Therefore, the distribution algorithm only indirectly considers the execution time of the algorithm. However, the developer can update the objectives by adding the calculation for the fitness value and the search function.

The algorithm divides the load balancing phase into six steps. Each step is active only for a certain number of iterations. C_{04} shows the total number of iterations in this phase. The first step rotates the guest graph with a quantization of C_{05} and maps its best fit to the host graph. Higher quantization would increase the accuracy, but also the computation time since this process is quite computationally intensive. However, further increasing the quantization did not result in a better fitness value. The algorithm has two parameters to adjust the calculation of the edge force (*Fedge*) and the partition force (*Fpart*) respectively. The tolerance factor C_{08} prevents a hysteresis effect between two nodes caused by the node force (*fnode*). The load balancing phase stores the C_{11} best solutions. Thereby, each grid element can only store a maximum of one solution to improve the exploration. The value of C_{11} must be higher than the number of parallel threads (C_{03}). Here the selected value is always twice the number of threads.

The initial SA value for the optimization phase is C_{12} times higher than the worst fitness value of the C_{11} initial solutions stored in the load balancing phase. Each thread has one LTM with a maximum of C_{13} STM elements and C_{14} MTM elements. Only C_{15} MTM elements in a row can have a worse fitness value than the previous one. The parameter values of C_{13} and C_{14} depend on the number of nodes in the guest graph. Their value has an upper bound to limit memory consumption and computation time, as shown in Equation (5.16). The parameter value of C_{15} follows the same calculation as the value of C_{13}. The offset was determined by measurements and is a trade-off between computation time and achieved fitness value.

$$
\begin{aligned}
C_{13} &= min(16, 4 + \lfloor log2(N) \rfloor) \\
C_{14} &= min(255, 32 + \left\lfloor log2(N)^2 \right\rfloor)
\end{aligned}
\tag{5.16}
$$

5.4.2 Measurement Methodology

This section evaluates the algorithm using eight parallel threads (C_{03}) on an AMD 3900X CPU that has 12 cores and 24 threads. The algorithm uses the 80th percentile of 256 measurements to measure the execution time and filter the deviations caused by the OS. It averages 100 runs to determine the fitness value since the multithreaded optimization phase is not deterministic. However, there are no boxplots of the fitness values because the variance of the results is too small to be visible.

The default host graph in this evaluation is regular, homogeneous, and has a 3×3 topology. Therefore, all partitions have the same number of available resources. The application graphs are heterogeneous and consist of 24, 36, 48, 60, 72, 84 or 96 nodes. They should demonstrate the differences between the different objectives. Figure 5.8 shows three of these application graphs. For a large partition with the same number of resources as the entire host graph, the application graphs would consume exactly 60 % of the resources of each resource type. The number of resources a node needs from each resource type varies. This leads to fragmentation when distributing the nodes across the different partitions. Each node has an average of 2.5 edges connected to it. A transaction utilizes 25 % of the available bandwidth of the NoC and has a duration of 3000 time units. The total execution time of the different application graphs would be between 8000 and 11 600 time units if there were no bandwidth constraints and no additional delays introduced by the NoC. Therefore, only the number of hops affects the total latency if the distribution meets the bandwidth constraints.

5.4.3 Memory Usage

Table 5.3 shows the total memory consumption of the proposed algorithm. It includes the main object of the algorithm and the memory that is dynamically allocated during the application distribution process. All memory is allocated as a large chunk and memory management takes place within the algorithm. The table shows the memory consumption for a given number of threads and nodes, since these are the variables that have the greatest impact. As the table shows, memory consumption scales linearly with the number of threads and nodes in the application graph. For each MTM or STM element in the TS algorithm, the memory consumption increases by 553 bytes. In summary, the table shows the low and scalable memory consumption of the implemented algorithm.

5.4.4 Load Balancing & Optimization Phase Comparison

Figure 5.9a shows the average fitness value for an application graph with N nodes distributed on a host graph with 3×3 partitions. There are three different configurations: only the

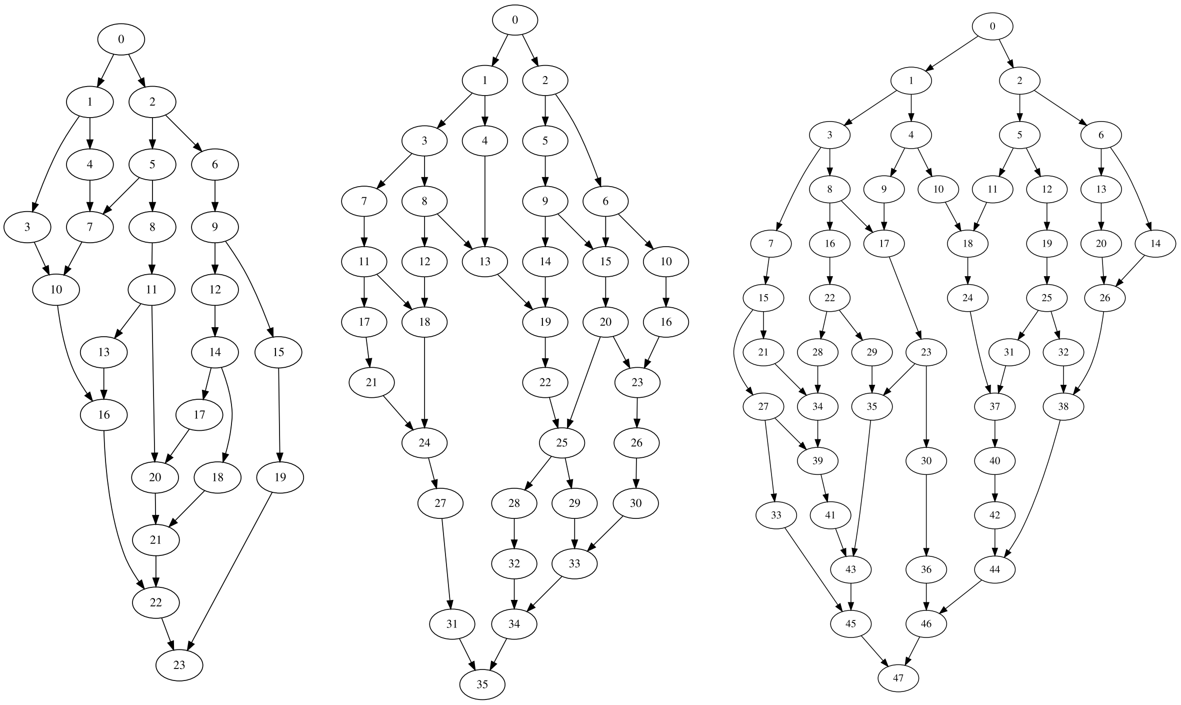


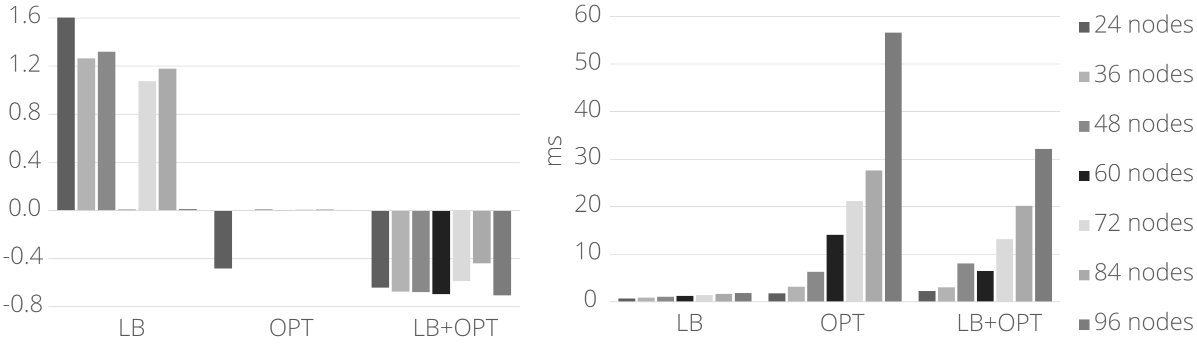
Figure 5.8: Example graphs with 24, 36 and 48 nodes.

Table 5.3: All dynamically allocated memory in kB of the proposed algorithm.

Nodes	1 thread	2 threads	4 threads	8 threads
24	18	27	47	86
48	25	38	66	121
72	31	49	85	156
96	38	59	101	186

load balancing phase (LB), only the optimization phase (OPT) or the complete algorithm (LB+OPT). All three configurations include the sequential part between LB and OPT phase, which stores the initial solutions and distributes them to the threads. This part generates random solutions when there is no load balancing. Processing the load balancing phase alone already meets the resource constraints for 2 out of 7 application graphs. Processing the optimization phase alone always satisfies the resource constraints. However, it satisfies the bandwidth constraints only for one graph. The fitness values show that finding a good initial solution using load balancing always improves the results of the multithreaded TS/SA algorithm. The reason for the outliers of some graphs is that the resource consumption of each node is different, resulting in various constellations. Graphs with more nodes are more likely to have less fragmentation, which can lead to better results.

Figure 5.9b shows the computation time for the same configuration. The load balancing phase computes faster than the optimization phase. It also reduces the computation time of the optimization phase for an increasing number of nodes. This means that load balancing also improves the scalability of the computation time with respect to the number of nodes. In addition, the computation time of the load balancing phase is predictable because it is



(a) fitness value (lower is better)

(b) computation time

Figure 5.9: Application graphs with N nodes placed on a 3 × 3 mesh for LB (load balancing phase only), OPT (optimization phase only) and LB+OPT (complete algorithm).

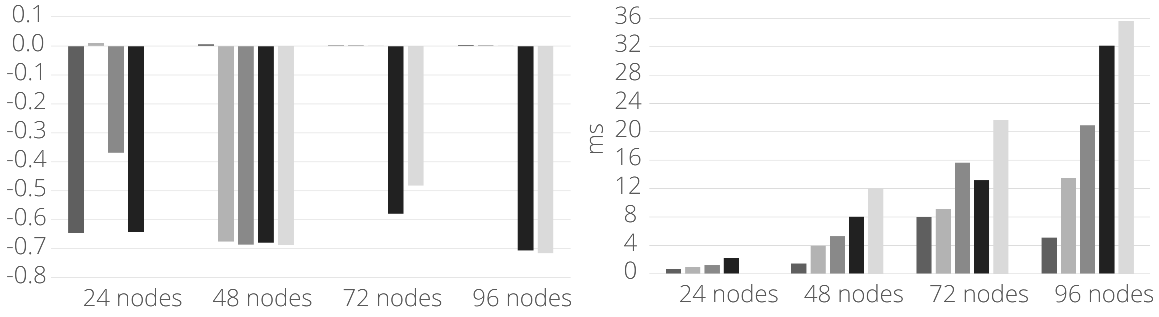
deterministic.

5.4.5 Amount of Threads

Figure 5.10a shows the fitness values of four application graphs distributed to the host graph depending on the number of parallel threads. The number of nodes must be twice the number of threads. Therefore, there is no result for 24 nodes and 16 threads. A higher number of threads does not always lead to a better result. One observation is that the more nodes there are in the graph, the more likely it is that a larger number of threads will contribute to find a better solution. Increasing the number of threads improves exploration and minimizes the possibility of getting stuck in a local minimum. Another fact that may cause the algorithm to get stuck in a local minimum may be related to the maximum number of STM and MTM elements and SA, which forces the algorithm to terminate.

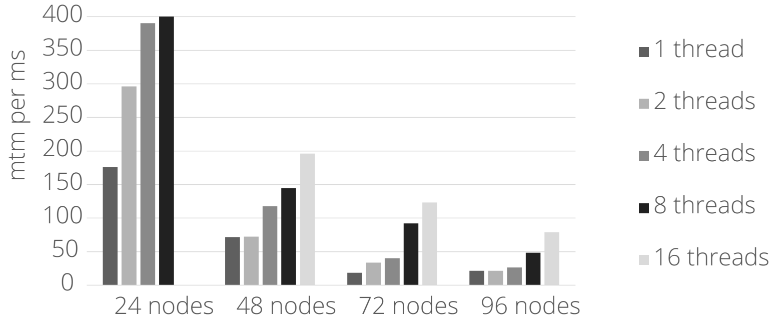
As shown in Figure 5.10b, the number of threads has a direct impact on the computation time. However, it does not increase linearly with the number of threads. The smallest increase is for the graph with 72 nodes. In this graph, 16 threads require only 2.7 times the computing time of one thread. The largest increase is in the graph with 48 nodes. In this graph, 16 threads require 8.4 times the computation time of one thread. The computation time of the load balancing phase is shorter than that of the optimization phase. The ratio between the computation times of the two phases increases with an increasing number of nodes, but not with an increasing number of threads.

Figure 5.10c evaluates whether an increase in threads causes the optimization phase to add more MTMs per time unit to the LTMs. As the number of threads increases, the optimization phase computes an increasing number of MTMs per ms. However, the ratio does not increase at the same rate as the number of threads. 16 (8) threads compute on average 4.4 (2.9) more MTMs. This is partly due to the sequential part of the algorithm, but also because different threads need to access shared elements. Furthermore, an increasing number of nodes also computes fewer MTMs per ms. This is because the algorithm has to place more nodes on the same partition on average. Therefore, the gradient descent method must compute more possible solutions per step.



(a) fitness value (lower is better)

(b) computation time

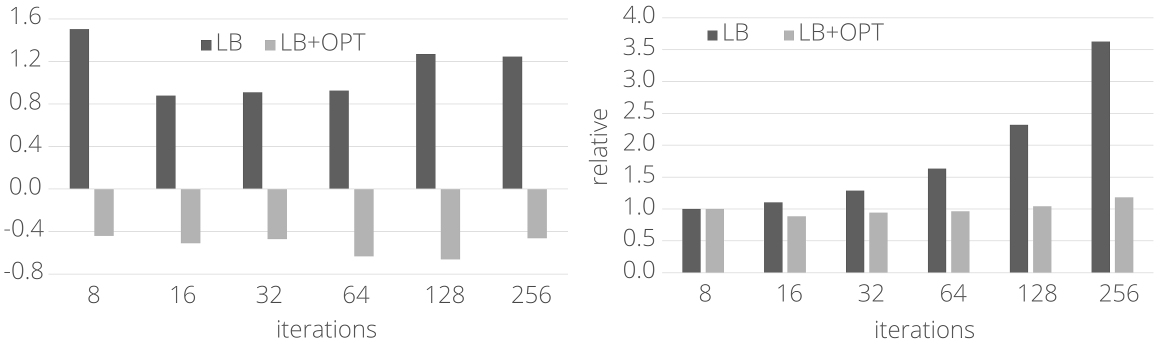


(c) MTM elements taken per ms in the optimization phase

Figure 5.10: Application graphs with N nodes placed on a 3 × 3 mesh using T threads.

5.4.6 Load Balancing Phase Iterations

Figure 5.11a shows the achieved fitness value in dependence of the number of iterations in the load balancing phase. It shows the average of the best fitness values achieved by all application graphs after the load balancing phase or the complete algorithm. The fitness value does not correlate with an increasing number of nodes in the application graph. For example, application graphs with more nodes do not require a higher number of iterations to achieve a good fitness value. The result also shows that after a certain number of iterations, further iterations do not improve the results.



(a) fitness value (lower is better)

(b) relative computation time

Figure 5.11: Average value for application graphs of size (24, 36, 48, 60, 72, 84, 96) placed on a 3 × 3 mesh after I iterations in the load balancing phase. Results shown for LB (load balancing phase only) and LB+OPT (complete algorithm).

Figure 5.11b shows the average computation time of the different application graphs in relation to the computation of eight iterations. The factor in computation time between 8 and 256 iterations is 2.25 for 24 nodes and 5.03 for 96 nodes. The smaller increase in the 24-node application graph is due to the sequential part. If the number of iterations is small, the impact on the total computation time remains small. The fact that eight iterations require more time is because the load balancing results are worse, and the optimization phase calculates more solutions. Furthermore, Figure 5.11a shows only the best fitness value after the load balancing process, but not the initial values of the other threads.

5.4.7 Load Balancing Complexity

To obtain a scalable algorithm, it is important that the computational complexity is low. The highest complexity of the load balancing algorithm is the collision detection in the fourth step of the algorithm. This step forces nodes that are close to each other away from each other according to their resource consumption. A naive method would calculate the distance from each node to every other node. This work uses a grid-based method to reduce the number of observations. The algorithm stores the grid elements in a sparse matrix to keep the memory requirements linear. Unfortunately, the sparse matrix must be sorted, which has a complexity of $\mathcal{O}(n \cdot \log_2(n))$. Figure 5.12 compares the complexity of the naive method with the grid-based method. The *collisions* curve shows the average number of detected collisions, which is the same for both methods. The complexity is linear for an increasing number of nodes. The *check if in grid* curve describes the average number of nodes in the sparse matrix for which the algorithm checks if they are neighbors in the grid. The number of observations that check if a node is within the grid depends on the number of collisions:

$$\mathit{observations} \approx \mathit{collisions} \cdot \log_2(n) \tag{5.17}$$

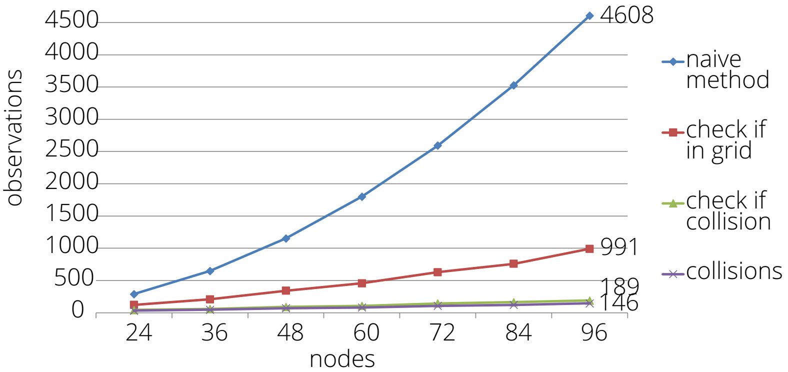
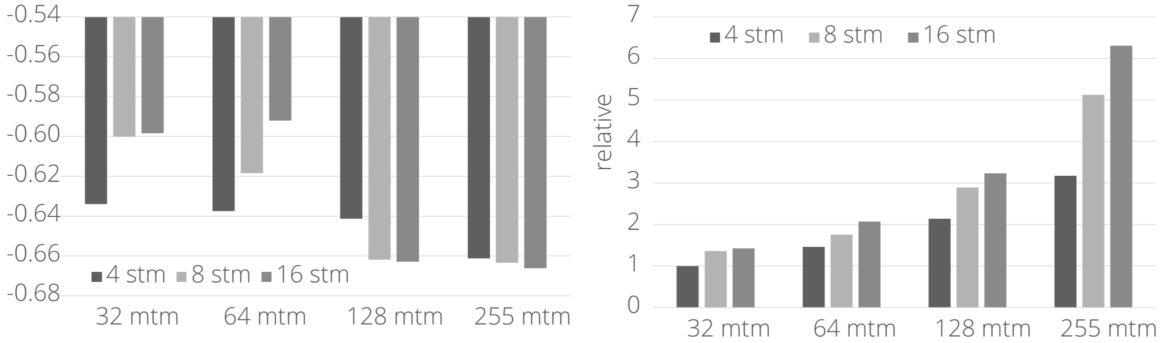


Figure 5.12: Compares the amount of observations of a naive method and a grid-based method for the collision detection in the load balancing phase.

The *check if collision* curve describes the average number of distances that the algorithm calculates to determine whether a grid collision is really a collision. This is the most computationally intensive part of the algorithm. On average, there are 30 % more grid collisions than true collisions for the example application graphs. The difference between real collisions and grid collisions depends on the different capacity utilization of the different nodes.

5.4.8 Optimization Phase: MTM and STM amount per LTM

Figure 5.13a shows the average fitness value achieved by the different application graphs depending on the size of the STM and MTM. The measurement did not consider the 96-node application graph because a size four STM did not produce results. A larger MTM size also leads to a better result. For a larger MTM size, a larger STM size further improves the fitness value. This indicates a correlation between the MTM and STM sizes.



(a) fitness value (lower is better)

(b) relative computation time

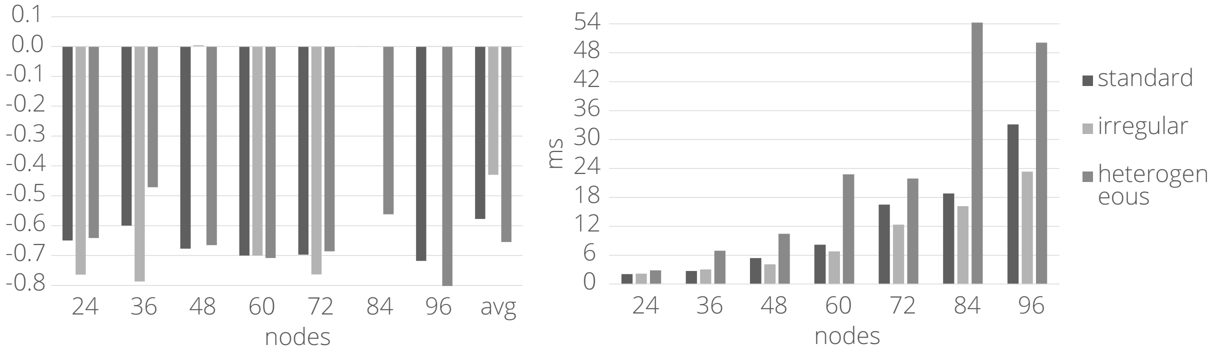
Figure 5.13: Average value for application graphs of size (24, 36, 48, 60, 72, 84, 96) placed on a 3 × 3 mesh. Evaluated for a different number of MTM and STM elements.

Figure 5.13b shows the corresponding computation time with respect to the smallest configuration of MTM and STM elements. The size of the MTM has a greater impact on computation time than the STM. As the MTM becomes larger, the STM has a greater impact on computation time. As the number of nodes in the application graph increases, the influence of the MTM size on the final fitness value also increases. Therefore, Equation (5.16), which calculates the number of STM (C_{13}) and MTM (C_{14}) elements, depends on the number of nodes.

5.4.9 Irregular and Heterogeneous Topologies

This part evaluates the influence of the topology on the fitness value and computation time. It uses the standard 3 × 3 host graph, an irregular graph (Figure 4.16b) and a heterogeneous graph (Figure 4.16a). The irregular graph results in one less connection and a more difficult graph to place the application graph. The heterogeneous graph has the same total amount of resources as the other two topologies. However, the partition in the center has 2.6 times as many resources as the standard partition and the ones in the corners only 0.6 times.

Figure 5.14a compares the fitness values. Due to the missing link and irregularity, it is more difficult to meet bandwidth requirements for the irregular topology. The heterogeneous topology achieves the best fitness value on average. This can be due to the middle partition with the most links also has most resources, and the corner partitions with fewer links have the least. Figure 5.14b compares the computation time. On average, the irregular topology requires the least amount of time. The heterogeneous topology, which is composed of three different types of partitions, takes most time. However, it is not possible to tell from the measurement results whether the increase follows a certain regularity.



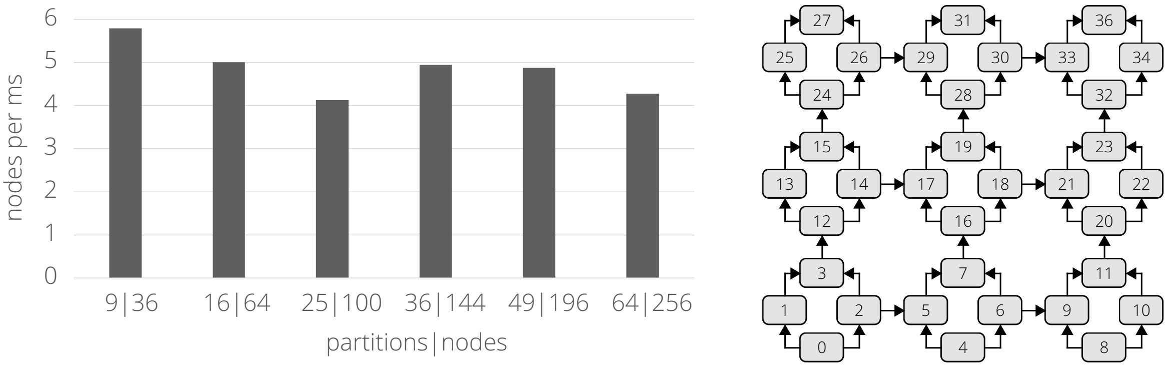
(a) fitness value (lower is better)

(b) computation time

Figure 5.14: Application graphs with N nodes placed on 9 partitions using different topologies.

5.4.10 Verifying Optimality and Scalability

This work created a set of application graphs whose optimal placement is known a priori to verify the optimality of the algorithm. The associated platform models for this experiment all have a square topology (P×P), but different sizes. An optimal placement would place four nodes on each partition. Each node connects to a node of the neighboring partitions (north, west, east, or south), if any. Figure 5.15b shows an example graph with 36 nodes. Each node has a random number of resources. However, each node has at least 15 % of each type of resource. Thus, the optimal placement consumes 75 % of the resources of each type on each partition. This is a higher resource utilization than the 60 % from the previous measurements. On the one hand, this makes it more difficult to find a near-optimal solution. On the other hand, it makes it easier to evaluate the optimality of the final solution. The only configuration parameters that changed in this measurement are the number of threads ($C_{03} = 16$) and load balancing iterations ($C_{04} = 512$). For this type of graph, a higher number of iterations led to better results.



(a) relative computation

(b) 36-node application graph

Figure 5.15: Relative computation time per node of different application graphs placed on different sized platform models.

Figure 5.15a shows the relative computation time required by the algorithm per node of an application graph to compute its final solution. Increasing C_{04} by a factor of eight did not increase the total calculation time much. The results show the good scalability of the algorithm in terms of the number of nodes and partitions. The algorithm found the intended optimal node placement for the first five application graphs. Only for the largest example with 64 partitions and 256 nodes this was not the case. This example achieved a fitness value of 0.0055, showing that the algorithm could satisfy the resource criterion but not the bandwidth criterion. Considering that the resource criterion is 80 %, this is still a very good solution.

5.4.11 Real Application

This part evaluates the proposed application distribution algorithm using the AKAZE feature detector [20] as a real application example. The implementation is based on the HLS design from Section 3.2.5, but for four octaves, which makes the graph larger. The OpenVX Graph Creation module generated the application graph, which contains all HLS-based IP-cores and DMA blocks. In total, the application graph has 271 nodes and 441 edges. It is irregular due to the nonlinear diffusion process of AKAZE.

The two source nodes and the one sink node are DMA blocks that read the input images and write the output feature vector. The input is a grayscale image (8 bit) with a resolution of 1920 × 1080. The output consists of 2048 64 bit wide features. The platform model is a 3 × 3 mesh consisting of nine PYNQ-Z1 FPGA boards (3.9). Without fragmentation, the application graph would consume 79.6 % LUTs, 30.5 % FFs, 24.9 % DSPs, and 63.5 % BRAMs of the complete host graph. The most utilized resources of a single node on a single FPGA would be: 18.6 % LUTs, 13.1 % FFs, 19.0 % DSPs, or 47.1 % BRAMs.

The maximum resource utilization C_{00} is set to 0.85 because the application graph would require almost 80 % of the available resources without fragmentation. The proposed algorithm needs an average of 149 ms (minimum 134 ms) until it terminates. The final distribution satisfies the resource constraints. Considering the complex graph and the low possibility of fragmentation, this is a very good result. Due to the streaming application, where almost all nodes stream their data through the network simultaneously, 32 bit wide connections are not sufficient to meet the bandwidth requirements. Assuming that the bandwidth can only be a multiple of two, the communication infrastructure needs a 256 bit wide interface to meet the requirements. With this approach, the developer can easily adapt the required communication infrastructure to the application. With the 256 bit wide link bandwidth, the distributed application has an average hop count of 0.6149. When increasing the resource constraints to 91 %, a link bandwidth of 128 bit would be sufficient.

5.4.12 Comparison to Previous Work

This part compares the proposed algorithm with a previous approach [30] to show the advantages of certain concepts. To compare both algorithms, certain parameters were set to the same value: maximum capacity utilization ($C_{00} = 0.85$), maximum bandwidth utilization ($C_{01} = 1000$), maximum hop count ($C_{02} = 1$) and threads ($C_{03} = 4$). The computation of the fitness value for the bandwidth is partially commented out because the bandwidth is not considered in [30]. Both algorithms distribute the application graphs used in this evaluation to a 3 × 3 mesh topology and the irregular graph from Figure 4.16b.

On average, the proposed algorithm needed 1.19 times the computation time for the regular graph and 1.34 time for the irregular graph for a hop count of one. One difference between the algorithms is that [30] first improves the hops (dilation) and then the maximum capacity utilization. This leads to results with a maximum of two hops. On average, the proposed algorithm required 1.17 times less computation time for the regular graph and 1.54 times less for the irregular graph for a hop count of two. The proposed algorithm shortens the computation time for the irregular host graph by using a simplified algorithm that encloses the host in a square instead of trying to fit the graph into the irregular shape. The reason that the computation time for ($C_{02} = 1$) is longer is that the proposed algorithm continues to compute until it finds no more improvements and does not stop after 256 iterations [30].

The next test adjusts the parameters for the maximum capacity utilization ($C_{00} = 0.6$) and threads ($C_{03} = 8$). The test should show which algorithm achieves a better result on the regular host graph if it has enough memory resources or time available. For this purpose, the number of MTMs and STMs per LTM in the proposed algorithm was set to 255 and 16, respectively. Additionally, the number of iterations in [30] was set to 512 to achieve a similar computation time for both algorithms. The proposed algorithm obtained a better result in terms of maximum capacity utilization for all application graphs. It achieves a value that is on average 1.65 percentage points better than [30]. The next test sets the maximum number of iterations to 16 384, since [30] does not terminate. In this case, the proposed algorithm achieved either better or equally good results for the application graphs. On average, the maximum utilization was 0.814 % lower than in [30], although the computation time in [30] was 90 times higher.

There are further advantages compared to [30]. The load balancing phase achieves better results, due to the step that considers the utilization of the partitions. The load balancing phase requires fewer iterations, and the number of iterations required does not increase with the number of nodes in the application graph. The threads work in a grid and communicate with each other to avoid duplicate solutions and improve exploration. In addition, memory consumption is lower because the TS algorithms cleans up its history depending on the MTM size and SA. The algorithm does not discard the best intermediate solutions not taken and stores them in the STM. Furthermore, the load balancing phase stores intermediate solutions. The algorithm considers the bandwidth of the communication infrastructure. This has an impact on the execution time of applications running on this architecture. The algorithm considers not only the maximum values of the objectives, but also the average values. In terms of hops and bandwidth, it has the advantage that the communication infrastructure has a lower dynamic energy consumption.

5.5 Summary

To distribute the application graph on partition-based and mesh-like FPGA topologies, this thesis proposes `APARMAP` [29, 30]. It covers a much wider range of use cases and uses several heuristics and load balancing techniques to find a near-optimal solution. Therefore, this work examines and evaluates the algorithm and its concepts separately from the `DECISION` toolchain with respect to its optimality, performance, scalability, topology and more.

Thanks to the multilevel memory design, the proposed algorithm achieves a low memory footprint that scales linearly with the number of threads and nodes in the application graph. Finding a good initial solution by load balancing always improves the achieved fitness values of the optimization phase. It also reduces the computation time of the optimization phase for an increasing number of nodes and makes the algorithm scalable. The computation time of the load balancing phase is predictable because it is deterministic. Its complexity is $\mathcal{O}(n \cdot \log_2(n))$, due to a grid-based collision detection method.

The more nodes there are in the application graph, the more likely it is that a larger number of threads will find a better solution, as this improves exploration and minimizes the possibility of getting stuck in a local minimum. The algorithm computes 4.4 (2.9) times more MTM elements with 16 (8) threads. This is partly due to the sequential part of the algorithm, but also because different threads need to access common elements. As the number of nodes in the application graph increases, the influence of the MTM size on the final fitness value also increases. There seems to be a correlation between the MTM and the STM size.

The evaluation showed that the algorithm can deal with heterogeneous and irregular host graph topologies. Depending on the distribution of partitions and their resources, a regular and heterogeneous topology can achieve a better fitness value but takes a bit more time due to the different partition types. The results show the good scalability of the algorithm in terms of computation time for an increasing number of nodes and partitions. The algorithm was able to achieve an optimal placement for a set of example graphs up to a size of 196 nodes on host graphs of up to 49 partitions. For a real application with 271 nodes and 441 edges, the algorithm was able to achieve a distribution with low resource fragmentation in an average time of 149 ms.

6 Conclusion and Outlook

6.1 Summary of Contributions

This thesis has addressed the efficient programming of object detection algorithms on FPGA-based heterogeneous systems. Thereby, it reduces the research gap with its three main contributions. `HiFlipVX` enables the implementation of performance-optimized and resource-efficient object detection applications on FPGAs. In addition, this thesis investigated methods for optimization, automatic generation, and extraction of vision kernels using OpenCL for CPU, GPU, and FPGA architectures. `APARMAP` enables the efficient distribution of application graphs over heterogeneous FPGA-based mesh-like topologies in a scalable algorithm. The `DECISION` framework contains several modules that integrate `APARMAP` and `HiFlipVX`, abstract the underlying hardware and implementation details, and automate the generation of software and hardware code for two different types of platforms.

`HiFlipVX` is an open-source, resource and performance optimized HLS-based FPGA library containing 66 computer vision functions for object detection algorithms. It consumes on average only 0.39 % FFs and 0.32 % LUTs for a set of image processing functions, compared to the xfOpenCV (2017.4) library. It consumes on average 1.42 times less BRAM than xfOpenCV (2017.4) for selected filter functions. The many compile-time parameters make the library very parameterizable and are ideal for optimized designs and extensive DSE. Achieving a frequency of at least 300 MHz for all filter functions on the ZCU104 demonstrates its latency optimizations. Besides the increase in performance, the vectorization of its functions has further benefits. It improves the ratio of resource consumption to operations and enables the creation of an energy-efficient design using DVFS. Although it uses XILINX devices to optimize the library, it does not require any external libraries and puts effort into being as vendor independent as possible, as shown for Intel devices. The neural network extension features two parallelization options to balance between resource efficiency and performance. It shows speedups of up to 18.7 for individual layers of the MobileNets algorithm compared to the related work. Furthermore, an AlexNet layer achieved a speedup of 3.23 compared to a related work, while consuming 73 % less BRAM.

A unique feature and contribution of the library is its extension for feature detection algorithms. It contains six individual feature-based functions based on the Canny, FAST, ORB and AKAZE feature detectors. For example, the feature extraction function includes several selectable functionalities, such as NMS, SR, and more. In addition to the library extension, this work also contributes to the improvement of feature extraction algorithms. An optimized implementation based on the AKAZE detector and the FREAK descriptor achieves a repeatability of 72.57 %, while the next best combination of algorithms achieves only 62.99 %

when using the geometric mean, which weights the more complex cases stronger. The `HiFlipVX` implementation of AKAZE computes between 3.56 and 4.13 times more PPS than achieved by other researchers on FPGAs. At the same time, the resource consumption is comparable to that of optimized VHDL designs. Since the input feature vectors of the AKAZE compare function arrive sorted by their coordinates, the proposed implementation reduces its complexity from $\mathcal{O}(n^2)$ to approximately $\mathcal{O}(n \cdot \log_2 b)$ for n features and b buffer elements. The feature retain best function makes the execution time of the algorithm more predictable while reducing the computation time of the descriptor and improving repeatability. Using the FREAK descriptor and an extensive examination of various parameters, the ORB could also improve its repeatability by 7.67 % in software. A key feature of the FREAK hardware implementation is its pattern generator, which reduces the required bandwidth of the intensity computation by a factor of three with a minimal resource overhead.

`DECISION` is a modular framework for FPGA-based embedded and high-performance systems that enables efficient programming with an OpenVX-based frontend that integrates `HiFlipVX`. This frontend gives the user an easy-to-use interface to implement applications without getting involved with the underlying hardware and implementation details. To further improve efficiency, it creates and verifies the application graph, automatically sets and propagates parameters, computes a schedule to determine buffer sizes. To further speedup the design process it creates and synthesizes all IP-cores in parallel. The framework consists of two toolchains that target different hardware platforms, which is possible due to the modular and model-based design. The Embedded System Vision toolchain can create an application specific and adaptive NoC-based architecture or a pure AC design. It uses `APARMAP` to create the NoC-based architecture and automatically creates the hardware design including additional IP-cores. The streaming-optimized architecture enables the reusability of vision functions by multiple applications to improve the resource efficiency while maintaining high performance. Its adaptable NIs allow the reusability of ACs by different applications and a configurable DMA controller gives them direct memory access. The application flow and configuration of these components is orchestrated by a MA. For a set of example applications, the resource consumption was more than halved, while its overhead was only 0.015 % in terms of performance for a resolution of 1920 × 1080. The example design also runs at a high frequency of 300 MHz on the ZCU104 without performance loss.

The High-Performance Vision toolchain targets x86-based HPC systems, which can consist of CPUs, GPUs, and FPGAs. To create a representative schedule, this work considers device profiles, kernel profiles and estimates, FPGA dataflow characteristics, and OpenCL synchronization overhead. It takes care of finding the shortest transfer paths, data coherence and synchronization mechanisms at design time, even between different vendors. This and the parallelization of the command queues make the overhead of its runtime system negligible compared to OpenCL. For an application similar to the ORB algorithm, the toolchain was able to achieve speedups of up to 13.39 on an FPGA in comparison to a GPU. In a heterogeneous schedule with constrained FPGA resources, it achieved a speedup of 1.63 when using both devices. Additionally, this thesis looks at the integration of OpenCL-based libraries, automatic OpenCL kernel generation, and OpenCL kernel optimization and comparison for different architectures. The framework enables an easy integration of OpenCL-based libraries, which it demonstrated using OpenCV and AMDOVX. OpenCV is 2.04 times faster than AMDOVX on the tested AMD GPUs for a Gaussian filter, while at the same time not being limited to a vendor platform. This thesis implemented and compared different optimization strategies for OpenCL kernel on CPU, GPU and FPGA architectures to make general assumptions about the best strategy. The FPGA achieved the highest performance for the example application and is 1.46 times faster than the GPU, which is due to the efficient use of its memory bandwidth by streaming data. The proposed source-to-source compiler successfully built an OpenCL kernel from C++ code and transforms `PHI` statements into local variables. The generated code achieves a speedup of up to 60 on a dual CPU system with 20 cores (40 threads) and the tiling optimization strategy enabled. A comparison of the `DECISION` framework with the SoA in Table 2.11 shows that no other work simultaneously: (1) has such a high abstraction level, (2) has such a rich set of vision functions, (3) addresses CPUs, GPUs, and FPGAs, (4) provides heterogeneous scheduling and mapping with automatic device and kernel profiling, (5) and includes a memory model and runtime system.

`APARMAP` is a scalable application distribution algorithm for partition-based and mesh-like topologies that uses load balancing techniques and heuristics in a multithreaded grid-based algorithm to generate an application-specific hardware architecture. Its constraints and objectives are the FPGA resource utilization, NoC bandwidth consumption, NoC hop count, and execution time of the proposed algorithm. The evaluation showed that using the proposed load balancing techniques to compute an initial solution has several advantages. It improves the achieved fitness values of the overall algorithm, reduces its computation time for an increasing number of nodes, and makes the algorithm scalable. The computation time of the load balancing phase is predictable and has a complexity of $\mathcal{O}(n \cdot \log_2 n)$, since it is deterministic and based on a grid-based collision detection method. The proposed multilevel memory design has a low memory footprint that scales linearly with the number of threads and nodes in the application graph. As the number of nodes in the application graph increases, the influence of the MTM size on the final fitness value also increases. In addition, the evaluation shows a correlation between MTM and STM size.

The proposed grid-based multithreaded approach not only computes more solutions per second, but also improves the exploration of the algorithm, minimizing the possibility of getting stuck in a local minimum. The more nodes there are in the application graph, the more likely it is that a larger number of threads will find a better solution. The algorithm computes 4.4 (2.9) times more MTM elements with 16 (8) threads. It does not scale completely with the number of threads, due to the sequential part of the algorithm and the shared regions. The evaluation shows the good scalability of the algorithm in terms of computation time for an increasing number of nodes and partitions. It also demonstrates its ability to manage heterogeneous and irregular host graph topologies. The algorithm was able to achieve an optimal placement for a set of example graphs up to a size of 196 nodes on host graphs of up to 49 partitions. For a real application with 271 nodes and 441 edges, the algorithm was able to achieve a distribution with low resource fragmentation in an average time of 149 ms. The combination of `APARMAP` with `HiFlipVX` and `DECISION` is the only work that deals at the same time with the problem of partitioning, tuning, mapping, scheduling, clustering, and placement of application graphs on reconfigurable devices compared to the SoA in Table 2.4.

6.2 Future Work

One problem with HLS estimates is that they can be quite inaccurate in terms of resource consumption. The Vivado synthesis results of the individual functions are significantly more accurate, but the additional time would be enormous. Therefore, more accurate and faster estimates of the resource consumption of the `HiFlipVX` functions could be very helpful for the middleend of the toolchains. These could be generated, for example, using analytical approaches, trained neural networks, or a combination of both. The `DECISION` framework could be used to automatically generate the data for training the neural networks.

`APARMAP` considers the scheduling and mapping of virtual tasks to physical nodes and the subsequent clustering and placement of these nodes on hardware separately. A future work could bring these processes closer together and merge them into one algorithm. However, changing the mapping of virtual tasks, while placing their physical nodes, is a complex task. This would further drive the creation of application-specific architectures. In addition, the process of tuning application parameters and partitioning them into a task graph could be integrated into this algorithm to further maximize system utilization.

The `DECISION` framework uses two toolchains with the same frontend to address different architectures from the embedded and HPC domains. A future work could bring these two domains closer together, by distributing applications among them. This could enable the distribution of applications between small edge-devices and large cloud computing systems. For example, a base station could more easily take over the computations of a drone to make it more energy efficient and reduce its weight. On the other hand, existing resources in various embedded systems could be better utilized by using them for other computations.

The use of OpenCL allows the use of a wide range of compute devices. However, there are other C++-based APIs that are worth integrating into the framework and its runtime system, to improve kernel performance and add more vision-based libraries. On one hand would be CUDA for NVIDIA GPUs. On the other hand, it requires standards like MPI to drive larger compute clusters consisting of many compute nodes. Furthermore, there are many efficient C++-based extensions for parallel programming on CPUs.

Bibliography

[1] T. Kalb, L. Kalms, D. Göhringer, C. Pons, F. Marty, A. Muddukrishna, M. Jahre, P. G. Kjeldsberg, B. Ruf, T. Schuchert, I. Tchouchenkov, C. Ehrenstrahle, M. Peterson, F. Christensen, A. Paolillo, C. Lemer, B. Rodriguez, G. Bernard, F. Duhem, and P. Millet. "TULIPP: Towards Ubiquitous Low-Power Image Processing Platforms". In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, July 2016, pp. 306–311. DOI: `10.1109/SAMOS.2016.7818363`.

[2] L. Kalms, J. Rettkowski, M. Hamme, and D. Göhringer. "Robust Lane Recognition for Autonomous Driving". In: *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Sept. 2017, pp. 1–6. DOI: `10.1109/DASIP.2017.8122130`.

[3] B. Ruf, S. Monka, M. Kollmann, and M. Grinberg. "Real-Time on-Board Obstacle Avoidance for UAVs Based on Embedded Stereo Vision". In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (ISPRS)* (July 2018), pp. 363–370. DOI: `10.5194/isprs-archives-XLII-1-363-2018`.

[4] T. Kalb, L. Kalms, D. Göhringer, C. Pons, A. Muddukrishna, M. Jahre, B. Ruf, T. Schuchert, I. Tchouchenkov, C. Ehrenstråhle, M. Peterson, F. Christensen, A. Paolillo, B. Rodriguez, and P. Millet. "Developing Low-Power Image Processing Applications with the TULIPP Reference Platform Instance". In: *Hardware Accelerators in Data Centers*. Springer, Aug. 2018, pp. 181–197. DOI: `10.1007/978-3-319-92792-3_10`.

[5] S. Said, L. Kalms, D. Göhringer, and M.A.A. El Ghany. "Hardware/Software-Codesign for Hand Gestures Recognition using a Convolutional Neural Network". In: *INTelligent Embedded Systems Architectures and Applications Workshop (INTENSA)*. ACM, Oct. 2019, pp. 23–28. DOI: `10.1145/3372394.3372395`.

[6] M. Hernández, A. Del Barrio, and G. Botella. "An Ultra Low-Cost Cluster Based on Low-End FPGAs". In: *Computer Simulation Conference (SummerSim)*. Society for Computer Simulation International, July 2018, pp. 1–12. DOI: `10.22360/summersim.2018.scsc.034`.

[7] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2016, pp. 1–4. DOI: `10.1109/FPL.2016.7577314`.

[8] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Feb. 2017, pp. 5–14. DOI: `10.1145/3020078.3021740`.

[9] L. Kalms and D. Göhringer. "Exploration of OpenCL for FPGAs using SDAccel and Comparison to GPUs and Multicore CPUs". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2017, pp. 1–4. DOI: `10.1109/HPCA.2016.7446058`.

[10] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: *International Conference on Embedded Software and Systems (ICESS)*. IEEE, June 2019, pp. 1–8. DOI: `10.1109/ICESS.2019.8782524`.

[11] A. Bhatele and L. V. Kale. "Heuristic-Based Techniques for Mapping Irregular Communication Graphs to Mesh Topologies". In: *International Conference on High Performance Computing and Communications*. IEEE, Sept. 2011, pp. 765–771. DOI: `10.1109/HPCC.2011.109`.

[12] L. Kalms, A. Podlubne, and D. Göhringer. "HiFlipVX: an Open Source High-Level Synthesis FPGA Library for Image Processing". In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, Apr. 2019, pp. 149–164. DOI: `10.1007/978-3-030-17227-5_12`.

[13] L. Kalms and D. Göhringer. "Accelerated High-level Synthesis Feature Detection for FPGAs using HiFlipVX". In: *Towards Ubiquitous Low-power Image Processing Platforms*. Springer, Jan. 2021, pp. 115–135. DOI: `10.1007/978-3-030-53532-2_7`.

[14] L. Kalms, P. Amini Rad, M. Ali, and A. Iskander D. Göhringer. "A Parametrizable High-Level Synthesis Library for Accelerating Neural Networks on FPGAs". In: *Journal of Signal Processing Systems (JSPS)* 93.5 (May 2021), pp. 1–27. DOI: `10.1007/s11265-021-01651-5`.

[15] M. A. Davila-Guzman, R. Gran Tejero, M. Villarroya-Gaud, D. Suarez Gracia, L. Kalms, and D. Göhringer. "A Cross-Platform OpenVX Library for FPGA Accelerators". In: *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, Mar. 2021, pp. 75–83. DOI: `10.1109/PDP52278.2021.00020`.

[16] M. A. Davila-Guzman, L. Kalms, R. Gran Tejero, M. Villarroya-Gaud, D. Suarez Gracia, and D. Göhringer. "A Cross-Platform OpenVX Library for FPGA Accelerators". In: *Journal of Systems Architecture (JSA)*. Elsevier, Feb. 2022, pp. 1–12. DOI: `10.1016/j.sysarc.2021.102372`.

[17] E. Rosten and T. Drummond. "Machine Learning for High-Speed Corner Detection". In: *European Conference on Computer Vision (ECCV)*. Springer, May 2006, pp. 430–443. DOI: `10.1007/11744023_34`.

[18] J. Canny. "A Computational Approach to Edge Detection". In: *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (Nov. 1986), pp. 679–698. DOI: `10.1109/TPAMI.1986.4767851`.

[19] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. "ORB: An Efficient Alternative to SIFT or SURF". In: *International Conference on Computer Vision (ICCV)*. IEEE, Nov. 2011, pp. 2564–2571. DOI: `10.1109/ICCV.2011.6126544`.

[20] J. Nuevo P. F. Alcantarilla and A. Bartoli. "Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces". In: *British Machine Vision Conference (BMVC)*. BMVA Press, Sept. 2013, pp. 1–11. DOI: `10.5244/C.27.13`.

[21] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *ArXiv* (Apr. 2017), pp. 1–9. DOI: `arXiv:1704.04861`.

[22] L. Kalms, M. Hajduk, and D. Göhringer. "Efficient Pattern Recognition Algorithm Including a Fast Retina Keypoint FPGA Implementation". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2019, pp. 121–128. DOI: `10.1109/FPL.2019.00028`.

[23] M. Nickel, L. Kalms, T. Häring, and D. Göhringer. "High-Performance AKAZE Implementation Including Parametrizable and Generic HLS Modules". In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, July 2022, pp. 139–147. DOI: `10.1109/ASAP54787.2022.00031`.

[24] A. Alahi, R. Ortiz, and P. Vandergheynst. "FREAK: Fast Retina Keypoint". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2012, pp. 510–517. DOI: `10.1109/CVPR.2012.6247715`.

[25] L. Kalms, K. Mohamed, and D. Göhringer. "Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA". In: *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. Best Paper. ACM, June 2017, pp. 1–6. DOI: `10.1145/3120895.3120898`.

[26] L. Kalms, H. Ibrahim, and D. Göhringer. "Full-HD Accelerated and Embedded Feature Detection Video System with 63fps using ORB for FREAK". In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2018, pp. 1–6. DOI: `10.1109/RECONFIG.2018.8641706`.

[27] L. Kalms, T. Häring, and D. Göhringer. "DECISION: Distributing OpenVX Applications on CPUs, GPUs and FPGAs using OpenCL". In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2022, pp. 84–91. DOI: `10.1109/IPDPSW55747.2022.00023`.

[28] L. Kalms, T. Hebbeler, and D. Göhringer. "Automatic OpenCL Code Generation from LLVM-IR using Polyhedral Optimization". In: *9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)*. ACM, Jan. 2018, pp. 45–50. DOI: `10.1145/3183767.3183779`.

[29] L. Kalms and D. Göhringer. "Clustering and Mapping Algorithm for Application Distribution on a Scalable FPGA Cluster". In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Best Paper. IEEE, May 2016, pp. 105–113. DOI: `10.1109/IPDPSW.2016.75`.

[30] L. Kalms and D. Göhringer. "Scalable Clustering and Mapping Algorithm for Application Distribution on Heterogeneous and Irregular FPGA Clusters". In: *Journal of Parallel and Distributed Computing (JPDC)* (Nov. 2019), pp. 367–376. DOI: `10.1016/j.jpdc.2018.02.033`.

[31] R. Mur-Artal and J. D. Tardós. "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras". In: *Transactions on Robotics (T-RO)* (Oct. 2017), pp. 1255–1262. DOI: `10.1109/TRO.2017.2705103`.

[32] R. Giduthuri and K. Pulli. “OpenVX: A Framework for Accelerating Computer Vision”. In: *International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, Sept. 2016, pp. 1–50. DOI: `10.1145/2988458.2988513`.

[33] D. Göhringer, M. Birk, Y. Dasse-Tiyo, N. Ruiter, M. Hübner, and J. Becker. “Reconfigurable MPSoC versus GPU: Performance, power and energy evaluation”. In: *International Conference on Industrial Informatics (INDIN)*. IEEE, July 2011, pp. 848–853. DOI: `10.1109/INDIN.2011.6035003`.

[34] B.S. Manjunath, C. Shekhar, and R. Chellappa. “A New Approach to Image Feature Detection with Applications”. In: *Pattern Recognition* (Apr. 1996), pp. 627–640. DOI: `10.1016/0031-3203(95)00115-8`.

[35] A. I. Awad and M. Hassaballah. *Image Feature Detectors and Descriptors*. Springer, Feb. 2016. DOI: `10.1007/978-3-319-28854-`.

[36] E. Rosten and T. Drummond. “Fusing Points and Lines for High Performance Tracking”. In: *International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2005, pp. 1508–1515. DOI: `10.1109/ICCV.2005.104`.

[37] E. Rosten, R. Porter, and T. Drummond. “Faster and Better: A Machine Learning Approach to Corner Detection”. In: *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (Jan. 2010), pp. 105–119. DOI: `10.1109/TPAMI.2008.275`.

[38] C. Harris and M. Stephens. “A Combined Corner and Edge Detector”. In: *Alvey Vision Conference (AVC)*. Alvety Vision Club, 1988, pp. 147–152. DOI: `10.5244/C.2.23`.

[39] D. G. Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”. In: *International Journal of Computer Vision (IJCV)* (Nov. 2004), pp. 91–110. DOI: `10.1023/B:VISI.0000029664.99615.94`.

[40] H. Bay, T. Tuytelaars, and L. Van Gool. “SURF: Speeded Up Robust Features”. In: *European Conference on Computer Vision (ECCV)*. Springer, May 2006, pp. 404–417. DOI: `10.1007/11744023_32`.

[41] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. “BRIEF: Binary Robust Independent Elementary Features”. In: *European Conference on Computer Vision (ECCV)*. Springer, Sept. 2010, pp. 778–792. DOI: `10.1007/978-3-642-15561-1_56`.

[42] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua. “BRIEF: Computing a Local Binary Descriptor Very Fast”. In: *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (Nov. 2012), pp. 1281–1298. DOI: `10.1109/TPAMI.2011.222`.

[43] S. Leutenegger, M. Chli, and R. Y. Siegwart. “BRISK: Binary Robust Invariant Scalable Keypoints”. In: *International Conference on Computer Vision (ICCV)*. IEEE, Nov. 2011, pp. 2548–2555. DOI: `10.1109/ICCV.2011.6126542`.

[44] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool. “A Comparison of Affine Region Detectors”. In: *International Journal of Computer Vision (IJCV)* (Oct. 2005), pp. 43–72. DOI: `10.1007/s11263-005-3848-x`.

[45] D. Dwarakanath, C. Griwodz, and P. Halvorsen. “Robustness of 3D Point Positions to Camera Baselines in Markerless AR Systems”. In: *International Conference on Multimedia Systems (MMSys)*. ACM, May 2016, pp. 1–12. DOI: `10.1145/2910017.2910611`.

[46] P. F. Alcantarilla, A. Bartoli, and A. J. Davison. “KAZE Features”. In: *European Conference on Computer Vision (ECCV)*. Springer, Oct. 2012, pp. 1–14. DOI: `10.1007/978-3-642-33783-3_16`.

[47] M. Song, Y. Cao, C. Yu, J. An, and C. Chang. “Solar Image Matching Based on Improved Freak Algorithm”. In: *International Conference on Machine Learning and Cybernetics (ICMLC)*. IEEE, July 2018, pp. 126–131. DOI: `10.1109/ICMLC.2018.8527012`.

[48] P. L. Rosin. “Measuring Corner Properties”. In: *Computer Vision and Image Understanding* (Feb. 1999), pp. 291–307. DOI: `10.1006/cviu.1998.0719`.

[49] K. Y. Lee. “A Design of an Optimized ORB Accelerator for Real-Time Feature Detection”. In: *International Journal of Control and Automation (IJCA)* (Mar. 2014), pp. 15–61. DOI: `10.14257/IJCA.2014.7.3.20`.

[50] M. Fularz, M. Kraft, A. Schmidt, and A. Kasiski. “A High-Performance FPGA-Based Image Feature Detector and Matcher Based on the FAST and BRIEF Algorithms”. In: *International Journal of Advanced Robotic Systems (IJARS)* (Oct. 2015), pp. 1–15. DOI: `10.5772/61434`.

[51] G. Jiang, L. Liu, W. Zhu, S. Yin, and S. Wei. “A 127 Fps in Full Hd Accelerator Based on Optimized AKAZE with Efficiency and Effectiveness for Image Feature Extraction”. In: *Design Automation Conference (DAC)*. ACM, June 2015, pp. 1–6. DOI: `10.1145/2744769.2744772`.

[52] E. D. Bello and P. A. Salvadeo. “An Image Descriptors Extraction Hardware-Architecture Inspired on Human Retina”. In: *Southern Conference on Programmable Logic (SPL)*. IEEE, Nov. 2014, pp. 1–6. DOI: `10.1109/SPL.2014.7002205`.

[53] J. Zhao. “Masters Thesis: Video/Image Processing on FPGA”. Apr. 2015, pp. 1–81.

[54] K. Y. Lee and K. J. Byun. “A Hardware Design of Optimized ORB Algorithm with Reduced Hardware Cost”. In: Dec. 2013, pp. 58–62. DOI: `10.14257/ASTL.2013.43.11`.

[55] C. He, A. Papakonstantinou, and D. Chen. “A Novel SoC Architecture on FPGA for Ultra Fast Face Detection”. In: *International Conference on Computer Design (ICCD)*. IEEE, Oct. 2009, pp. 412–418. DOI: `10.1109/ICCD.2009.5413122`.

[56] H. Lai, M. Savvides, and T. Chen. “Proposed FPGA Hardware Architecture for High Frame Rate (100 fps) Face Detection Using Feature Cascade Classifiers”. In: *International Conference on Biometrics: Theory, Applications, and Systems (BTAS)*. IEEE, Sept. 2007, pp. 1–6. DOI: `10.1109/BTAS.2007.4401930`.

[57] J. Svab, T. Krajnik, J. Faigl, and L. Preucil. “FPGA based Speeded Up Robust Features”. In: *International Conference on Technologies for Practical Robot Applications (TePRA)*. IEEE, Nov. 2009, pp. 35–41. DOI: `10.1109/TEPRA.2009.5339646`.

[58] S. V. Chakrasali and S. Kuthale. “Optimized Face Detection on FPGA”. In: *International Conference on Circuits, Controls, Communications and Computing (I4C)*. IEEE, Oct. 2016, pp. 1–6. DOI: `10.1109/CIMCA.2016.8053269`.

[59] L. Wanhammar. “11 - Processing Elements”. In: *DSP Integrated Circuits*. Elsevier, 1999, pp. 461–530. DOI: `10.1016/B978-012734530-7/50011-8`.

[60] R. Buyya, C. Vecchiola, and T. S. Somasundaram. “Chapter 2 - Principles of Parallel and Distributed Computing”. In: *Mastering Cloud Computing*. Elsevier, 2013, pp. 29–70. DOI: `10.1016/B978-0-12-411454-8.00002-4`.

[61] A. K. Singh, A. Kumar, T. Srikanthan, and Y. Ha. “Mapping Real-Life Applications on Run-Time Reconfigurable NoC-Based MPSoC on FPGA”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2010, pp. 365–368. DOI: `10.1109/FPT.2010.5681427`.

[62] J. Delorme and D. Houzet. “A Complete 4G Radiocommunication Application Mapping onto a 2D Mesh NoC Architecture”. In: *North-East Workshop on Circuits and Systems (NEWCAS)*. IEEE, June 2006, pp. 93–96. DOI: `10.1109/NEWCAS.2006.250955`.

[63] S. Bayar and A. Yurdakul. “An Efficient Mapping Algorithm on 2-D Mesh Network-on-Chip with Reconfigurable Switches”. In: *International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, Apr. 2016, pp. 1–4. DOI: `10.1109/DTIS.2016.7483808`.

[64] D. Göhringer, M. Hübner, V. Schatz, and J. Becker. “Runtime Adaptive Multi-Processor System-on-Chip: RAMPSoC”. In: *International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, Apr. 2008, pp. 1–7. DOI: `10.1109/IPDPS.2008.4536503`.

[65] J. Rettkowski and D. Göhringer. “RAR-NoC: A Reconfigurable and Adaptive Routable Network-on-Chip for FPGA-Based Multiprocessor Systems”. In: *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2014, pp. 1–6. DOI: `10.1109/ReConFig.2014.7032552`.

[66] N. Kapre and J. Gray. “Hoplite: Building Austere Overlay NoCs for FPGAs”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2015, pp. 1–8. DOI: `10.1109/FPL.2015.7293956`.

[67] J. Siast, A. uczak, and M. Domaski. “RingNet: A Memory-Oriented Network-On-Chip Designed for FPGA”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* (June 2019), pp. 1284–1297. DOI: `10.1109/TVLSI.2019.2899575`.

[68] L. Ost, G. M. Almeida, M. Mandelli, E. Wachter, S. Varyani, G. Sassatelli, L. S. Indrusiak, M. Robert, and F. Moraes. “Exploring Heterogeneous NoC-Based MPSoCs: From FPGA to High-Level Modeling”. In: *International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, June 2011, pp. 1–8. DOI: `10.1109/ReCoSoC.2011.5981517`.

[69] S. K. Rethinagiri, O. Palomar, J. A. Moreno, O. Unsal, and A. Cristal. “Trigeneous Platforms for Energy Efficient Computing of HPC Applications”. In: *International Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2015, pp. 264–274. DOI: `10.1109/HiPC.2015.19`.

[70] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers. “OmpSsZynq All-Programmable SoC Ecosystem”. In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Feb. 2014, pp. 137–146. DOI: `10.1145/2554688.2554777`.

[71] K. Vipin and S. A. Fahmy. “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications”. In: *Computing Surveys (CSUR)* (July 2018), pp. 1–39. DOI: `10.1145/3193827`.

[72] O. Knodel, A. Georgi, P. Lehmann, W. E. Nagel, and R. G. Spallek. “Integration of a Highly Scalable, Multi-FPGA-Based Hardware Accelerator in Common Cluster Infrastructures”. In: *International Conference on Parallel Processing (ICPP)*. IEEE, Dec. 2013, pp. 893–900. DOI: `10.1109/ICPP.2013.106`.

[73] Z. Lin and P. Chow. “ZCluster: A Zynq-based Hadoop cluster”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2013, pp. 450–453. DOI: `10.1109/FPT.2013.6718411`.

[74] P. Moorthy and N. Kapre. “Zedwulf: Power-Performance Tradeoffs of a 32-Node Zynq SoC Cluster”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2015, pp. 68–75. DOI: `10.1109/FCCM.2015.37`.

[75] A. Theodore Markettos, P. J. Fox, S. W. Moore, and A. W. Moore. “Interconnect for Commodity FPGA Clusters: Standardized or Customized?” In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Oct. 2014, pp. 1–8. DOI: `10.1109/FPL.2014.6927472`.

[76] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. “A Cloud-Scale Acceleration Architecture”. In: *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, Oct. 2016, pp. 1–13. DOI: `10.1109/MICRO.2016.7783710`.

[77] X. Bai, L. Jiang, Q. Dai, J. Yang, and J. Tan. “Acceleration of RSA processes based on hybrid ARM-FPGA cluster”. In: *Symposium on Computers and Communications (ISCC)*. IEEE, July 2017, pp. 682–688. DOI: `10.1109/ISCC.2017.8024607`.

[78] T. Ueno, T. Miyajima, A. Mondigo, and K. Sano. “Hybrid Network Utilization for Efficient Communication in a Tightly Coupled FPGA Cluster”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2019, pp. 1–4. DOI: `10.1109/ICFPT47387.2019.00068`.

[79] K. Takano, T. Oda, R. Ozaki, A. Uejima, and M. Kohata. “Implementation of Distributed Processing Using a PC-FPGA Hybrid System”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2019, pp. 387–390. DOI: `10.1109/ICFPT47387.2019.00074`.

[80] J. Hou, Y. Zhu, L. Kong, Z. Wang, S. Du, S. Song, and T. Huang. “A Case Study of Accelerating Apache Spark with FPGA”. In: *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) and International Conference On Big Data Science And Engineering (BigDataSE)*. IEEE, Aug. 2018, pp. 855–860. DOI: `10.1109/TrustCom/BigDataSE.2018.00123`.

[81] Y. Osana and Y. Sakamoto. “Performance Evaluation of a CPU-FPGA Hybrid Cluster Platform Prototype”. In: *International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, June 2017, pp. 1–6. DOI: `10.1145/3120895.3120917`.

[82] AVNET. *ZedBoard - Hardware Users Guide*. Jan. 2014.

[83] N. Mentens, J. Vandorpe, J. Vliegen, A. Braeken, B. Silva, A. Touhafi, S. Knappmann, A. Kern, J. Rettkowski, M. Kadi, D. Göhringer, and M. Hübner. “DynamIA: Dynamic Hardware Reconfiguration in Industrial Applications”. In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, Mar. 2015, pp. 513–518. DOI: `10.1007/978-3-319-16214-0_47`.

[84] M. Owaida and G. Alonso. “Application Partitioning on FPGA Clusters: Inference over Decision Tree Ensembles”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2018, pp. 295–2955. DOI: `10.1109/FPL.2018.00057`.

[85] M. Owaida, A. Kulkarni, and G. Alonso. “Distributed Inference over Decision Tree Ensembles on Clusters of FPGAs”. In: *Transactions on Reconfigurable Technology and Systems (TRETS)* (Sept. 2019), pp. 1–28. DOI: `10.1145/3340263`.

[86] J. Sheng, Q. Xiong, C. Yang, and M. C. Herbordt. “Collective Communication on FPGA Clusters with Static Scheduling”. In: *Special Interest Group on Computer Architecture (SIGARCH)* (Jan. 2017), pp. 2–7. DOI: `10.1145/3039902.3039904`.

[87] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong. “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM, Aug. 2016, pp. 326–331. DOI: `10.1145/2934583.2934644`.

[88] A. Wu, X. Jin, X. Du, and S. Guo. “A Flexible FPGA-to-FPGA Communication System”. In: *International Conference on Advanced Communication Technology (ICACT)*. IEEE, Feb. 2017, pp. 836–843. DOI: `10.23919/ICACT.2017.7890234`.

[89] X. Y. Niu, K. H. Tsoi, and W. Luk. “Reconfiguring Distributed Applications in FPGA Accelerated Cluster with Wireless Networking (FPL)”. In: *International Conference on Field Programmable Logic and Applications*. IEEE, Oct. 2011, pp. 545–550. DOI: `10.1109/FPL.2011.106`.

[90] O. Knodel and R. G. Spallek. “Integration of a Multi-FPGA System in a Common Cluster Environment”. In: *International Conference on Field programmable Logic and Applications (FPL)*. IEEE, Oct. 2013, pp. 1–2. DOI: `10.1109/FPL.2013.6645612`.

[91] P. J. Fox, A. T. Markettos, and S. W. Moore. “Reliably Prototyping Large SoCs Using FPGA Clusters”. In: *International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, May 2014, pp. 1–8. DOI: `10.1109/ReCoSoC.2014.6861350`.

[92] S. Buscemi and R. Sass. “Design and Utilization of an FPGA Cluster to Implement a Digital Wireless Channel Emulator”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Oct. 2012, pp. 635–638. DOI: `10.1109/FPL.2012.6339253`.

[93] Y. Kono, K. Sano, and S. Yamamoto. “Scalability Analysis of Tightly-Coupled FPGA-Cluster for Lattice Boltzmann Computation”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Oct. 2012, pp. 120–127. DOI: `10.1109/FPL.2012.6339275`.

[94] Y. Thoma, A. Dassatti, and D. Molla. “FPGA2: An Open Source Framework for FPGA-GPU PCIe Communication”. In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2013, pp. 1–6. DOI: `10.1109/ReConFig.2013.6732296`.

[95] R. Bittner and E. Ruf. “Direct GPU/FPGA Communication via PCI Express”. In: *International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, Sept. 2012, pp. 135–139. DOI: `10.1109/ICPPW.2012.20`.

[96] M. Hübner, D. Göhringer, J. Noguera, and J. Becker. “Fast Dynamic and Partial Reconfiguration Data Path with Low Hardware Overhead on Xilinx FPGAs”. In: *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, Mar. 2010, pp. 1–8. DOI: `10.1109/IPDPSW.2010.5470736`.

[97] Takaaki Miyajima, Tomoya Hirao, Naoya Miyamoto, Jeongdo Son, and Kentaro Sano. "A Software Bridged Data Transfer on a FPGA Cluster by Using Pipelining and InfiniBand Verbs". In: *nternational Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, June 2019, pp. 1–6. DOI: `10.1145/3337801.3337808`.

[98] A. Mondigo, T. Ueno, D. Tanaka, K. Sano, and S. Yamamoto. "Design and Scalability Analysis of Bandwidth-Compressed Stream Computing with Multiple FPGAs". In: *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, July 2017, pp. 1–8. DOI: `10.1109/ReCoSoC.2017.8016148`.

[99] NVIDIA. *Whitepaper: NVIDIA Tegra Multi-processor Architecture*. Feb. 2010.

[100] Xilinx. *Zynq-7000 SoC Data Sheet: Overview*. July 2018.

[101] F. Robino and J. Öberg. "From Simulink to NoC-based MPSoC on FPGA". In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, Mar. 2014, pp. 1–4. DOI: `10.7873/DATE.2014.341`.

[102] S. Bandaru and K. Deb. "Metaheuristic Techniques". In: ed. by R. N. Sengupta, A. Gupta, and J. Dutta. Taylor & Francis, Feb. 2016. Chap. Decision Sciences: Theory and Practice, pp. 693–750.

[103] R. E. Burkard, E. Çela, P. M. Pardalos, and L. S. Pitsoulis. "Handbook of Combinatorial Optimization". In: ed. by D. Z. Du and P. M. Pardalos. Springer, 1998. Chap. The Quadratic Assignment Problem, pp. 1713–1809. DOI: `10.1007/978-1-4613-0303-9_27`.

[104] T. Weise. *Metaheuristic Optimization: 7. Simulated Annealing*. Hefei University. 2021.

[105] T. Weise. *Metaheuristic Optimization: 8. Tabu Search*. Hefei University. 2021.

[106] T. Weise. *Metaheuristic Optimization: 10. Genetic Algorithms*. Hefei University. 2021.

[107] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. "Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends". In: *Design Automation Conference (DAC)*. IEEE, Apr. 2013, pp. 1–10. DOI: `10.1145/2463209.2488734`.

[108] H. R. Lewis. *Computers and intractability. A guide to the theory of NP-completeness*. 1983.

[109] A. Goens, R. Khasanov, J. Castrillon, M. Hähnel, T. Smejkal, and H. Härtig. "TETRiS: A Multi-Application Run-Time System for Predictable Execution of Static Mappings (SCOPES)". In: *International Workshop on Software and Compilers for Embedded Systems*. ACM, June 2017, pp. 11–20. DOI: `10.1145/3078659.3078663`.

[110] D. Grewe and M. F. P. O'Boyle. "A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL". In: *Compiler Construction*. Springer, Apr. 2011, pp. 286–305. DOI: `10.1007/978-3-642-19861-8_16`.

[111] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)". In: *International Symposium on Computer Architecture (ISCA)*. IEEE, June 2012, pp. 213–224. DOI: `10.1109/ISCA.2012.6237019`.

[112] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng. "Comparative Evaluation Of The Robustness Of DAG Scheduling Heuristics". In: *Grid Computing*. Springer, 2008, pp. 73–84. DOI: `10.1007/978-0-387-09457-1_7`.

[113] H. Topcuoglu, S. Hariri, and M.-Y. Wu. “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing”. In: *Transactions on Parallel and Distributed Systems (TPDS)* (Aug. 2002), pp. 260–274. DOI: `10.1109/71.993206`.

[114] J. G. Filho, M. Strum, and W. J. Chau. “Using Genetic Algorithms for Hardware Core Placement and Mapping in NoC-Based Reconfigurable Systems”. In: *International Journal of Reconfigurable Computing (IJRC)* (Feb. 2015), pp. 1–13. DOI: `10.1155/2015/902925`.

[115] I. Beretta, V. Rana, D. Atienza, and D. Sciuto. “Run-Time Mapping of Applications on FPGA-Based Reconfigurable Systems”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE, Aug. 2010, pp. 3329–3332. DOI: `10.1109/ISCAS.2010.5537893`.

[116] N. P. Mand, F. Robino, and J. Öberg. “Artificial Neural Network Emulation on NoC Based Multi-Core FPGA Platform”. In: *NORCHIP*. IEEE, Nov. 2012, pp. 1–4. DOI: `10.1109/NORCHP.2012.6403122`.

[117] M. S. Mohammed, J. Wei Tang, A. A. Ab Rahman, N. Paraman, and M. N. Marsono. “Rapid Prototyping of NoC-based MPSoC Based on Dataflow Modeling of Real-World Applications”. In: *Control and System Graduate Research Colloquium (ICSGRC)*. IEEE, Aug. 2018, pp. 217–222. DOI: `10.1109/ICSGRC.2018.8657542`.

[118] G. Zhu and Y. Wang. “A Multi-Application Mapping Case Study for NoC-Based MPSoCs”. In: *International Conference on Signal Processing, Communication and Computing (ICSPCC)*. IEEE, Aug. 2013, pp. 1–6. DOI: `10.1109/ICSPCC.2013.6664092`.

[119] F. Robino and J. Öberg. “The HeartBeat Model: A Platform Abstraction Enabling Fast Prototyping of Real-Time Applications on NoC-based MPSoC on FPGA”. In: *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, July 2013, pp. 1–8. DOI: `10.1109/ReCoSoC.2013.6581536`.

[120] M. Mandelli, L. Ost, E. Carara, G. Guindani, T. Gouvea, G. Medeiros, and F. G. Moraes. “Energy-Aware Dynamic Task Mapping for NoC-Based MPSoCs”. In: *International Symposium of Circuits and Systems (ISCAS)*. IEEE, May 2011, pp. 1676–1679. DOI: `10.1109/ISCAS.2011.5937903`.

[121] S. Bayar and A. Yurdakul. “PFMAP: Exploitation of Particle Filters for Network-on-Chip Mapping”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* (Oct. 2015), pp. 2116–2127. DOI: `10.1109/TVLSI.2014.2360791`.

[122] K. Pang, V. Fresse, S. Yao, and O. A. De Lima. “Task Mapping and Mesh Topology Exploration for an FPGA-Based Network on Chip”. In: *Microprocessors and Microsystems (MICPRO)* (May 2015), pp. 189–199. DOI: `10.1016/j.micpro.2015.03.006`.

[123] B. N. K. Reddy and Sireesha. “An Energy-Efficient Core Mapping Algorithm on Network on Chip (NoC)”. In: *International Symposium on VLSI Design and Test (VDAT)*. Springer, Jan. 2019, pp. 631–640. DOI: `10.1007/978-981-13-5950-7_52`.

[124] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. “Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks”. In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Feb. 2015, pp. 161–170. DOI: `10.1145/2684746.2689060`.

[125] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016, pp. 770–778. DOI: `10.1109/CVPR.2016.90`.

[126] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt. “FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2018, pp. 81–84. DOI: `10.1109/FCCM.2018.00021`.

[127] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt. “A Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Work and Weight Load Balancing”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2018, pp. 1–5. DOI: `10.1109/FPL.2018.00074`.

[128] A. Krizhevsky, I. Sutskever, and G. Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. In: *Communications of the ACM* (May 2017), pp. 84–90. DOI: `10.1145/3065386`.

[129] K. Bouaziz, S. Chtourou, Z. Marrakchi, M. Abid, and A. Obeid. “Exploration of Clustering Algorithms effects on Mesh of Clusters based FPGA Architecture Performance”. In: *International Conference on High Performance Computing Simulation (HPCS)*. IEEE, July 2019, pp. 658–665. DOI: `10.1109/HPCS48598.2019.9188138`.

[130] S. Chtourou, Z. Marrakchi, E. Amouri, V. Pangracious, M. Abid, and H. Mehrez. “Improvement of Cluster-Based Mesh FPGA Architecture using Novel Hierarchical Interconnect Topology and Long Touting Wires”. In: *Microprocessors and Microsystems* (Feb. 2016), pp. 16–26. DOI: `10.1016/j.micpro.2015.11.011`.

[131] S. Chtourou, M. Abid, Z. Marrakchi, E. Amouri, and H. Mehrez. “On Exploiting Partitioning-Based Placement Approach for Performances Improvement of 3D FPGA”. In: *International Conference on High Performance Computing Simulation (HPCS)*. IEEE, July 2017, pp. 572–579. DOI: `10.1109/HPCS.2017.91`.

[132] S.M. Mohtavipour and H.S. Shahhoseini. “A Link-Elimination Partitioning Approach for Application Graph Mapping in Reconfigurable Computing Systems”. In: *The Journal of Supercomputing* (Nov. 2019), pp. 726–754. DOI: `10.1007/s11227-019-03056-5`.

[133] S. M. Mohtavipour and H. Shahriar Shahhoseini. “A Low-Cost Distributed Mapping for Large-Scale Applications of Reconfigurable Computing Systems”. In: *International Computer Conference, Computer Society of Iran (CSICC)*. IEEE, Jan. 2020, pp. 1–6. DOI: `10.1109/CSICC49403.2020.9050063`.

[134] S. M. Mohtavipour and H. S. Shahhoseini. “A Large-Scale Application Mapping in Reconfigurable Hardware Using Deep Graph Convolutional Network”. In: *International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, Oct. 2020, pp. 382–387. DOI: `10.1109/ICCKE50421.2020.9303679`.

[135] A. Marquardt, V. Betz, and J. Rose. “Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, Feb. 1999, pp. 37–46. DOI: `10.1145/296399.296426`.

[136] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. “Resource Elastic Virtualization for FPGAs using OpenCL”. In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2018, pp. 1–8. DOI: `10.1109/FPL.2018.00028`.

[137] U. I. Minhas, R. Woods, and G. Karakonstantis. “Optimisation of System Throughput Exploiting Tasks Heterogeneity on Space Shared FPGAs”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2019, pp. 359–362. DOI: `10.1109/ICFPT47387.2019.00067`.

[138] G. A. Silva Novaes, L. C. Moreira, and W. J. Chau. "Exploring Tabu Search Based Algorithms for Mapping and Placement in NoC-based Reconfigurable Systems". In: *Symposium on Integrated Circuits and Systems Design (SBCCI)*. ACM, Aug. 2019, pp. 1–6. DOI: `10.1145/3338852.3339843`.

[139] M. Y. Rad and S. Shahbandegan. "An Intelligent Algorithm for Mapping of Applications on Parallel Reconfigurable Systems". In: *Iranian Conference on Signal Processing and Intelligent Systems (ICSPIS)*. IEEE, Dec. 2020, pp. 1–6. DOI: `10.1109/ICSPIS51611.2020.9349558`.

[140] J. G. Filho and W. J. Chau. "Exploring the Problems of Placement and Mapping in NoC-Based Reconfizurable Systems". In: *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, Dec. 2013, pp. 1–4. DOI: `10.1109/ReConFig.2013.6732289`.

[141] G. A. Silva Novaes, L. C. Moreira, and W. J. Chau. "Mapping and Placement in NoC-based Reconfigurable Systems Using an Adaptive Tabu Search Algorithm". In: *Latin American Symposium on Circuits Systems (LASCAS)*. IEEE, Feb. 2019, pp. 145–148. DOI: `10.1109/LASCAS.2019.8667553`.

[142] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen. "DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2005, pp. 153–158. DOI: `10.1109/FPL.2005.1515715`.

[143] J. Nickolls, I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?" In: *Queue* (Mar. 2008), pp. 40–53. DOI: `10.1145/1365490.1365500`.

[144] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science Engineering (CiSE)* (May 2010), pp. 66–73. DOI: `10.1109/MCSE.2010.69`.

[145] W. W. Hwu J. A. Stratton S. S. Stone. "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs". In: *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, July 2008, pp. 16–30. DOI: `10.1007/978-3-540-89740-8_2`.

[146] G. De Fabritiis M.J. Harvey. "wan: A Tool for Porting CUDA Programs to OpenCL". In: *Computer Physics Communications (CPC)* (Apr. 2011), pp. 1093–1099. DOI: `0.1016/j.cpc.2010.12.052`.

[147] I. Mavroidis, I. Mavroidis, I. Papaefstathiou, L. Lavagno, M. Lazarescu, E. de la Torre, and F. Schäfer. "FASTCUDA: Open Source FPGA Accelerator & Hardware-Software Codesign Toolset for CUDA Kernels". In: *Euromicro Conference on Digital System Design (DSD)*. IEEE, Sept. 2012, pp. 343–348. DOI: `10.1109/DSD.2012.58`.

[148] M. Knaust, F. Mayer, and T. Steinke. "OpenMP to FPGA Offloading Prototype Using OpenCL SDK". In: *International Workshop High-Level Parallel Programming Models and Supportive Environment (HIPS)*. IEEE, May 2019, pp. 387–390. DOI: `10.1109/IPDPSW.2019.00072`.

[149] S. Lee, J. Kim, and J. S. Vetter. "OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, July 2016, pp. 544–554. DOI: `10.1109/IPDPS.2016.28`.

[150] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. W. Hwu. “FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs”. In: *Symposium on Application Specific Processors (SASP)*. IEEE, July 2009, pp. 35–42. DOI: `10.1109/SASP.2009.5226333`.

[151] L. Sommer, J. Korinth, and A. Koch. “OpenMP Device Offloading to FPGA Accelerators”. In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, July 2017, pp. 201–205. DOI: `10.1109/ASAP.2017.7995280`.

[152] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and C. Cong. “AutoPilot: A Platform-Based ESL Synthesis Syste”. In: ed. by P. Coussy and A. Morawiec. Springer, 2008. Chap. High-Level Synthesis: From Algorithm to Digital Circuit, pp. 99–112. DOI: `10.1007/978-1-4020-8588-8_6`.

[153] Khronos OpenCL Working Group. *The OpenCL Specification*. Dec. 2020.

[154] A. Munshi. “The OpenCL Specification”. In: *Hot Chips 21 Symposium (HCS)*. IEEE, Aug. 2009, pp. 1–314. DOI: `10.1109/HOTCHIPS.2009.7478342`.

[155] S. Lee and J. S. Vetter. “OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing”. In: *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, June 2014, pp. 115–120. DOI: `10.1145/2600212.2600704`.

[156] L. Dagum and R. Menon. “OpenMP: An Industry Standard API for Shared-Memory Programming”. In: *Computational Science and Engineering*. IEEE, Mar. 1998, pp. 46–55. DOI: `10.1109/99.660313`.

[157] OpenACC-Standard.org. *The OpenACC Application Programming Interface: Version 2.7*. Sept. 2018.

[158] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller. “A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing”. In: *International European Conference on Parallel Processing (Euro-Par)*. Springer, Aug. 2014, pp. 812–823. DOI: `10.1007/978-3-319-09873-9_68`.

[159] F. Mayer, M. Knaust, and M. Philippsen. “OpenMP on FPGAs—A Survey”. In: *OpenMP: Conquering the Full Hardware Spectrum*. Springer, Aug. 2019, pp. 94–108. DOI: `10.1007/978-3-030-28596-8_7`.

[160] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. “Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures”. In: *Parallel Processing Letters* (Mar. 2011), pp. 173–193. DOI: `10.1142/S0129626411000151`.

[161] J. M. Perez, R. M. Badia, and J. Labarta. “A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures”. In: *International Conference on Cluster Computing (CLUSTER)*. IEEE, Oct. 2008, pp. 142–151. DOI: `10.1109/CLUSTR.2008.4663765`.

[162] J. Bosch, A. Filgueras, M. Vidal, D. Jimenez-Gonzalez, C. Alvarez, and X. Martorell. “Exploiting Parallelism on GPUs and FPGAs with OmpSs”. In: *Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems (ANDARE)*. ACM, Sept. 2017, pp. 1–5. DOI: `10.1145/3152821.3152880`.

[163] Khronos SYCL Working Group. *The SYCL Specification*. Apr. 2020.

[164] H. C. da Silva, F. Pisani, and E. Borin. “A Comparative Study of SYCL, OpenCL, and OpenMP”. In: *International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, Oct. 2016, pp. 61–66. DOI: `10.1109/SBAC-PADW.2016.19`.

[165] Khronos SYCL Working Group. *SYCL Implementations*. Jan. 2021. URL: `www.khronos.org/sycl/`.

[166] S. Memeti, L. Li, S. Pllana, J. Koodziej, and C. Kessler. “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption”. In: *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC)*. ACM, July 2017, pp. 1–6. DOI: `10.1145/3110355.3110356`.

[167] Khronos SYCL Working Group. *SYCL Projects*. Jan. 2021. URL: `www.sycl.tech/projects/`.

[168] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. “Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries”. In: *Journal on Scientific Computing (SISC)* (Sept. 2013), pp. 453–472. DOI: `10.1137/120903683`.

[169] N. Bell and J. Hoberock. “Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA”. In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 359–371. DOI: `10.1016/B978-0-12-385963-1.00026-5`.

[170] P. Gottschling and T. Hoefler. “Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption”. In: *nternational Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE/ACM, May 2012, pp. 9–16. DOI: `10.1109/CCGrid.2012.51`.

[171] D. Demidov. *VexCL Documentation*. May 2017. URL: `https://vexcl.read%5C-the%5C-docs.io/`.

[172] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr. “ViennaCL—Linear Algebra Library for Multi- and Many-Core Architectures”. In: *Journal on Scientific Computing (SISC)* (Oct. 2016), pp. 412–439. DOI: `10.1137/15M1026419`.

[173] V. Shkarupa, R. Mencis, and M. Sabatelli. “Offline Handwriting Recognition Using LSTM Recurrent Neural Networks”. In: *Benelux Conference on Artificial Intelligence*. Research Gate, Nov. 2016, pp. 1–9.

[174] M. Kowalczyk, D. Przewlocka, and T. Krvjak. “Real-Time Implementation of Contextual Image Processing Operations for 4K Video Stream in Zynq UltraScale+ MPSoC”. In: *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Oct. 2018, pp. 37–42. DOI: `10.1109/DASIP.2018.8597105`.

[175] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. Nov. 2016, pp. 265–283. DOI: `arXiv:1605.08695`.

[176] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *ACM International Conference on Multimedia (MM)*. ACM, Nov. 2014, pp. 675–678. DOI: `10.1145/2647868.2654889`.

[177] S. Shi, Q. Wang, P. Xu, and X. Chu. "Benchmarking State-of-the-Art Deep Learning Software Tools". In: *International Conference on Cloud Computing and Big Data (CCBD)*. IEEE, Sept. 2016, pp. 99–104. DOI: `10.1109/CCBD.2016.029`.

[178] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., Jan. 2008.

[179] OpenCV. *OpenCV OpenCL Acceleration*. 2021. URL: `https://opencv.org/opencl`.

[180] R. Haase, L.A. Royer, P. Steinbach, D. Schmidt, A. Dibrov, U. Schmidt, M. Weigert, N. Maghelli, P. Tomancak, F. Jug, and E.W. Myers. "CLIJ: GPU-Accelerated Image Processing for Everyone". In: *Nature Methods* (Jan. 2020), pp. 1–2. DOI: `10.1038/s41592-019-0650-1`.

[181] Xilinx. *Xilinx xfOpenCV Library*. June 2019.

[182] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. "A Highly Efficient and Comprehensive Image Processing Library for C++-based High-Level Synthesis". In: *International Workshop on FPGAs for Software Programmers (FSP)*. VDE, Sept. 2017, pp. 1–10.

[183] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow. "OpenCL Library of Stream Memory Components Targeting FPGAs". In: *International Conference on Field Programmable Technology (FPT)*. IEEE, Dec. 2015, pp. 104–111. DOI: `10.1109/FPT.2015.7393134`.

[184] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. "Programming Heterogeneous Systems from an Image Processing DSL". In: *Transactions on Architecture and Code Optimization (TACO)* (Aug. 2017), pp. 1–25. DOI: `10.1145/3107953`.

[185] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Special Interest Group on Programming Languages (SIGPLAN)* (June 2013), pp. 519–530. DOI: `10.1145/2499370.2462176`.

[186] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. "HIPAcc: A Domain-Specific Language and Compiler for Image Processing". In: *Transactions on Parallel and Distributed Systems (TPDS)* (Jan. 2016), pp. 210–224. DOI: `10.1109/TPDS.2015.2394802`.

[187] M. A. Özkan, O. Reiche, F. Hannig, and J. Teich. "FPGA-based Accelerator Design from a Domain-Specific Language". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2016, pp. 1–9. DOI: `10.1109/FPL.2016.7577357`.

[188] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. "Code Generation from a Domain-Specific Language for C-based HLS of Hardware Accelerators". In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM, Oct. 2014, pp. 1–10. DOI: `10.1145/2656075.2656081`.

[189] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. "A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs". In: *International Conference on Parallel Architectures and Compilation (PACT)*. ACM, Sept. 2016, pp. 327–338. DOI: `10.1145/2967938.2967969`.

[190] R. T. Mullapudi, V. Vasista, and U. Bondhugula. "Polymage: Automatic Optimization for Image Processing Pipelines". In: *Special Interest Group on Computer Architecture (SIGARCH)* (Mar. 2015), pp. 429–443. DOI: `10.1145/2786763.2694364`.

[191] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. “Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines”. In: *Transactions on Graphics (TOG)* (July 2014), pp. 1–11. DOI: `10.1145/2601097.2601174`.

[192] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. “Rigel: Flexible Multi-Rate Image Processing Hardware”. In: *Transactions on Graphics (TOG)* (July 2016), pp. 1–11. DOI: `10.1145/2897824.2925892`.

[193] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar. 2021, pp. 2–14. DOI: `10.1109/CGO51591.2021.9370308`.

[194] J. Kessenich, B. Ouriel, and R. Krisch. *SPIR-V Specification*. Jan. 2021. URL: `www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf`.

[195] S. Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. Polly Labs and KU Leuven, Jan. 2016. DOI: `10.13140/RG.2.1.1174.6323`.

[196] S. Verdoolaege. “ISL: An Integer Set Library for the Polyhedral Model”. In: *International Congress on Mathematical Software (ICMS)*. Springer, Sept. 2010, pp. 299–302. DOI: `10.1007/978-3-642-15582-6_49`.

[197] A. Darte, Y. Robert, and F. Vivien. “Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems”. In: ed. by S. Pande and D. P. Agrawal. Springer, May 2001. Chap. Loop Parallelization Algorithms, pp. 141–171. DOI: `10.1007/3-540-45403-9_5`.

[198] M. Griebl and C. Lengauer. “The Loop Parallelizer LooPo”. In: *Workshop on Compilers for Parallel Computers (CPC)*. Feb. 1997, pp. 311–320. DOI: `10.1007/BFb0017283`.

[199] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. “A Practical Automatic Polyhedral Parallelizer and Locality Optimizer”. In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 2008, pp. 101–113. DOI: `10.1145/1375581.1375595`.

[200] P. Feautrier. “Parametric Integer Programming”. In: *RAIRO-Operations Research* (Aug. 1988), pp. 243–268. DOI: `10.1051/ro/1988220302431`.

[201] C. Bastoul. “Code Generation in the Polyhedral Model is Easier than you Think”. In: *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, Oct. 2004, pp. 7–16. DOI: `10.1109/PACT.2004.1342537`.

[202] S. Verdoolaege and T. Grosser. “Polyhedral Extraction Tool”. In: *Workshop on Polyhedral Compilation Techniques (IMPACT)*. Jan. 2012, pp. 1–8. DOI: `10.13140/RG.2.1.4213.4562`.

[203] C. Lengauer T. Grosser A. Groesslinger. “Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* (Dec. 2012), pp. 1–27. DOI: `10.1142/S0129626412500107`.

[204] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. “GRAPHITE: Polyhedral Analyses and Optimizations for GCC”. In: *Proceedings of the GCC Developers’ Summit*. June 2006, pp. 179–197.

[205] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. “Automatic C-to-CUDA Code Generation for Affine Programs”. In: *International Conference on Compiler Construction (CC)*. Springer, Mar. 2010, pp. 244–263. DOI: `10.1007/978-3-642-11970-5_14`.

[206] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. “Polyhedral Parallel Code Generation for CUDA”. In: *Transactions on Architecture and Code Optimization (TACO)* (Jan. 2013), pp. 1–23. DOI: `10.1145/2400682.2400713`.

[207] S. Verdoolaege and A. Cohen. “Live Range Reordering”. In: *International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Polly Labs and KU Leuven, Jan. 2016. DOI: `10.13140/RG.2.1.3272.9680`.

[208] W. Klingauf and R. Gunzel. “From TLM to FPGA: Rapid Prototyping with SystemC and Transaction Level Modeling”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2005, pp. 285–286. DOI: `10.1109/FPT.2005.1568563`.

[209] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi. “Chisel: Constructing hardware in a Scala embedded language”. In: *Design Automation Conference (DAC)*. June 2012, pp. 1212–1221. DOI: `10.1145/2228360.2228584`.

[210] Intel. *Intel FPGA SDK for OpenCL ProEdition*. Dec. 2020.

[211] I. Janik, Q. Tang, and M. Khalid. “An Ooverview of Altera SDK for OpenCL: A User Perspective”. In: *Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, May 2015, pp. 559–564. DOI: `10.1109/CCECE.2015.7129336`.

[212] Xilinx. *Vivado Design Suite UserGuide - High-Level Synthesis*. July 2019.

[213] Xilinx. *SDSoC Environment User Guide*. May 2019.

[214] Xilinx. *SDAccel Environment UserGuide*. May 2019.

[215] K. Hill, S. Craciun, A. George, and H. Lam. “Comparative Analysis of OpenCL vs. HDL with Image-Processing Kernels on Stratix-V FPGA”. In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, July 2015, pp. 189–193. DOI: `10.1109/ASAP.2015.7245733`.

[216] J. Villarreal, A. Park, W. Najjar, and R. Halstead. “Designing Modular Hardware Accelerators in C with ROCCC 2.0”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ACM, May 2010, pp. 127–134. DOI: `10.1109/FCCM.2010.28`.

[217] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. “LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems”. In: *International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, Feb. 2011, pp. 33–36. DOI: `10.1145/1950413.1950423`.

[218] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, and J. Anderson. “LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems”. In: *Transactions on Embedded Computing Systems (TECS)* (Sept. 2013), pp. 1–27. DOI: `10.1145/2514740`.

[219] A. Canis, J. Choi, R. L. Lian, and J. Anderson. *LegUp: Benefits of High-Level Synthesis for FPGA Design*. Sept. 2020. URL: `www.legupcomputing.com`.

[220] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. "High-Level Language Tools for Reconfigurable Computing". In: *Proceedings of the IEEE* (Apr. 2015), pp. 390–408. DOI: `10.1109/JPROC.2015.2399275`.

[221] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications". In: *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, June 2004, pp. 69–70. DOI: `10.1109/MEMCOD.2004.1459818`.

[222] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. "A Survey and Evaluation of FPGA High-Level Synthesis Tool". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Oct. 2016), pp. 1591–1604. DOI: `10.1109/TCAD.2015.2513673`.

[223] C. Pilato and F. Ferrandi. "Bambu: A modular Framework for the High Level Synthesis of Memory-Intensive Applications". In: *International Conference on Field programmable Logic and Applications (FPL)*. IEEE, Sept. 2013, pp. 1–4. DOI: `10.1109/FPL.2013.6645550`.

[224] R. Nane, V. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. "DWARV 2.0: A CoSy-based C-to-VHDL Hardware Compiler". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012, pp. 619–622. DOI: `10.1109/FPL.2012.6339221`.

[225] F. Winterstein, S. Bayliss, and G. A. Constantinides. "High-Level Synthesis of Dynamic Data Structures: A Case Study using Vivado HLS". In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2013, pp. 362–365. DOI: `10.1109/FPT.2013.6718388`.

[226] M. Hosseinabady and J. L. Nunez-Yanez. "Optimised OpenCL Workgroup Synthesis for Hybrid ARM-FPGA Devices". In: *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Sept. 2015, pp. 1–6. DOI: `10.1109/FPL.2015.7294016`.

[227] Z. Wang, B. He, W. Zhang, and S. Jiang. "A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs". In: *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Apr. 2016, pp. 114–125. DOI: `10.1109/HPCA.2016.7446058`.

[228] S. O. Ayat, M. Khalil-Hani, and R. Bakhteri. "OpenCL-Based Hardware-Software Co-Design Methodology for Image Processing Implementation on Heterogeneous FPGA Platform". In: *International Conference on Control System, Computing and Engineering (ICCSCE)*. IEEE, Sept. 2015, pp. 36–41. DOI: `10.1109/ICCSCE.2015.7482154`.

[229] Khronos OpenVX Working Group. *The OpenVX Specification (version 1.3)*. Sept. 2020.

[230] J. Hascoë, B. D. de Dinechin, K. Desnos, and J. Nezan. "A Distributed Framework for Low-Latency OpenVX over the RDMA NoC of a Clustered Manycore". In: *High Performance extreme Computing Conference (HPEC)*. IEEE, Sept. 2018, pp. 1–7. DOI: `10.1109/HPEC.2018.8547736`.

[231] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. "ADRENALINE: An OpenVX Environment to Optimize Embedded Vision Applications on Many-core Accelerators". In: *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, Sept. 2015, pp. 289–296. DOI: `10.1109/MCSoC.2015.45`.

[232] H. Omidian, N. Ivanov, and G. G. F. Lemieux. "An Accelerated OpenVX Overlay for Pure Software Programmers". In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2018, pp. 290–293. DOI: `10.1109/FPT.2018.00056`.

[233] H. Omidian, N. Ivanov, and G. G. F. Lemieux. “An Accelerated OpenVX Overlay for Pure Software Programmers”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, Dec. 2018, pp. 290–293. DOI: `10.1109/FPT.2018.00056`.

[234] S. Taheri, J. Heo, P. Behnam, J. Chen, A. Veidenbaum, and A. Nicolau. “Acceleration Framework for FPGA Implementation of OpenVX Graph Pipelines”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 2018, pp. 227–227. DOI: `10.1109/FCCM.2018.00061`.

[235] M. A. Özkan, B. Ok, B. Qiao, O. Reiche, J. Teich, and F. Hannig. “HipaccVX: Wedding of OpenVX and DSL-based Code Generation”. In: *Journal of Real-Time Image Processing (JRTIP)* (June 2021), pp. 1861–8219. DOI: `10.1007/s11554-020-01015-5`.

[236] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. “A Framework for Optimizing OpenVX Applications Performance on Embedded Manycore Accelerators”. In: *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, June 2015, pp. 125–128. DOI: `10.1145/2764967.2776858`.

[237] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. “Optimizing Memory Bandwidth Exploitation for OpenVX Applications on Embedded Many-Core Accelerators”. In: *Journal of Real-Time Image Processing (JRTIP)* (June 2018), pp. 73–92. DOI: `10.1007/s11554-015-0544-0`.

[238] H. Omidian and G. G. F. Lemieux. “Exploring automated space/time tradeoffs for OpenVX compute graphs”. In: *International Conference on Field Programmable Technology (FPT)*. IEEE, Dec. 2017, pp. 152–159. DOI: `10.1109/FPT.2017.8280133`.

[239] S. Taheri, P. Behnam, E. Bozorgzadeh, A. Veidenbaum, and A. Nicolau. “AFFIX: Automatic Acceleration Framework for FPGA Implementation of OpenVX Vision Algorithms”. In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, Feb. 2019, pp. 252–261. DOI: `10.1145/3289602.3293907`.

[240] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. “SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters”. In: *International Conference on Supercomputing (ICS)*. ACM, June 2012, pp. 341–352. DOI: `10.1145/2304576.2304623`.

[241] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee. “A Distributed OpenCL Framework Using Redundant Computation and Data Replication”. In: *Conference on Programming Language Design and Implementation (SIGPLAN)*. ACM, June 2016, pp. 553–569. DOI: `10.1145/2908080.2908094`.

[242] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Concurrency and Computation: Practice and Experience (CCPE)* (Feb. 2011), pp. 187–198. DOI: `10.1002/cpe.1631`.

[243] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. “StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators”. In: *Recent Advances in the Message Passing Interface (EuroMPI)*. Springer, Sept. 2012, pp. 298–299. DOI: `10.1007/978-3-642-33518-1_40`.

[244] J. Kim, T. T. Dao, J. Jung, J. Joo, and J. Lee. “Bridging OpenCL and CUDA: A Comparative Analysis and Translation”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, Nov. 2015, pp. 1–12. DOI: `10.1145/2807591.2807621`.

[245] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. “Locality and Loop Scheduling on NUMA Multiprocessors”. In: *International Conference on Parallel Processing (ICPP)*. IEEE, Aug. 1993, pp. 140–147. DOI: `10.1109/ICPP.1993.112`.

[246] L. Kalms, A. Podlubne, and D. Göhringer. *HiFlipVX*. Aug. 2020. URL: `https://github.com/TUD-ADS/HiFlipVX`.

[247] F. d. Dinechin and M. Istoan. “Hardware Implementations of Fixed-Point Atan2”. In: *22nd Symposium on Computer Arithmetic (ARITH)*. IEEE, June 2015, pp. 34–41. DOI: `10.1109/ARITH.2015.23`.

[248] J. M. Palomares, J. Gonzalez, E. Ros, and A. Prieto. “General Logarithmic Image Processing Convolution”. In: *Transactions on Image Processing* (Nov. 2006), pp. 3602–3608. DOI: `10.1109/TIP.2006.881967`.

[249] A. Hematian, S. Chuprat, A. A. Manaf, and N. Parsazadeh. “Zero-delay FPGA-based odd-even sorting network”. In: *Symposium on Computers Informatics (ISCI)*. IEEE, Apr. 2013, pp. 128–131. DOI: `10.1109/ISCI.2013.6612389`.

[250] D. E.Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., May 1998.

[251] L. A. Aranda, P. Reviriego, and J. A. Maestro. “A fault-tolerant implementation of the median filter”. In: *16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*. IEEE, Sept. 2016, pp. 1–4. DOI: `10.1109/RADECS.2016.8093153`.

[252] J. Han, S. Yang, and B. Lee. “A Novel 3-D Color Histogram Equalization Method With Uniform 1-D Gray Scale Histogram”. In: *Transactions on Image Processing* (Feb. 2011), pp. 506–512. DOI: `10.1109/TIP.2010.2068555`.

[253] An Analysis of the SURF Method. “E. Oyallon and R. Julien”. In: *Image Processing On Line* (July 2015), pp. 176–218. DOI: `10.5201/ipol`.

[254] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *32nd International Conference on Machine Learning*. PMLR, July 2015, pp. 448–456.

[255] P. F. Alcantarilla. *A-KAZE Features*. Oct. 2016. URL: `https://github.com/pablofdezalc/akaze`.

[256] A. Alahi. *FREAK: Fast Retina Keypoint*. Sept. 2014. URL: `https://github.com/kikohs/freak`.

[257] K. Mikolajczyk and C. Schmid. “A Performance Evaluation of Local Descriptors”. In: *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (Aug. 2005), pp. 1615–1630. DOI: `10.1109/TPAMI.2005.188`.

[258] G. Akgün, L. Kalms, and D. Göhringer. “Resource Efficient Dynamic Voltage and Frequency Scaling on Xilinx FPGAs”. In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, Apr. 2020, pp. 178–192. DOI: `10.1007/978-3-030-44534-8_14`.

[259] E. Mair, G.D. Hager, D. Burschka, M. Suppa, and G. Hirzinger. “Adaptive and Generic Corner Detection Based on the Accelerated Segment Test”. In: *European Conference on Computer Vision (ECCV)*. Springer, Sept. 2010, pp. 183–196. DOI: `10.1007/978-3-642-15552-9_14`.

[260] M. Göbel, C.C. Chi, M. Alvarez-Mesa, and B. Juurlink. “High Performance Memory Accesses on FPGA-SoCs: A Quantitative Analysis”. In: *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, May 2015, pp. 32–32. DOI: `10.1109/FCCM.2015.23`.

[261] P. Soleimani, D.W. Capson, and K.F. Li. “Real-time FPGA-based implementation of the AKAZE algorithm with nonlinear scale space generation using image partitioning”. In: *Journal of Real-Time Image Processing (JRTIP)* (Mar. 2021), pp. 2123–2134. DOI: `10.1007/s11554-021-01089-9`.

[262] S. Du, Y. LI, and T. Ikenaga. “Temporally Forward Nonlinear Scale Space for High Frame Rate and Ultra-Low Delay A-KAZE Matching System”. In: *Transactions on Information and Systems (IEICE)* (June 2020), pp. 1226–1235. DOI: `10.1587/transinf.2019MVP0019`.

[263] L. Kalms, A. Elhossini, and B. Juurlink. “FPGA Based Hardware Accelerator for KAZE Feature Extraction Algorithm”. In: *International Conference on Field-Programmable Technology (FPT)*. IEEE, July 2016, pp. 281–284. DOI: `10.1109/FPT.2016.7929553`.

[264] R. O. Hassan. “Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC”. In: *Analog Integrated Circuits and Signal Processing* (Feb. 2021), pp. 399–408. DOI: `10.1007/s10470-020-01638-5`.

[265] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li. “An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution”. In: *Electronics* (Mar. 2019), pp. 1–18. DOI: `10.3390/electronics8030281`.

[266] Tensorflow. *SSD MobileNet v1*. May 2021. URL: `https://tensorflow.org/lite/models/object_detection/overview`.

[267] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. arXiv: `2004.09602 [cs.LG]`.

[268] A. Sadek, A. Muddukrishna, L. Kalms, A. Djupdal, A. Podlubne, A. Paolillo, D. Göhringer, and M. Jahre. “Supporting utilities for heterogeneous embedded image processing platforms (sthem)): An overview”. In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, May 2018, pp. 737–749. DOI: `10.1007/978-3-319-78890-6_59`.

[269] A. Podlubne, J. Haase, L. Kalms, G. Akgün, M. Ali, H.u.H. Khan, A. Kamal, and D. Göhringer. “Low power image processing applications on FPGAs using dynamic voltage scaling and partial reconfiguration”. In: *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Oct. 2018, pp. 64–69. DOI: `10.1109/DASIP.2018.8596910`.

[270] Intel. *Optimizing Simple OpenCL Kernels: Sobel Kernel Optimization*. 2014.

[271] O. Bachmann, P. S. Wang, and E. V. Zima. “Chains of Recurrencesa Method to Expedite the Evaluation of Closed-Form Functions”. In: *International Symposium on Symbolic and Algebraic Computation (ISSAC)*. ACM, Aug. 1994, pp. 242–249. DOI: `10.1145/190347.190423`.

[272] Tim Häring. *AMDOVX OpenCL Kernel Extraction for High-Performance Vision Toolchain*. May 2021. URL: `https://github.com/timhae/amdovx-core`.

[273] cameron314. *Reader Writer Queue*. Feb. 2021. URL: `https://github.com/cameron314/readerwriterqueue`.

[274] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. June 2019. URL: `https://riscv.org/`.

[275] Xilinx. *UG984: MicroBlaze Processor Reference Guide (v2019.2)*. Oct. 2019. URL: `https://www.xilinx.com/support.html`.

Student Work

[STD01] Daniel Neudek. "Implementierung eines Computer Vision Algorithmus (FAST-Detektor und FREAK-Deskriptor) auf einem heterogenen System in OpenCL". Bachelor Thesis. July 2016, pp. 7–54.

[STD02] Marvin Diedrich. "Implementierung des AKAZE Feature Detektors auf einem heterogenen System in OpenCL zur Objekterkennung". Bachelor Thesis. July 2016, pp. 4–65.

[STD03] Hasan Ibrahim. "Accelerated Embedded Feature Detection Using ORB Algorithm". Bachelor Thesis. Aug. 2016, pp. 6–70.

[STD04] Holger Hantusch. "Implementierung und Parallelisierung des Voila-Jones-Algorithmus für Objekterkennung in OpenCL auf einem heterogenen System". Bachelor Thesis. Aug. 2016, pp. 1–44.

[STD05] Khaled Mohamed. "Accelerated Embedded Feature Detection using the AKAZE Algorithm". Bachelor Thesis. Aug. 2016, pp. 7–67.

[STD06] Marc Hamme. "Robuste Spurerkennung für autonomes Fahren auf einem eingebetteten System". Bachelor Thesis. Feb. 2017, pp. 9–80.

[STD07] Henry Bathke. "GPU-FPGA Kommunikation über PCIe". Seminar. June 2017, pp. 1–10.

[STD08] Patrick Kappen. "OpenCL Deap Learning Framework for Automated Text Recognition". Master Thesis. June 2017, pp. 9–115.

[STD09] Tim Hebbeler. "Automatische Erstellung von OpenCL-Code für GPUs aus LLVM-IR mit Hilfe der polyhedralen Optimierung". Masters Thesis. July 2017, pp. 1–87.

[STD10] Sarah Milad. "Design and Implementation of High performance Gesture Recognition using deep learning". Master Thesis. Apr. 2018, pp. 1–67.

[STD11] Maximilian Hajduk. "Entwicklung eines Speichercontrollers auf einem Xilinx SoC für den FREAK Algorithmus". Grosser Beleg. Nov. 2018, pp. 1–62.

[STD12] Mirko Schäfer. "An Evaluation of SYCL". Seminar. Aug. 2019, pp. 1–13.

[STD13] Arsany Eskander. "HW/SW Co-design For Object Detection Using a Machine Learning Accelerator". Bachelor Thesis. Aug. 2019, pp. 1–74.

[STD14] Matthias Nickel. "Implementation of the graph based OpenVX approach for the automated application distribution of HiFlipVX". Project. Aug. 2019, pp. 3–11.

[STD15] Matthias Nickel. "Entwicklung von High-Level Synthese Komponenten zur Integration von HiFlipVX in eine NoC-Topologie". Project. Aug. 2019, pp. 1–7.

[STD16] Karl Friebel. “Extending A Source-to-source Compiler For Use In A Hls Toolchain”. Project. Aug. 2019, pp. 1–11.

[STD17] Tim Häring. “Computer Vision Algorithm Design using High-level Synthesis”. Project. Sept. 2020, pp. 1–19.

[STD18] Karl Friedrich Alexander Friebel. “Sycl to C99 Based Transpiler for Generating Data Flow Oriented FPGA Accelerators”. Diploma Thesis. Dec. 2020, pp. 1–60.

[STD19] Matthias Nickel. “Implementation and Optimization of the AKAZE Feature Detection Algorithm for the HiFlipVX Library using High-Level Synthesis”. Masters Thesis. May 2021, pp. 1–79.

[STD20] Xinyue Shi. “DMA Controller and Library Implementation and Integration for NoC-based RISC-V Systems”. Project. May 2021, pp. 1–19.

[STD21] Tim Häring. “A Framework to Schedule OpenVX-based Applications on x86-based Systems Containing CPUs, GPUs and FPGAs”. Masters Thesis. June 2021, pp. 1–68.

[STD22] Osama Ibrahim. “HW/SW Co-Design of a Feature Matching Algorithm for an ARM-FPGA System”. Project. Aug. 2021, pp. 5–26.

[STD23] Tianyu Xing. “Building a Feature Extraction Algorithm for a Video System on an FPGA-ARM SoC”. Project. Aug. 2021, pp. 5–20.

A Appendix

A.1 High-Performance Vision toolchain: Transaction Patterns Between Kernels

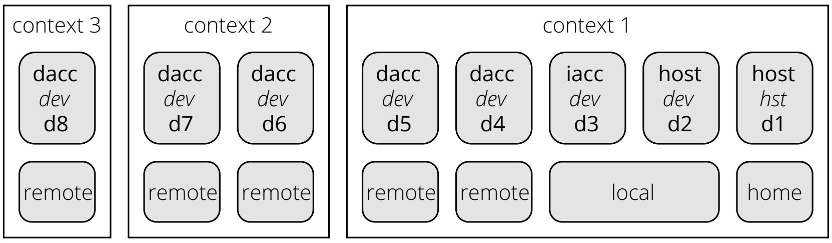


Figure A.1: Example system to cover all possible types of data transfers.

This section gives a more detailed description of the transfer patterns in Section 4.5.5. Figure A.1 shows an example system consisting of three contexts to represent all different data transfer possibilities within one compute node. In this case, $d1$ – $d2$ can be a CPU, which can be used as a host or as a device, $d3$ could be an integrated GPU, and $d4$ – $d8$ could be dedicated GPUs and FPGAs. While $d1$ computes on the home node, $d2$ – $d3$ compute on the local node. All other devices have their own remote node. Each node represents a memory object in the same buffer. Memory objects are only created if they are really accessed.

For each device in Figure A.1, Table A.1 shows which is the closest device that will respond to a `bus-read` or `bus-update` action. For simplicity, the table does not show the calculated distances (latencies), but the order in which the source device is closest to the target device. If the variable has a higher number, the source node is either further away or has at least the same distance. Some variables, like $C1$ and $C2$ are repeated in the columns, because the data transfers have exactly the same sequence of commands.

In the last step of `compute_transaction()` function from Listing 4.3, all commands needed for the transaction are generated. Figure A.2 and A.3 show all transfer possibilities for the example system from Figure A.1. The variable names on the left side of the figure are the same as those used in Table A.1. Additionally, all transfer possibilities for a system that does not have a local node are shown ($E1$, $E2$ or $E3$). For this case the devices $d1$ – $d3$ are omitted.

The left side of Figure A.2 and A.3 shows the state of the various devices before the transfer starts. The destination of the transfer is the device which is enclosed with a thick border. The source of the transfer is the device with the dark gray background. A device can be source if a kernel, which has read or written the data, has been executed on one of its command

Table A.1: The labels in the table refer to Figure A.1 and represent the required data transfer from source device (1. column) to destination device (1. row). If the variable inside a column has a higher number, the source node is either further away or has the same distance.

src\dst	d1	d2	d3	d4	d5	d6	d7	d8
d1	A1	B3	B3	C4	C4	D3	D3	D3
d2	A2	B1	B2	C3	C3	D4	D4	D4
d3	A2	B2	B1	C3	C3	D4	D4	D4
d4	A3	B4	B4	C1	C2	D5	D5	D5
d5	A3	B4	B4	C2	C1	D5	D5	D5
d6	A4	B5	B5	C5	C5	C1	C2	D6
d7	A4	B5	B5	C5	C5	C2	C1	D6
d8	A4	B5	B5	C5	C5	D6	D6	C1

queues. All devices that could also have a copy of the data have a light gray background color. All devices without a background color cannot have a copy of the data, since these devices are closer to the destination than the source device. With the transfer sequences from Figure A.2 and A.3, all other transfer device pairs can also be represented, as shown in Table A.1.

On the right side of the figure the enqueue commands and the command queues where these commands are executed are shown. Commands which are executed only under certain conditions are shown in light gray. For synchronization of data movement and kernels there is a series of events ($e1$ – $e5$). Inter context events are represented by solid lines ($e1$) and intra context events by dashed lines ($e2$). The barriers in the device and host queues wait until all input events of all transfers have been generated so that the kernel can be executed ($e5$). In case of a `bus-update`, the system additionally waits until all readers of the buffer are finished. After a `bus-read`, commands that read can then read from these events ($e3$). If the data is copied via another node during the transfer, additional synchronization points are generated from which data can be read ($e4$). A closer look at these points reveals that they occur either on the local or on the home node, which is used for further simplification.

There is a set of conditions ($c1$ – $c6$) for the execution of several commands. For a `bus-update`, the barrier waits until all readers have finished reading or the owner has finished writing ($c3$). Only then data can be written to the memory object. Memory objects that have been `mapped` for reading into the host space can be read from the local node without an `unmap` command. However, an `unmap` is necessary to read from the local node if the data was `mapped` for writing ($c2$). Conversely, it would also not be possible to copy data from a remote node to a local node if it was `mapped` to host space for writing. However, this case will not occur because the home node is closer to the local node as compared to the remote nodes. For a `bus-update`, an `unmap` is also needed if the data is `mapped` for reading before it can be written to the buffer ($c1$). If the host's `map` flags change, the `unmap` and `map` commands are needed to change these flags ($c4$). For example, after the flags have been changed from write to read, the local node could read from the same memory object in parallel without an `unmap`. It can happen that data was written to the home before the application was executed. In these cases there

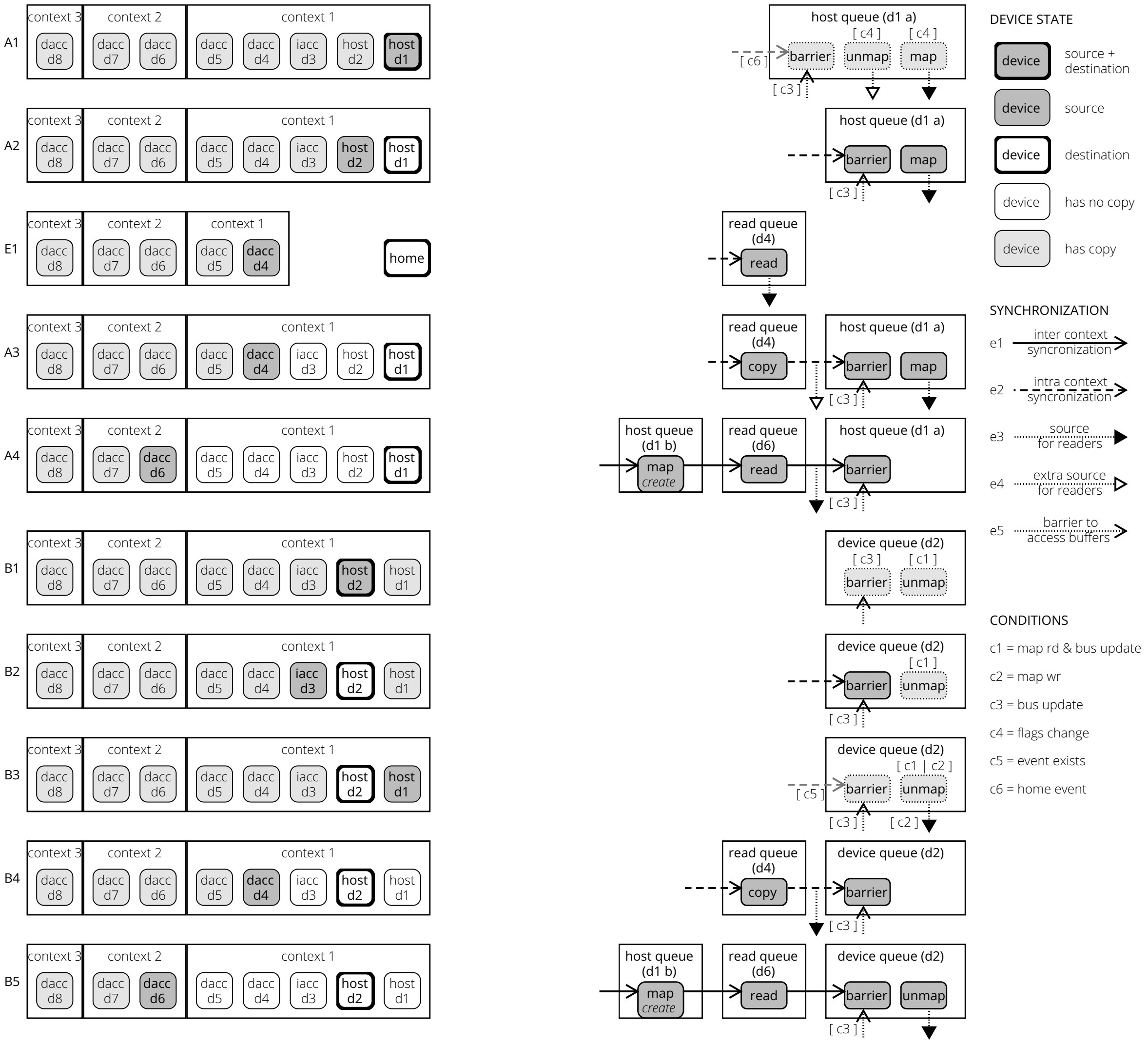


Figure A.2: All possible data transfers between memory objects of a buffer with a host or IACC as destination for the example system in Figure A.1.

is no input event for synchronization (*c*5). For *A*1, it makes a difference whether the source is the device itself or whether it is an event that occurred on the home node by copying data to it (*c*6).

For all transfers that go via the host, without it being the destination, the `map`/`unmap` commands are executed on an extra command queue (*d*1*b*). This queue is necessary so that on the one hand the host queue is not blocked and on the other hand the `map`/`unmap` commands are not blocked. If a DACC from context two or three triggers a `bus-update`, the `unmap` is also executed on this extra command queue (*d*1*b*), due to similar reasons. For a read command that writes to the host space, the host space must be `mapped` for writing (*A*4, *B*5, *C*5 or *D*6). For a write command that reads from host space to a DACC, the memory object must at least be `mapped` for reading (*D*4 or *D*5).

In a few cases, the data may have been `mapped` to host space for writing even though other devices still have a copy of the data (*A*4 or *D*6). If the data is `mapped` for writing, no device on the local node can have a current copy of the data (*A*2, *A*3 or *A*4). The host is not selected as

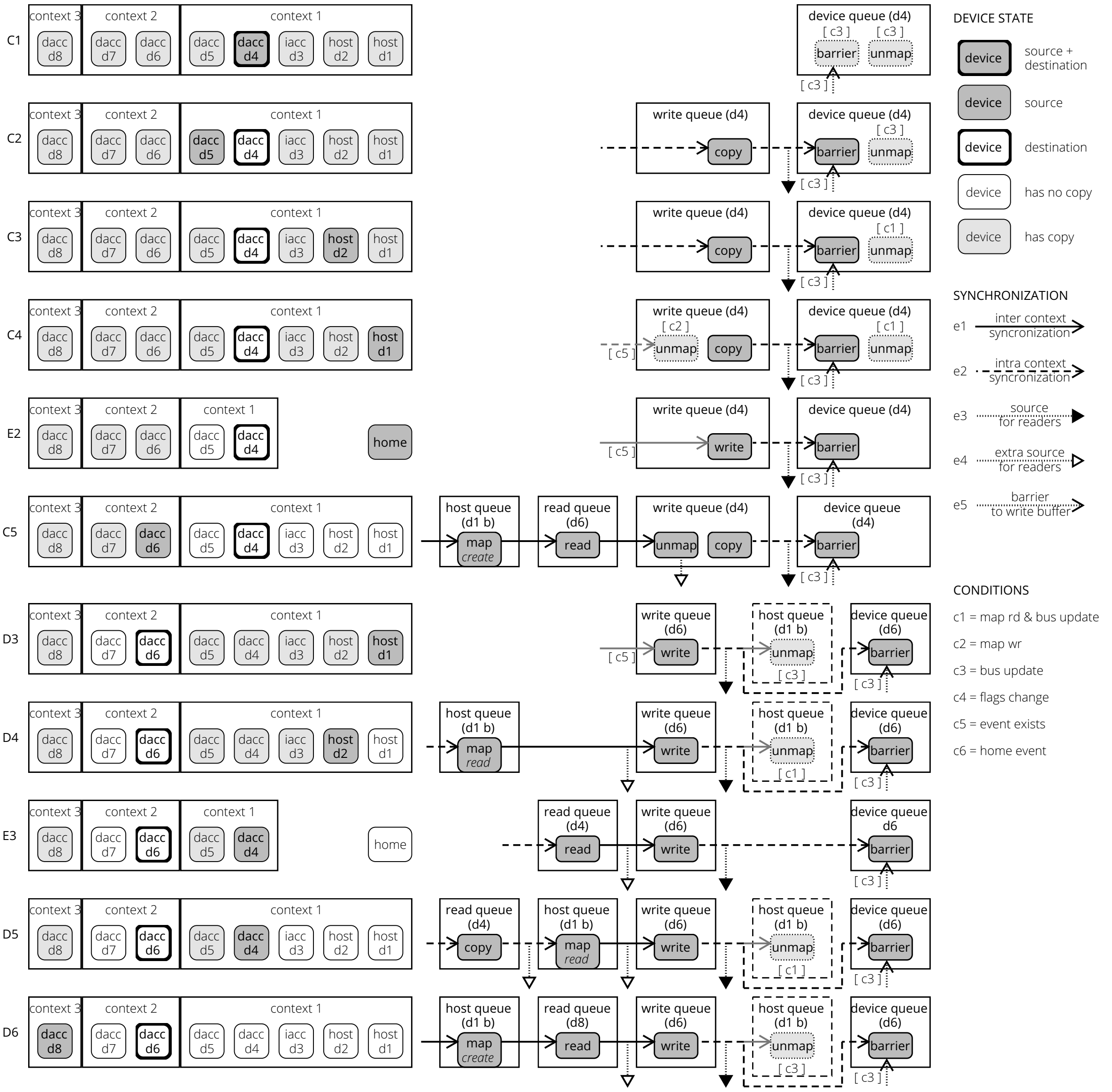


Figure A.3: All possible data transfers between memory objects of a buffer with a DACC as destination for the example system in Figure A.1.

the source if the data is not `mapped`, but it can still be a reader or the owner (*A*1, *B*3, *C*4 or *D*3). However, in these cases at least one device of the local node has the data and is selected as source. In some cases it is therefore not possible for the data to be `mapped` for writing (*B*1, *B*2, *C*3 or *C*4). Because the host is either owner or reader in *C*4, the `unmap` can be executed before the copy without any further synchronization points.

There are three special cases where the home node is used without the existence of a host or IACC device (*E*1, *E*2 or *E*3). If the data was written before the application, it must be read from the home node (*E*2). In this case there is no input event for synchronization (*c*5). If data is transferred between two DACC from two different contexts, it must be copied via the home node (*E*3). In this case, data can also be read from the home node using a synchronization point (*E*2). If the data is needed after the application, it must be written back to the home node (*E*1).