Declarative Programming Techniques for Hardware Synthesis of Image Processing Applications

Deklarative Programmierungstechniken zur Hardware-Synthese von Bildverarbeitungsanwendungen

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

M. Akif Özkan

aus Ankara

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Tag der mündlichen Prüfung: 17.11.2023

Gutachter: Prof. Dr.-Ing. Jürgen Teich
Prof. Dr.-Ing. Dirk Koch

"Simplicity is the ultimate sophistication."
Leonardo da Vinci

Abstract

Traditional hardware description languages (HDLs), such as VHDL and Verilog, are widely used for designing digital electronic circuits, e.g., application-specific integrated circuits (ASICs), or programming field-programmable gate arrays (FPGAs). However, using HDLs for implementing complex algorithms or maintaining large projects is tedious and time-consuming, even for *experts*. This also prevents the widespread use of FPGAs. As a solution, High-Level Synthesis (HLS) has been studied for decades to increase productivity by, ultimately, taking a behavioral description of an algorithm (*what the circuit does?*) as design entry and automatically generating a register-transfer level (RTL) implementation. Commercial HLS tools start from well-known programming languages (e.g., C, C++ or OpenCL), which were initially developed for programmable devices with an instruction set architecture (ISA). Yet, these tools deliver a satisfactory quality of hardware synthesis results only when programmers describe hardware-favorable implementations for their applications (*how the circuit is built?*) exploiting, e.g., a specific memory architecture, control path, and data path. This requires an in-depth understanding of hardware design principles. To adopt software programming languages for hardware design, each HLS tool uses its own language dialect and introduces a non-standard set of pragmas. The mixed-use of software and hardware language abstractions hinders a purely behavioral design and makes optimizations hard to understand since the expected code is neither a pure hardware description nor a regular software implementation. Furthermore, a code optimized for one HLS tool has to be changed significantly to target another HLS tool and performs poorly on an ISA. We believe that the next step in HLS will be on the language side, overcoming productivity, portability, and performance hurdles caused by behavioral design deficiencies of existing tools.

This dissertation presents and evaluates three distinct solutions to separate the description of the behavior (*what?*) of an algorithm from its implementation (*how?*) while providing high-quality hardware synthesis results for the class of image processing applications. This is achieved by generating highly optimized target-specific input code to commercial HLS tools from high-level abstractions that capture parallelism, locality, and memory access information of an input application. In these approaches, an image processing application is described as a set of basic building

blocks, namely point, local and global operators, without low-level implementation concerns. Then, optimized input code is generated for the selected HLS tool (Vivado HLS or Intel OpenCL SDK for FPGAs) using one of the following different programming techniques: (i) a source-to-source compiler developed for an image processing domain-specific language (DSL), or (ii) template metaprogramming to specialize input C++ programs at compile time, (iii) a partial evaluation technique for specializing higher-order functions.

We present the first source-to-source compiler that generates optimized input code for Intel OpenCL SDK for FPGAs from a DSL. We use Heterogeneous Image Processing Acceleration (Hipacc), an image processing DSL and a source-to-source compiler initially developed for targeting graphics processing units (GPUs). The Hipacc DSL offers high-level abstractions for point, local, and global operators in form of language constructs. During code generation, the compiler front end transforms input DSL code to an abstract syntax tree (AST) representation using Clang/LLVM compiler infrastructure. By leveraging domain knowledge captured from input DSL code, our backend applies several transformations to generate a description of a streaming hardware pipeline. At the final step, Hipacc generates OpenCL code as input to Intel's HLS compiler. The quality of our hardware synthesis results rivals with those obtained from Intel's hand-optimized OpenCL code examples in terms of throughput and resource usage. Furthermore, Hipacc's code generation achieves significantly higher throughput and uses fewer resources compared to Intel's parallelization intrinsics.

Second, we present an approach based on template metaprogramming for developing modular and highly parameterizable function libraries that also deliver high-quality hardware synthesis results when compiled with HLS tools. In this approach, the library application programming interface (API) consists of high-level generic functions for declaring building blocks of image processing applications, e.g., point, local, global operators, unlike typical libraries that offer functions for complete algorithms, e.g., OpenCV. The library is optimized with Vivado HLS best practices as well as hardware-centric design techniques such as deep pipelining, coarse-level parallelization, and bit-level optimizations. The library contains more than one template design for each algorithmic instance to be able to utilize implementations optimized for input parameters. For example, it includes multiple implementations of image border handling and coarse-level parallelization strategies considered for different input parameters of a local operator specification. Furthermore, a compile-time selection algorithm is proposed for selecting the most suitable implementation according to an analytical model derived for resource usage, speed, and latency. In this way, low-level implementation details are hidden from users.

In addition to the presented advantages of using high-level abstractions for raising the abstraction level in HLS, we show that this approach is beneficial for achieving performance portability across different computing platforms. Similar to FPGAs, the

performance capabilities of central processing units (CPUs) and GPUs can fully be leveraged only when application programs are tuned with low-level architecture-specific optimizations. These optimizations are based on fundamentally different programming paradigms and languages. As a solution, Khronos released OpenVX as the first industrial standard for graph-based specification of computer vision (CV) applications. The graph-based specification allows optimizing memory transfers between different CV functions from a device-specific backend. Furthermore, the standard hides low-level implementation details from the algorithm description. For instance, memory hierarchy and device synchronization are not exposed to the user. However, the OpenVX standard supports only a small set of computer vision functions and does not offer a mechanism to incorporate user code as part of an OpenVX graph. As the next step, HipaccVX is presented as an OpenVX implementation and extension, supporting code generation for a wide variety of computing platforms. HipaccVX leverages OpenVX's standard API and graph specification while offering new language constructs to describe algorithms using high-level abstractions that adhere to distinct memory access patterns (e.g., local operators). Thus, it supports the acceleration of user-defined code as well as OpenVX's CV functions. In this way, HipaccVX combines the benefits of DSL design techniques with an industrial standard specification.

Finally, AnyHLS, a novel approach to raise the abstraction level in HLS by using partial evaluation as a core compiler technology is presented. Solely *one* language and *one* function library are used to generate target-specific input code for two commercial HLS tools, namely Xilinx Vivado HLS and Intel FPGA SDK for OpenCL. Hardware-centric optimizations requiring code transformations are implemented as higher-order functions, without using tool-specific pragma extensions. Extending AnyHLS with new functionality does not require modifications to a compiler or a code generator written in a different (host) language. Contrary to metaprogramming, the well-typedness of a residual program is guaranteed. As a result, significantly higher productivity than the existing techniques and an unprecedented level of portability across different HLS tools are achieved. Productivity, modularity, and portability gains are demonstrated by presenting an image processing library as a case study.

Acknowledgments

I would like to express my gratitude towards my doctoral adviser Prof. Dr.-Ing. Jürgen Teich for his invaluable supervision, the flexibility to explore and pursue new research ideas, and the opportunity to work under excellent research conditions. I would also like to thank Professor Dirk Koch for being the co-examiner of my thesis.

I would like to extend my sincere gratitude to PD Dr.-Ing. Frank Hannig, the leader of the architecture and compiler design (ACD) group, for his support, constant encouragement, and guidance on my doctoral work.

I would like to thank all my colleagues at the Chair for Hardware/Software Co-Design for the pleasant work environment and intriguing discussions. I would like to particularly thank my former office mates Oliver Reiche and Bo Qiao for the engaging conversations and fruitful collaborations. My special thanks go to Oliver Reiche for his technical and social support during the early stages of my PhD work and settlement in Germany.

I am also grateful to Prof. Dr. S. Berna Örs, Prof. Dr. H. Fatih Uğurdağ, and Assis. Prof. Dr. Ramazan Yeniçeri for advising me on my first scientific research projects and publications, guiding me in gaining in-depth knowledge of digital circuit design, and encouraging me to pursue a PhD. Without them, I would not have been fully prepared to begin my doctoral studies.

Last, and by no means least, thanks to my family for their unconditional love and never-ending support. I would like to thank my parents for fostering my scientific curiosity as a child and supporting my technical-oriented advancement. Finally, I would like to thank my wife for her support and patience.

Contents

1 Introduction **1**

1.1 Compute Performance is Power Constrained 1
1.2 Post-Dennard Scaling Era 2
1.2.1 Ongoing Multicore Evaluation has Hit the Power Wall . . . 3
1.2.2 Supplying Data and Instructions has High Energy Cost . . . 5
1.2.3 Specialized Hardware Provides Energy Efficiency 6
1.2.4 Specializing Software for Hardware is Crucial for Achieving High Performance 9
1.2.5 The Cost of Mapping Algorithms to Specialized Hardware is High 11
1.3 Traditional Hardware Design Flow 12
1.4 High-Level Synthesis 15
1.4.1 A Brief History of HLS Tools: Analysis of Past Successes and Failures 19
1.4.2 Limitations of Existing HLS Tools 21
1.5 Our Approach: Raising the Abstraction Level in HLS for a Restricted Application Domain 23
1.6 Contributions 26

I Mapping Image Processing Algorithms to Hardware **29**

2 Image Processing with Hardware Pipelines **31**

2.1 Image Processing Applications 31
2.1.1 The Power and Computational Requirements 33
2.1.2 Performance-Relevant Abstractions for Domain-Specific Code Generation 34
2.2 Mapping Algorithms to FPGAs using C-based HLS 38
2.2.1 An Introduction to FPGAs 38
2.2.2 Area and Speed Considerations in FPGA Design 39
2.2.3 Writing Software for C-based HLS 41

2.2.4 Optimizing Software for C-based HLS 43
2.3 The Challenge of Designing Image Processing Circuits with C-based HLS Tools 49
2.3.1 Hardware Implementation of Image Processing Operators . 49
2.3.2 Motivational Example 51
2.3.3 Proposed Approach 55

3 Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing 57
3.1 Introduction 58
3.1.1 Loop Coarsening 61
3.1.2 Notation 65
3.2 Schmid's Loop Coarsening 66
3.3 Proposed Loop Coarsening 66
3.3.1 Fetch and Calc 67
3.3.2 Calc and Pack 70
3.4 Analysis of Border Handling 70
3.4.1 Naïve Border Handling 72
3.4.2 Separated Border Handling 73
3.5 Hardware Architectures for Image Border Handling 76
3.5.1 Row Selection 76
3.5.2 Column Selection 77
3.5.3 Loop Coarsening 80
3.6 Architecture Selection 82
3.6.1 Border Handling Type Selection 82
3.6.2 Loop Coarsening Architecture Selection 84
3.6.3 Architecture Selection Algorithm 85
3.7 Evaluation and Results 85
3.7.1 Coarsening Types 87
3.7.2 Border Handling Architectures 88
3.7.3 Effects of Target Speed Constraint 90
3.7.4 Implementation Results 91
3.8 Conclusion 91

II Lifting High-Level Synthesis for Image Processing 93

4 FPGA-Based Accelerator Design from a Domain-Specific Language 95
4.1 Introduction 95
4.2 Background: The Hipacc Framework 98
4.2.1 Overview of Hipacc DSL 99
4.2.2 Data Storage 99

4.2.3 Read/Write Operations on Images 101
4.2.4 Stencil Patterns 102
4.2.5 Describing Computational Patterns 103
4.2.6 Point and Local Operators 104
4.2.7 Global Reduction 107
4.3 Generating Hardware Accelerators From Hipacc 108
4.3.1 Background: Overview of the Compiler Work Flow 110
4.3.2 Generating a Streaming Pipeline 110
4.3.3 Generating Host Code 113
4.3.4 Generating Kernel Codes (Hardware Accelerators) 114
4.4 Metaprogramming in C-based HLS: Alleviating the Tasks of a Source-to-Source Compiler 115
4.4.1 Metaprogramming Techniques in OpenCL and C++ 116
4.4.2 C++ Metaprogramming for Building High-Level Abstractions 119
4.5 A Comprehensive Metaprogramming Library for Highly Efficient C++-based High-Level Synthesis of Image Processing Applications . 122
4.5.1 Motivational Example 122
4.5.2 Overview of the Library 125
4.5.3 An Example Application: Harris Corner Detection 133
4.5.4 A Deeper Look into the Library 135
4.6 Evaluation and Results 142
4.6.1 Algorithms 142
4.6.2 Hipacc Compiler Backend Targeting Intel FPGAs 143
4.6.3 The Metaprogramming Library Targeting Xilinx Vivado HLS 149
4.7 Related Work 153
4.8 Conclusion 155

5 HipaccVX: Wedding of OpenVX and DSL-based Code Generation 157
5.1 Introduction 158
5.2 OpenVX programming model 160
5.3 Deficiencies of OpenVX 162
5.4 Our Approach: DSL-based Code Generation for OpenVX 163
5.4.1 Computational Abstractions 163
5.4.2 Advantages of using Computational Abstractions 165
5.5 Implementation: The HipaccVX Framework 166
5.5.1 Image Processing DSLs 166
5.5.2 DSL Back End and User-Defined Kernels 167
5.5.3 OpenVX Graph and System-Level Optimizations 170
5.6 Evaluation and Results 172
5.6.1 Acceleration of User-Defined Nodes 173
5.6.2 System-Level Optimizations based on OpenVX Graph . . . 174
5.6.3 Evaluation of the Performance 174

5.7 Related Work . . . 178
5.8 Conclusion . . . 179

III High-Level Synthesis with Partial Evaluation 181

6 AnyHLS: High-Level Synthesis using Partial Evaluation 183
6.1 Introduction . . . 184
6.1.1 Raising the Abstraction Level in HLS . . . 184
6.1.2 Main Contributions . . . 185
6.2 Overview, Background, and Related Work . . . 187
6.2.1 AnyDSL Compiler Framework . . . 187
6.3 The AnyHLS Library . . . 189
6.3.1 HLS Code Generation . . . 190
6.3.2 Building Abstractions for FPGA Designs . . . 191
6.4 A Library for Image Processing on FPGA . . . 197
6.4.1 Vectorization . . . 198
6.4.2 Memory Abstractions for Image Processing . . . 199
6.4.3 Loop Abstractions for Image Processing . . . 202
6.5 Evaluation and Results . . . 204
6.5.1 Applications . . . 205
6.5.2 Optimizations . . . 205
6.5.3 Hardware Design Evaluation . . . 208
6.6 Conclusions . . . 213

7 Conclusion and Future Directions 215
7.1 Summary . . . 216
7.2 Future Work . . . 219

German Part 221

Bibliography 227

Author's Own Publications 243

Acronyms 247

1 Introduction

1.1 Compute Performance is Power Constrained

In 1965, Gordon Moore predicted that the number of transistors fabricated on a chip of the same size would double every 18–24 months. His prediction (known as Moore's law) has held true for the past 50 years, making transistors smaller, faster, and cheaper to manufacture. Correspondingly, transistor gate speed has been improved by 100x since commercial CMOS microprocessors were introduced in the mid-80s. Correspondingly, the performance of modern uniprocessors is increased by over 3000x at the application level [Hor14].

Moore's law is expected to be valid for at least another decade, but exploiting its advantages has become very hard since power became the primary constraint on performance [IRT16]. Dennard's constant-field scaling meant that power consumption per chip area stayed constant as the transistors shrank [DGY+74]. That is, smaller transistors operate with a faster clock frequency but require less voltage, current, and chip area and, therefore, consume less power. However, it is discovered that leakage current in CMOS technology rises unacceptably high when the supply voltage is decreased to below 1V. Consequently, Dennard Scaling broke down around 2005 [Tay12], and so did the exponential growth in computing performance achieved by transistor technology scaling.

If the frequency scaling had been continued, chips would be running at 10-30 GHz in 2011 [Gel01], dissipating heat equivalent to that of nuclear reactors (proportional to size) [CCF+10]. However, commercial processors hit the *power wall* around 4GHz of clock frequency in the early 2000s (see Figure 1.1) [IRT16]. The rate of heat transfer to the environment defines thermal limits. A low-cost heatsink air cooling system that produces noise at an acceptable level for an office allows desktop computers and servers to be powered around 130W [MCC06; Hor14]. The requirements are more stringent for laptops (15-20W) and hand-held devices such as smartphones (2-3W) and tablets (7-10W) [GPW+16; MCC06].

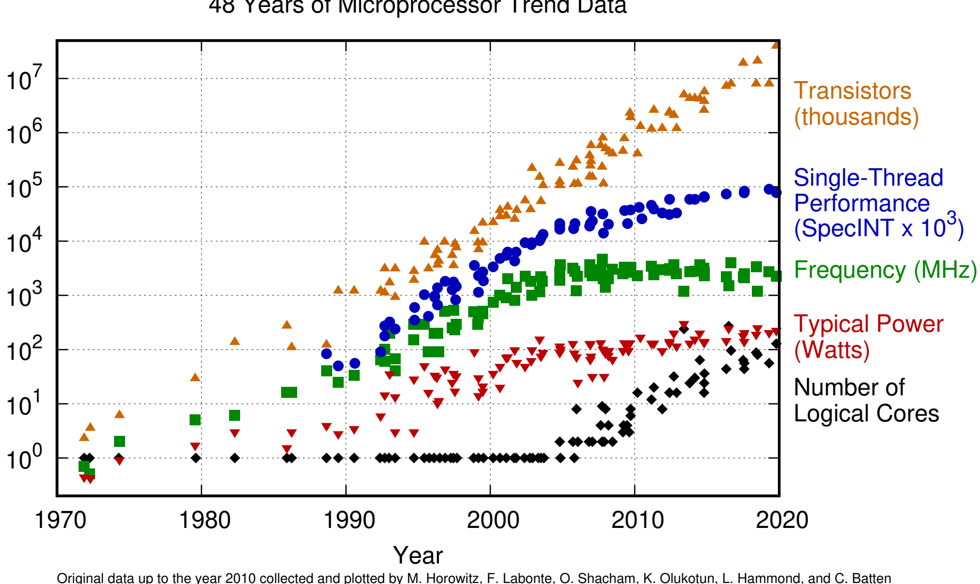


Figure 1.1: Performance of computing has grown exponentially until the processors hit the *power wall* around 4GHz of clock frequency. Despite the exponential growth of the transistor count, the performance of a sequential processor performance reached its limit. Power has become a critical constraint on performance. (© 48 Years of Microprocessor Trend Data by Karl Rupp used under CC BY 4.0)

1.2 Post-Dennard Scaling Era

Despite the limitations of technology scaling, the demand for computing significantly increased in the last decade and is expected to rise even with a higher speed in the next decade [IRT16; IDC16]. Every day, large volumes of digital data are created, replicated, and consumed. According to International Data Corporation (IDC), this amount was 59 zettabytes (ZB) in 2020 [IDC16], and the world will create more than three times the data over the next five years than it did in the previous five. The widespread craze of embedded media devices made computing and communication over the internet ubiquitous, e.g., more than one million mobile phones are sold every year [DBB+08]. Communicating through a 100-Mbs orthogonal frequency-division multiplexing (OFDM) channel requires 210 to 290 giga operations per second (GOPS) [SJ07]. Entertainment services such as Netflix and Amazon prime have become a regular part of daily life, demanding high-bandwidth data streaming from the "cloud "for millions of users. Netflix is accounted for more than one-third of internet traffic in USA [Jon18]. Advancements in science, e.g., in astrophysics, require hundred- to thousandfold increases in data volumes from supercomputers, accelerators, sensor networks, telescopes, satellites, and high-throughput instruments compared to a

decade ago [BHS09].

In 2018, the Information and Computing Technologies (ICT) ecosystem accounted for more than 2 % of the global carbon emission [Jon18]. For instance, training one of the common natural language processing models can emit 626,155 lbs of carbon dioxide – that is nearly five times the lifetime emissions of the average American car [SGM19]. According to forecasts [Jon18], ICT will consume 8 % of total electricity demand by 2030 in the best-case scenario, whereas more pessimistic scenarios predict this amount to be 21 % of the whole globe. For these reasons, energy consumption and the carbon footprint of computation have become a serious concern.

Moore's law started to slow down around 2000 but remained valid for the last two decades (see Figure 1.1). The International Technology Roadmap for Semiconductors (ITRS) organization expects transistor density to increase for at least another decade [IRT16]. However, Dennard scaling almost diminished by 2012. Having an abundance of transistors but a limited power budget encouraged computer architects to improve energy efficiency (measured by Joules per operation (J/op)) by exploring a new class of architectural techniques [Hor14; HP19]. This boosted the so-called multicore evaluation [Mar14; BC11].

1.2.1 Ongoing Multicore Evaluation has Hit the Power Wall

Computer architects achieved a yearly performance improvement of 50 % during the Dennard scaling era (between 1986 - 2002) mainly by frequency scaling of transistors along with exploiting instruction-level parallelism (ILP) (see Figure 1.2). Processors are designed with deep pipelines that simultaneously fetch and execute several instructions using branch prediction techniques. However, power becoming the primary constraint on performance, computer architects noticed that ILP wastes significant energy in real-world applications because of branch mispredictions. For instance, reducing a processor's computation "waste" to 10 % requires a branch prediction mechanism to work correctly 99.3 % of the time [HP19].

Instead, manufacturers started including more processor cores in each die to exploit parallelism in algorithms. The power spent from a single core is decreased by operating processor cores at a lower peak performance. This technique decreases the performance of a single core but multiplies the overall performance and increases energy efficiency since several cores are able to perform operations in parallel using the power budget at the thermal limits.

This approach was initially beneficial but quickly reached its diminishing returns at a point where increasing energy efficiency causes huge performance penalties and vice versa [Hor14]. Consequently, the era of "dark silicon" started, where fractions of a chip were turned off, dimmed to idle or underclocked to decrease power dissipation. These approaches are seen as »spending« area for »buying« energy efficiency [Tay12].

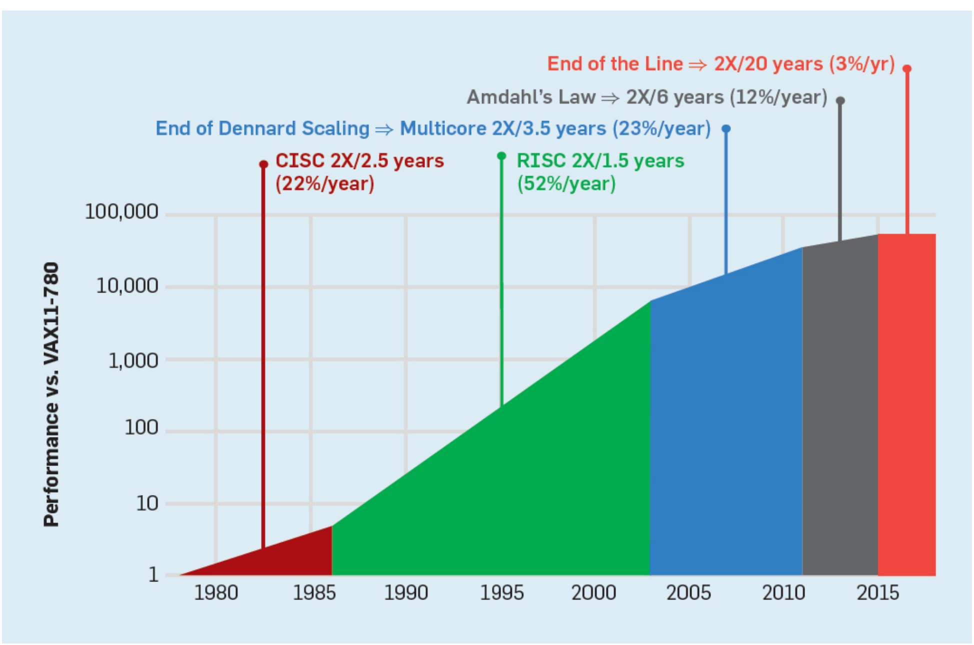


Figure 1.2: Forty years of growth in computer performance, where the benchmarks use integer programs. (© Used with permission of ACM, from [HP19]; permission conveyed through Copyright Clearance Center, Inc.)

Furthermore, real-world applications are never fully parallel or sequential; in fact, they vary significantly. As Amdahl states: "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude " [Amd67]. Therefore, executing an application with an optimal parallelization factor according to Amdahl's law remains a complex problem for general-purpose computing despite the research efforts summarized below: The concept of turbo modes is one solution to avoid performance decrease for serial code, where the supply voltage of one core increases while the other cores are put in a low-power state. Dynamic voltage and frequency scaling (DVFS) techniques are developed to optimize battery life for mobile processors by adjusting frequency and supply voltage. Finally, three classes of multicores have emerged to tackle the limitations of Amdahl's law: symmetric, asymmetric, and dynamic architectures [HM08]. While the symmetric multicore processors include a number of the same processor cores, the asymmetric architectures consist of one or more cores that are more powerful than others and utilized to execute sequential code. Dynamic multicore chips provide a more balanced parallelism support that allows configuring multiple small cores to a larger one at runtime, similar to superscalar processors [MSD17; IKK+07]. However, scheduling software tasks for asymmetric and dynamic multicore chips is difficult and often adds overhead.

What is more, multicore processors periodically waste energy to communicate

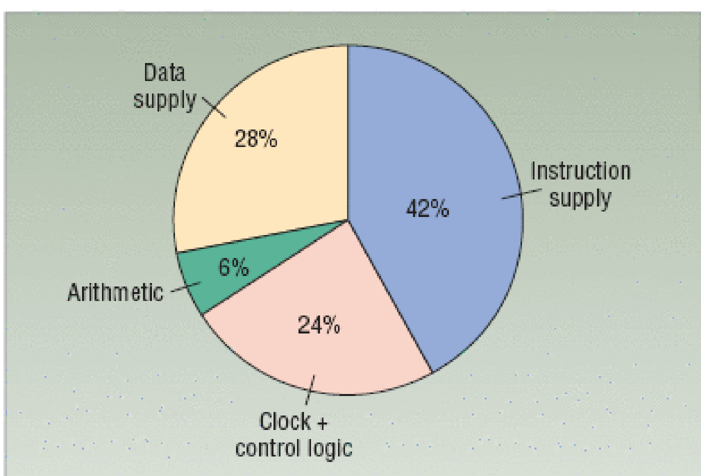


Figure 1.3: 70% of the processor's energy is spent on supplying data and instructions, whereas only 6% is spent on arithmetic operations. (Figure reprinted from [DBB+08], © 2008 IEEE)

and synchronize with each other. The data distribution to cores adds additional overhead because of the underlying complexity. The research on multicore chips is ongoing. However, the performance improvements are decreased to a few percents per year (see Figure 1.2). The fraction of a multicore chip running at full peak performance frequency decreases with each process generation [Tay12; IRT16]. For these reasons, multicore processing is not seen as the final solution to satisfy exponentially increasing customer demand for performance [Hor14; HP19; Tay12; IRT16].

1.2.2 Supplying Data and Instructions has High Energy Cost

Modern processors spend much more energy on supplying data and instructions than actual computations. For instance, Dally et al. 's analysis on an embedded processor [DBB+08] shows that only 6 % of the processor's energy is used for performing arithmetic operations (see Figure 1.3). Of this 6 %, 59 % is spent on useful computations, while 41 % is spent on operations like calculating memory addresses or updating loop indices [DBB+08]. The control logic of the fetch-decode-execute architecture uses 24 % of the processor energy. Further research [DBB+08] shows that an average processor fetches 1.7 instructions for every useful instruction, where algorithmic calculations are considered as useful operations.

This huge inefficiency is better understood when the instruction energy cost breakdown shown in Figure 1.4 is analyzed. An addition instruction costs 70 pJ, while a 32-bit integer addition operation costs only 0.1 pJ. This means only 0.15 % of the energy consumption is used for the actual computation, whereas 55.65 %

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3.1pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1.1pJ
32 bit	3.7pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

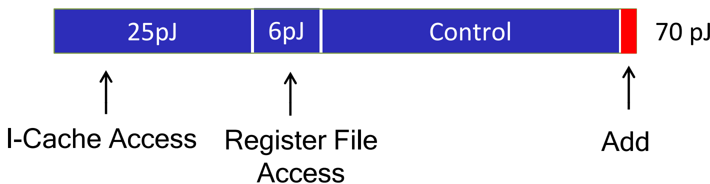


Figure 1.4: Instruction energy cost breakdown of a simple in-order processor for various operations in 49nm 0.9V. (Figure reprinted from [Hor14], © 2014 IEEE)

overhead is introduced solely for the control logic. It is also clearly seen that memory access costs significantly more than arithmetic operations. External memory (DRAM) access costs 1.3-2.6 nJ, spending 130x to 260x energy than the cache access. Yet, access to the cheapest cache for the processor shown in Figure 1.4 consumes 10 pJ, which is 100x of a 32-bit integer addition and 2.7x of a multiplication.

A cache is a small on-die memory that is fast and more energy efficient. Modern processors use multi-level caches to minimize external memory access. This decreases the external memory (DRAM) access, thus significantly optimizes energy efficiency. However, caches still consume roughly half the energy of the processor [Hor14] and work well only when the locality is very high, and data is not very large [HP19].

1.2.3 Specialized Hardware Provides Energy Efficiency

Flexibility (programmability) costs energy efficiency, as shown in Figure 1.5. Processors are designed to be more flexible at the cost of inefficiencies, such as high control overhead and energy waste caused by a complicated memory hierarchy.

FPGAs and ASICs are quite different compared to *programmable* processors such as CPUs and GPUs since they do not follow the concept of an ISA. An ISA is an abstract computer architecture model that serves as an interface between software and hardware. It defines the format of the instructions and how they are fetched

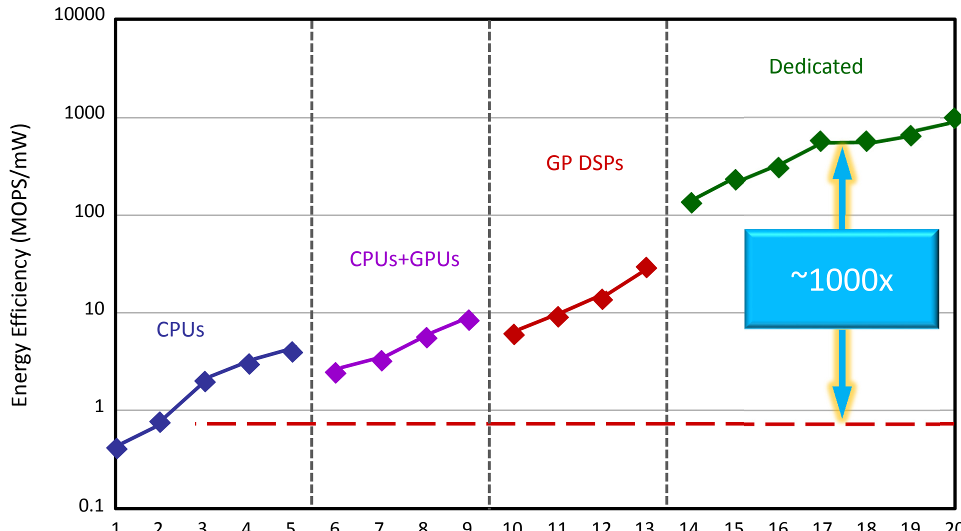


Figure 1.5: Specializing hardware implementations for ASIC could improve energy efficiency by 100x. 20 chips from ISSCC, the top international conference in chip design, over 5 years (2009-2013) are analyzed in terms of energy efficiency. (© Used with permission of Springer, from [MB12]; permission conveyed through Copyright Clearance Center, Inc.)

from program memory, decoded, and then executed on a number of arithmetic logic units (ALUs). ALUs operate on a fixed memory system – that is, data resides in a register file or is loaded from memory.

Domain-Specific Architectures

Domain-specific architectures (DSAs) are ISA designed for a class of applications [HP19]. They are often Turing-complete, but they provide good performance only for the target application domain. DSAs include tensor processing units (TPUs) developed for artificial intelligence (AI), GPUs that exploit massive data-level parallelisms (DLPs) in application domains such as graphic processing, and various digital signal processor (DSP) architectures designed for different classes of applications, e.g., audio processing. As shown in Figure 1.5, DSAs are able to provide 10x better energy efficiency since they allow

(i) tailoring control- and data-path of an ISA to a target application domain for exploiting parallelism more effectively. For instance, single instruction, multiple data (SIMD)[1] engines reduce instruction overhead and exploit DLP

[1]Single Instruction, Multiple Data (SIMD) units are CPU components for vector processing, i.e., they execute the same operation on multiple data elements in parallel.

at the cost of flexibility. The optimal size of a SIMD unit and amount of SIMD engines utilized for an ISA depends on the target application and the performance goal. GPUs have many cores with their own SIMD ALUs, called single instruction, multiple threads (SMT). Similarly, a DSA could be designed as a very long instruction word (VLIW) or as an out-of-order superscalar processor.

(ii) deploying a more suitable memory hierarchy, often controlled by the user for the target application, instead of a standard cache mechanism. This specialization decreases external memory access as well as energy and performance waste caused by cache misses.

(iii) using a lower precision, e.g., 8-16 bit integer arithmetic units instead of floating point.

DSAs are often controlled by a CPU, and called *accelerators*. Today's computing systems are typically heterogeneous, consisting of different computing architectures for different tasks. In fact, modern processor chips consist of a bunch of dedicated circuits and accelerators, such as GPUs. This diversity makes both designing and programming today's computing platforms challenging.

Application-Specific Integrated Circuits

Chips dedicated to one or a limited set of applications sacrifice programmability for a few orders of magnitude better energy efficiency. Designing dedicated chips (ASICs) allows

(i) utilizing only the required resources,

(ii) reducing the bit-precision to a minimum,

(iii) exploiting the parallelism of an algorithm with a dedicated data path, thus achieving the targeted throughput with a slower clock frequency,

(iv) eliminating unnecessary instructions and the control logic that would be required for fetch-decode-execute stages of an ISA, and

(v) reducing memory access by exploiting locality and the overheads of a complicated memory hierarchy.

However, designing a chip specific to one application is often not worthwhile even when programmability is not required or inaccessible for application developers due to its high monetary cost and lengthy time-to-market. Furthermore, the costs of designing chips increases even more with the transistor technology scale since designing, verifying, and validating more transistors are more complex tasks.

Field-Programmable Gate Arrays

In between dedicated chips and processors, FPGAs provide both programmability and high specialization of the hardware. Similar to dedicated chips, an FPGA implementation can be tailored and highly parallelized for the target application by utilizing only the necessary amount of resources. An FPGA consists of a mass of reconfigurable digital logic cells surrounded by a sea of reconfigurable interconnects (conductors and switches) for wiring up the logic cells. Reconfiguration happens in the field. The end-user, who can implement any logic function—even a processor—, is only limited by physical constraints.

FPGAs especially shine whenever huge amounts of data can stream through the FPGA circuitry while being processed at high speed, low latency, and with high energy efficiency since instruction decoding becomes needless and off-chip memory accesses can be reduced to a minimum. Higher throughputs are achieved via very deep pipelines with hundreds (even thousands) of stages that are not stalled by branch mispredictions. This paradigm is also known as data-flow computing, where a great many computing units (CUs) can work in parallel, similar to GPUs. However, FPGA designs can be fully application-tailored since each CU can be specialized differently.

A drawback of FPGAs compared to ISAs, e.g., state-of-the-art CPUs and GPUs, is that typically FPGA implementations offer a lower achievable clock frequency. Another is the expensive market prices, caused by many factors, including large chip area, engineering cost, and lower market volume. Therefore, FPGAs are used in industry when they provide significantly better results regarding a design goal, which could be power, energy, or performance. Mostly, FPGA implementations provide better results only when the hardware implementation of an application is tailored (*specialized*) to its specification. That is, a custom memory hierarchy, data path, and control path have to be implemented to exploit a given application's spatial and temporal locality.

1.2.4 Specializing Software for Hardware is Crucial for Achieving High Performance

The performance of a programmable architecture depends on how fast software does its task on that hardware. Typically, programmers write software in a high-level programming language such as C++ or Python and compile it to a machine code (instructions) to obtain an executable binary for a device. The content of a machine code depends on the target device's instruction set and the compiler that translates the high-level program description into instructions. Through this phase, compilers apply transformations on user code to improve performance, yet these are not enough to achieve optimal performance. Often, modifying user code can lead to

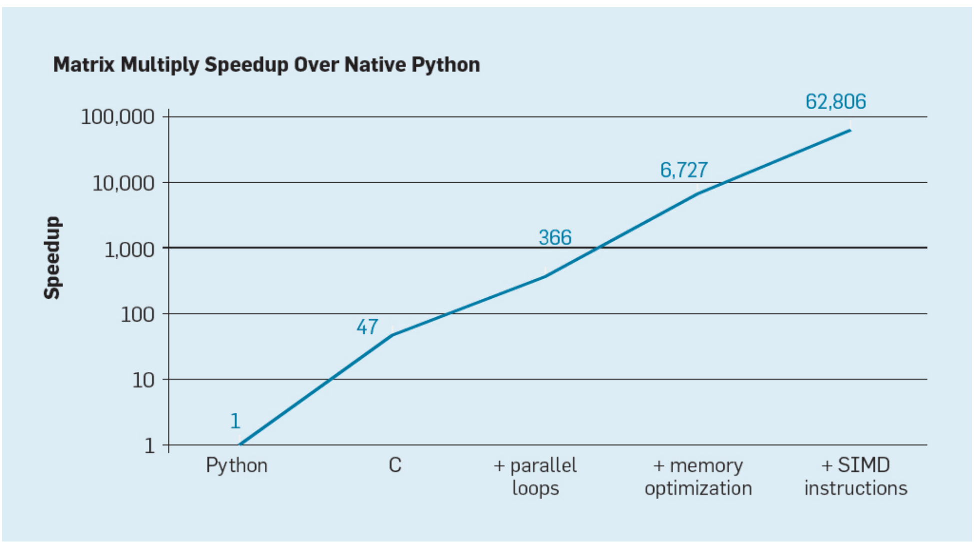


Figure 1.6: Software that runs hardware has a dramatic effect on performance. This is shown with five different versions of a matrix-multiplication program. The width and height of matrixes are 4096-by-4096, and the initial code is written with three nested loops using Python's `xrange`. The results show that optimizing a program for a target device significantly improves performance. However, this requires rewriting the program, leading to non-portable, lengthy code. For instance, hereby, the fastest program is written with AVX instructions and 20x longer than the slowest code. (© Used with permission of ACM, from [HP19]; permission conveyed through Copyright Clearance Center, Inc. The data is from [LTE+20])

performance gains of several orders of magnitude.

One factor that affects the performance of a program is the quality of an algorithm – that is, a program is considered to be more efficient when it requires less computational work or fewer resources to accomplish a task. Typically it is hard to find an optimal algorithm that provides the *best* performance for all possible inputs and the selected computing platform. For instance, many-core processors perform better when an algorithm has fewer sequential dependencies. Similarly, the time spent on each instruction depends on the target device. Thus, the optimality of an algorithm depends on how the target platform executes that algorithm.

Another factor is how well a description of an algorithm (software) is tailored for a specific architecture. Leiserson et al. [LTE+20] show that a performance improvement of 62.9x is achieved by rewriting the same matrix multiplication algorithm (see Figure 1.6). Initial python code is written with three nested loops. Simply rewriting this in a more efficient language increases the speed by 47x. Then, restructuring the code to leverage specific features of the underlying processor improves the performance

by 1300x faster. These include parallelizing loops to utilize all 18 processor cores and exploiting memory hierarchy. In the last step, Intel's Advanced Vector Extensions (AVX) instructions are used to better utilize the SIMD units. These optimizations require programmers to understand the underlying computing platform and consume a considerable time. The final code becomes 20x longer than the initial one, which is hard to read and understand. Furthermore, the optimized code is not portable – it cannot be run on platforms that do not support AVX instructions and perform poorly on devices with fewer processor cores or a different memory system.

1.2.5 The Cost of Mapping Algorithms to Specialized Hardware is High

Despite the enormous benefits, designing a specialized accelerator is not available to a great portion of application developers because of its high cost. The expensive cost of lithography masks and tooling, in addition to the high engineering effort required for place and route, and verification, makes ASIC design unattractive for small-scale products. FPGAs do not entail the aforementioned costs and offer higher flexibility by reconfiguration of logic. However, one needs to *design* hardware for FPGA-based acceleration (similar to ASIC design), unlike traditional software *programming*. In fact, a hardware implementation that is not specialized for the target algorithm is rarely beneficial since FPGAs have significantly slower clock frequency (speed) compared to alternative solutions, i.e., CPUs, GPUs, and mostly higher price per processing capacity.

An ASIC or an FPGA implementation, mostly, needs to be controlled by a CPU in a heterogeneous system. This introduces further development costs for drivers and controlling software. What is more, executing portions of an algorithm by the controlling software (instead of an additional specialized logic, which might require additional memory copies) often optimizes overall energy efficiency and performance.

In summary, designing a hardware accelerator, deploying it in a heterogeneous system (e.g., FPGA/CPU system), and optimizing the software is a complex and time-consuming task, which requires both hardware design and software programming skills as well as using different tools and programming languages. Modern systems-on-chips (SoCs) and data centers handle the complexity of the computing tasks, performance, and power requirements by such heterogeneity. However, the aforementioned costs hinder the development of such systems for small-scale products. A major aim of this thesis is to decrease the development costs of application-specific hardware design and controller software programming for a large class of image processing applications.

1.3 Traditional Hardware Design Flow

Today's industrial practice separates algorithm development, software optimization, and hardware design tasks to different developers, teams, or departments in a company. Many tools and frameworks increase an algorithm developer's productivity by hiding low-level programming details. However, this is not the case for a hardware designer. Figure 1.7 shows the typical design flow for designing digital circuits for a given algorithm.

A hardware designer starts with an algorithm (or compute-intensive parts of it) and a set of product requirements. The behavior of an algorithm is often described as a software program that abstracts from the execution model of an ISA (e.g., sequential execution of loops). The product requirements contain physical constraints such as chip area, number of FPGA resources, or design goals such as speed and throughput. Design constraints and goals highly depend on circuit implementation. Therefore, a team of chief architects (or the hardware designer) makes a rough estimation in the initial phase. This point in the design flow is called system-level abstraction level [GAG+09; Tei12], which also deals with early design space exploration of alternative hardware/software implementations, specification of interfaces and protocols, and ultimately, computing system design. This thesis takes as input a system specification (e.g., an FPGA/CPU system with a finite number of resources) and concentrates on solving the challenges of accelerator design for a given algorithm[2].

Traditionally, accelerator hardware is designed at register-transfer (RT) level. At this abstraction level, a digital circuit is described by modeling digital signals flowing between registers, their timing, and operations performed on them. Typically, a digital circuit contains two parts: a datapath and a control logic. The datapath consists of data storage (e.g., memories, multiplexers (MUXs), data buses) and computational circuits that produce output signals from input data (e.g., ALU, multipliers). The control logic produces control signals for the datapath (e.g., a finite state machine with data paths (FSMDs)). The behavior of a circuit is manually described clock cycle by clock cycle. Since all these low-level concerns are exposed to hardware designers in the traditional flow, design times are measured in weeks or months for software development tasks that usually take hours or days. Furthermore, reiterating an RTL design is a tedious task, even for *experts*: Small modifications require spending

[2]Mapping an algorithm into a heterogeneous system is a complex task. Electronic system-level (ESL) design methodologies [GAG+09; Tei12] offer a systematic approach to cope with this complexity, refining from a high-level functional specification to a low-level physical implementation. Tools and methodologies developed for profiling and partitioning an input algorithm for hardware-software co-design help specify design requirements and accelerator systems. This is a complex task and not the scope of this thesis. However, we will use the well-established terminology used to define abstraction layers in an ESL design flow and discuss how our work could contribute to the implementation of top-down ESL methodology in Chapter 7.

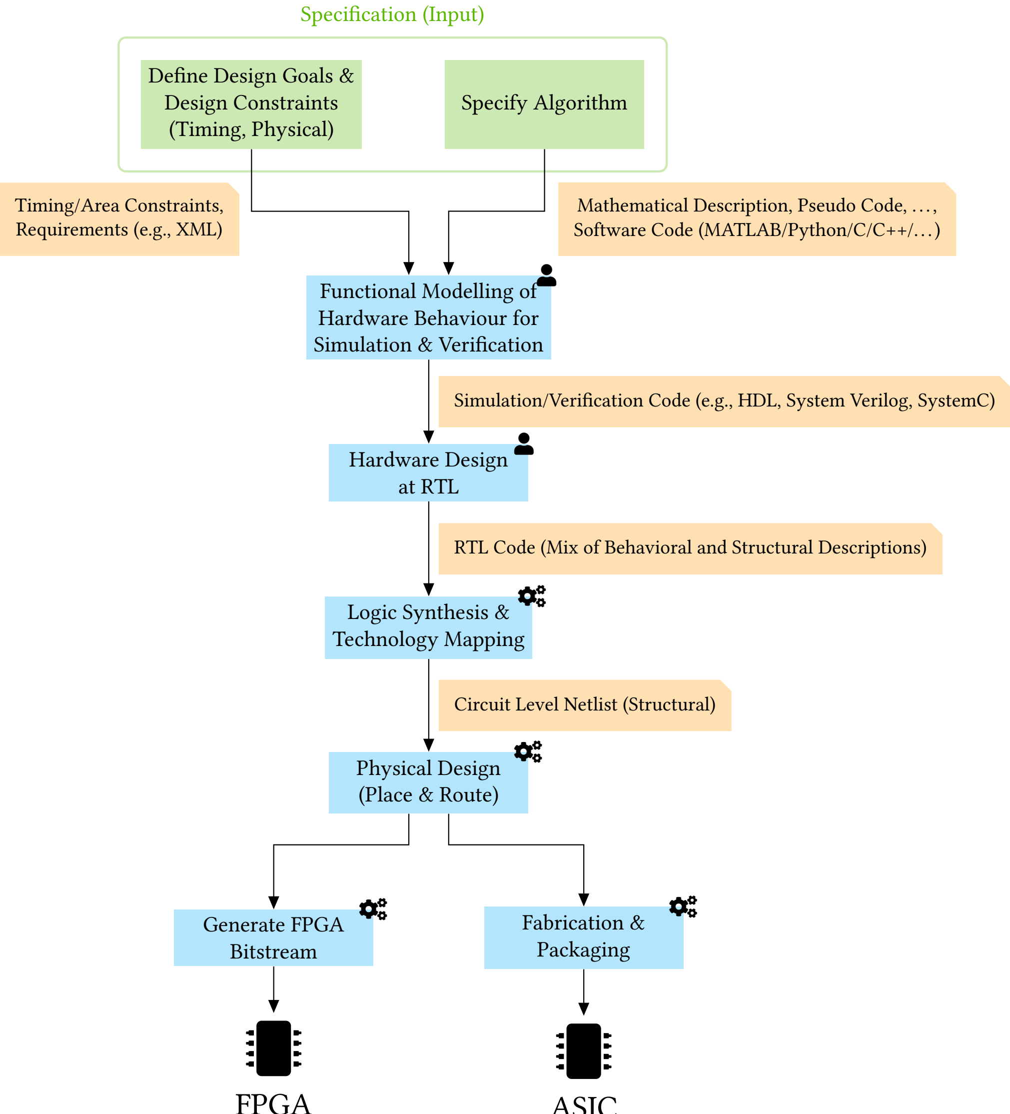


Figure 1.7: Traditional hardware design flow, which takes as input the specification of the algorithm, design specifications, and design constraints (denoted by green). Hardware design and synthesis tasks are colored blue, whereas inputs to these tasks are colored beige. The 👤 denotes the manual work of a hardware designer and ⚙ denotes the tasks automated by electronic design automation (EDA) tools.
A hardware designer must understand the input algorithm to design a circuit at RTL, which is an error-prone task that might take days to months. Typically, a test code is written for simulation and verification purposes before developing the hardware, mostly in a HDL using non-synthesizable behavioral language features. Further steps include the tasks of logic synthesis and physical design. Outputs of every design step must be simulated and verified, which might result in reiterating the RTL description and/or changing the design specifications. These requirements make hardware design time-consuming and complex for programmers without hardware design expertise.

many hours to debug the errors. Modifying the implementation of an algorithm for different design goals/constraints often entails designing a completely new RTL circuit.

Fortunately, mapping RTL descriptions to billions of transistors is highly automated through EDA tools. Given an RTL description, multiple synthesis steps are applied at the RT-level, gate-level, and circuit level (from abstract to concrete). First, control flow (state machines) and datapath are parsed from the given user code. Then, logic synthesis generates a logic gate network (also called gate-level netlist) from the Boolean expressions. This is a structural representation of the generated circuit in terms of technology-independent logic gates (such as OR, NOT, AND) and wires. Then, technology mapping transforms the gate-level logic into a set of pre-characterized and pre-designed gate layouts (cells) from a specific technology library. In the case of FPGAs, the technology mapping transforms a gate-level logic into an equivalent netlist of lookup table (LUT) blocks. Various optimizations are applied before and after technology mapping to increase design quality without changing functionality, e.g., synthesizing circuits that use fewer resources and/or perform faster.

Finally, the physical design step transforms the circuit-level netlist into a geometric description. The main concern of the physical design is to define the cells' positions and the routing between them by minimizing the chip area and wire length. A typical ASIC physical design cycle has the following main steps: logic partitioning, floorplanning, power planning, placement, clock tree synthesis, routing, and timing closure. Similarly, the FPGA physical design flow configures actual resources (e.g., LUTs, DSPs) and wiring connections.

Producing an actual ASIC from an RTL description still requires verification efforts in every step. Companies employ physical design experts to decrease the risk of producing incorrect circuits, thus wasting a tremendous amount of time and capital. FPGA designers have to verify the functionality and timing before and after physical design, but this is significantly easier. However, designing RTL circuits requires hardware design knowledge, where the learning curve is steep. Eliminating the required hardware design knowledge in RTL design would make FPGAs more amenable to software developers as an acceleration platform.

To sum up, most of the EDA tools take an RTL description as input for simulation and synthesis of circuits (as in Figure 1.7). This design flow has many limitations that decrease productivity:

(i) A hardware designer needs to understand the algorithm and extract the beneficial parallelism (i.e., exploit temporal and spatial parallelism) by looking at a source code initially written for sequential program execution. The design task becomes even more challenging when the code consists of function calls to a compiled runtime library where the code is not visible or low-level optimizations specific to an ISA (e.g., AVX instructions for an Intel processor).

(ii) Designing hardware at RTL is a time-consuming and complex task where developers are exposed to low-level details. At this abstraction level, designers describe a digital circuit by modeling digital signals flowing between registers, their timing, and operations performed on them. The behavior of a circuit is described by writing finite state machines (FSMs) to produce control signals for every clock cycle and a datapath such as an ALU to produce output signals from input data. A custom memory architecture is designed to increase data reuse and decrease off-chip communication. The precision of the memory elements and operations is defined at the bit level.

(iii) HDLs such as Verilog or VHDL provide language abstractions to describe hardware at a low level but require significant coding effort and verification time. They lack language features for writing generic and modular code, which are common in software programming languages. This makes design iterations time-consuming and error-prone, even for *experts*: The code needs to be rewritten for different performance or area objectives. In modern hardware description languages such as Chisel [BVR+12], VeriScala [LLQ+19], and MyHDL [Dec04], programmers can create a functional description of their design but stick to the RTL.

(iv) Furthermore, the gap between the concerns of algorithm development and hardware design prevents exploring optimizations at the algorithm level. Exploring/optimizing algorithms for ASIC/FPGA requires hardware design knowledge, whereas modifying algorithms for optimized hardware implementations requires algorithm knowledge.

High-Level Synthesis (HLS) was introduced to remedy these issues and has received much attention over the last 50 years.

1.4 High-Level Synthesis

High-level synthesis (HLS) is an automated method that generates a structural representation of a circuit at the RTL from a behavioral description at the algorithmic level. The input description could be untimed or partially timed [CLN+11]. The generated circuit must have the same functionality as the behavioral description. The main goal is to increase the productivity of hardware designers by raising the abstraction level from RT level by automating the low-level tasks such as register allocation and clock-level timing.

Figure 1.8 shows the design flow of a typical HLS tool. It starts by parsing (or compiling) the (behavioral) input description into a formal model. This model must be suitable for being used as an intermediate representation of the HLS flow, where data dependencies and control constructs, such as loops and branches, are expressed

within an acceptable memory footprint. It must also be capable of representing transformations applied for synthesis and optimization tasks, such as loop unrolling and loop merging. Many HLS tools use a control and data flow graph (CDFG) representation [NPP+20] for this purpose, where edges represent the control flow and nodes represent a sequence of statements with no branches, internal exit and entry points [CGM+09; GAG+09].

Further analysis and transformations are required to expose parallelism between nodes, extract useful information for the synthesis tasks, and minimize the effects of syntactic variations of the input code (behavioral description) on the generated circuit. Typically, the initial parsing/compilation is followed by optimizations such as strength reduction, dead code elimination, and constant folding. A great portion of HLS compilers rely on the algorithms used by software compilers for these optimizations and use the Low Level Virtual Machine (LLVM) compiler infrastructure for the compilation of user code.

In HLS, synthesis refers to three major tasks: allocation, scheduling, and binding [GR94; Tei12]. The allocation task deals with selecting the hardware resources that are necessary to map the behavioral functionality. It defines both the type and amount of resources according to a set of design constraints. The allocated resources could be for functional units, storage, or communication and interface synthesis. Defining the memory hierarchy, clocking scheme, and pipelining style is part of the allocation tasks. Achieving higher performance often requires using more resources. For instance, allocating multiple adders to compute in parallel increases the performance but uses more chip area. The allocation step aims to optimize the trade-offs between the performance requirements and resource usage constraints. For this, the allocation tools need to have exact specifications of resources (in terms of their area and performance values) as well as accurate metrics to reflect performance and resource usage [GR94] to help the user to select the appropriate design for his/her needs. Typically, allocation algorithms select from an RTL component library, in which specifications of common hardware building blocks such as physical components for the target chip design process or FPGA resource types are stored.

The scheduling maps the behavioral description, i.e., operations and memory access, into control steps (or states), and ultimately, to clock cycles. Existing scheduling algorithms have different strategies according to optimization goals and design constraints. For instance, a resource-constrained scheduling algorithm optimizes performance, i.e., minimizes the number of clock cycles for a given set of resources and clock cycle length. A time-constrained scheduling algorithm minimizes the number of resources (i.e., functional units) for a given number of control steps. A scheduler can assign multiple operations to the same control step or an operation to multiple control steps (by multicycling): Operations with no data dependencies could simply be assigned to the same control step(s) for parallel processing. Alternatively, multiple operations with small component delays could be chained according to their

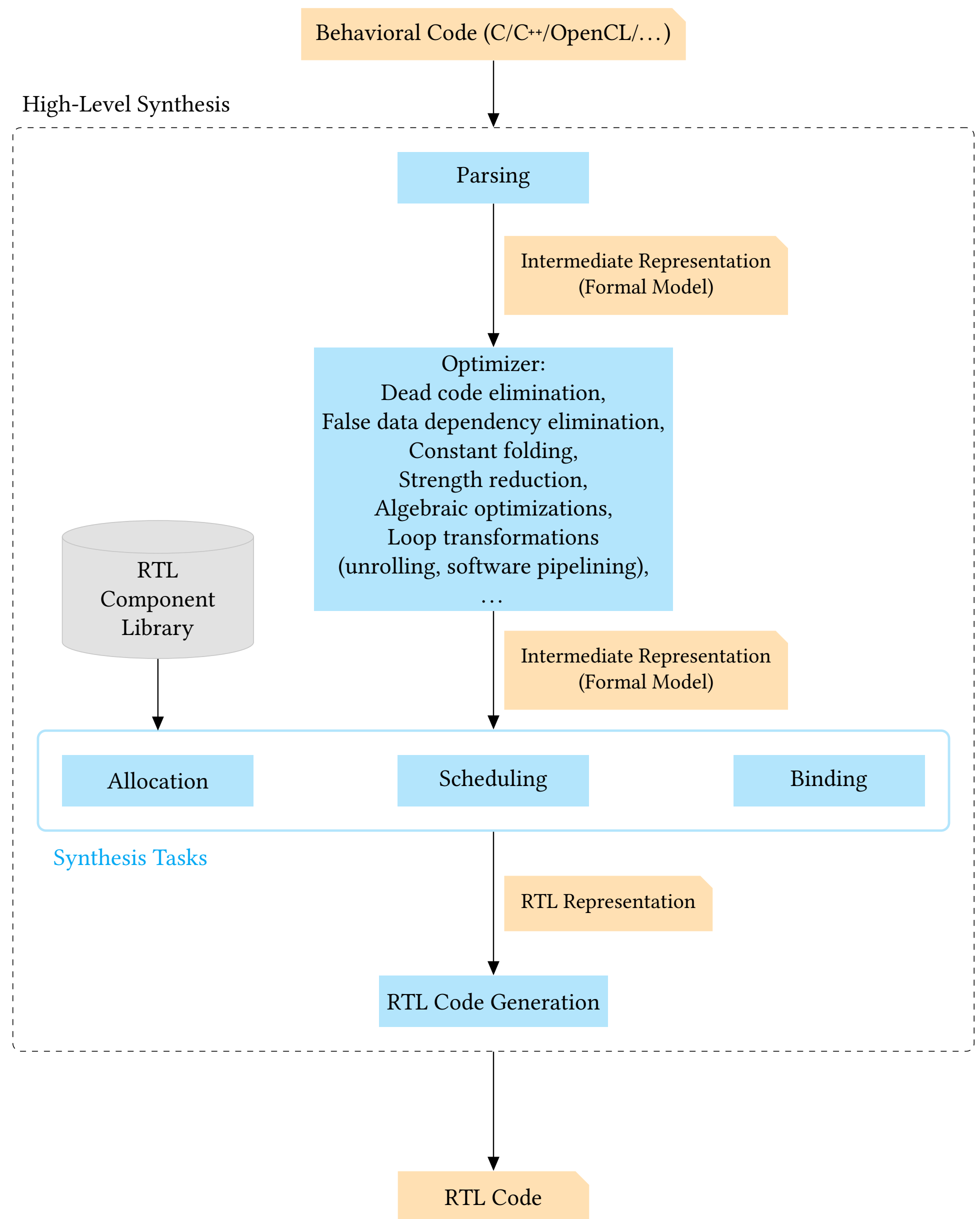


Figure 1.8: Typical HLS tool flow takes as input a behavioral description and generates a fully timed description of a circuit at RTL. The synthesis refers to three major tasks: allocation, scheduling, and binding. The generated code typically describes a control path and a data path in the form of an FSMD.

data dependencies to schedule them into fewer control steps. These methods increase performance by decreasing the total number of control steps, thus, the latency of the synthesized hardware. Multicycling an operation with a high component delay

decreases the time spent for a control step; thus, it improves the achievable clock frequency.

The binding assigns each operation to a specific hardware resource. That is, variables are bound/mapped to storage units (e.g., registers, register files, memory units), operations are assigned to functional units (e.g., adders, multipliers), and data transfers to interconnection units such as multiplexers or data buses. The binding algorithms optimize area usage by resource sharing, where nonoverlapping values/operations or mutually exclusive data transfers are mapped to the same hardware resource. For instance, when possible, using (sharing) the same adder for multiple addition operations without changing the functionality instead of utilizing multiple adders reduces the area cost. Doing so, fewer resources are utilized in the actual hardware (fewer stalls in their execution), and the utilized hardware is used more efficiently. Another goal of binding algorithms is reducing the wiring used in the synthesized hardware. For instance, utilizing regular structures like register files or n-port memories instead of distributed registers reduces the interconnection cost.

Finally, an RTL code is generated from the output of the synthesis tasks (allocation, scheduling, and binding). The generated code typically describes a control path and a data path in the form of an FSMD. The FSMD model is expressive enough to represent both control-dominated and data-dominated circuits [GR94]. The states are mapped to clock cycles in the implementation. The datapath is mainly defined from the output of allocation and binding [WCC09]. The controller FSM is synthesized using the information derived by scheduling and binding steps [WCC09].

Usually, the behavioral input description of an HLS tool is a magnitude of an order shorter than its RTL implementation. In the traditional flow (see Section 1.3), the tasks of allocation, scheduling, and binding must be done by hand. This becomes a complex task very easily. For instance, using multi-cycle pipelined hardware is crucial for implementing floating point operations in hardware to avoid slow clock frequencies or/and large circuits. That is, operations such as additions and multiplication take more than one clock cycle to finish. Even for implementing a few operations, a hardware designer needs to deal with allocating the right amount and type of resources, coming up with an efficient schedule, and mapping the operations to the allocated resources. Such a design uses unnecessary resources unless the arithmetic circuits are shared among different operations. What is more, all of these tasks need to be done again when design constraints are changed, i.e., the designer, most probably, is required to allocate a new set of resources better suited to the task. For instance, high-speed multipliers not using significantly more area resources require more clock cycles to finish the task (i.e., they operate at a higher clock frequency at the cost of a higher latency). Changing the multipliers to accelerate an existing hardware requires repeating the allocation, scheduling, and binding steps, thus the RTL code. A similar scenario in HLS does not require modifications to the behavioral description since the functionality remains the same. Hence, HLS eases

exploring the design space of a hardware implementation, increases the productivity of hardware design, and improves the readability, maintainability, and extensibility of its description in code.

1.4.1 A Brief History of HLS Tools: Analysis of Past Successes and Failures

HLS is still not fully embraced by the community, i.e., neither by algorithm/software developers nor by hardware designers, despite the research interest almost over the last five decades. It is essential to understand the reasons for the failure and success stories of past HLS tools as well as the lessons learned to go one step further. We refer to [NSP+15; CLN+11; NPP+20] for a detailed look at the history and evaluation of HLS tools, and to [CLN+11; MS09; BRS13] for analysis of past failures. Here we provide a brief overview.

Early works in the 1970s have produced pioneering HLS tools, such as CMU design automation system (CMU-DA) [PTS+79]. These tools were already based on hardware synthesis, doing the tasks such as datapath allocation and controller generation, as well as code transformations used in software compilers, such as dead code elimination, common subexpression extraction, and constant propagation. Later in the 1980s and the early 1990s, many fundamental HLS techniques were developed [GR94; Mic94]. These include scheduling algorithms (e.g., [PK89; PPM86]), resource sharing techniques (e.g., [KP87; PK86]), and design space exploration solutions (e.g., [JPP88]). The so-called *layered approach* that separates the tasks of code compilation, synthesis, and code generation (see Figure 1.8) became well-established. Correspondingly, a number of HLS tools have been developed in this era by academia, e.g., ADAM [GKP85], HAL [PK86], MIMOLA [Mar84], Hercules [DKM+90], Hyper-LP [CPR+92], as well as by industry, e.g., Cathedral [DRS+86], Yorktown Silicon Compiler [Cam88], BSSC [YJH+87]. These efforts led to many important innovations that have built the foundations of HLS. However, RTL synthesis was not mature back then. Hence the HLS tools developed in this era could not synthesize circuits that deliver high quality of results (QoR). In the hardware design context, QoR design refers to the ratio between the performance of the circuit (latency, throughput) and design cost (circuit area, energy consumption).

Subsequent to improvements in RTL synthesis, the so-called *second generation* of HLS tools has been built by the industry [CLN+11; MS09]. Major EDA companies offered commercial HLS tools, including Behavioral Compiler [Kna96] from Synopsis, Visual Architect from Cadance, and Monet [Ell99] from Mentor Graphics. Furthermore, semiconductor companies such as IBM, Siemens, Philips, and Motorola built proprietary HLS tools [BOS+95; BKL+93; LvMvdW+91; KCG+98]. The performance of HLS tools was promising, but they failed to be an integral part of the digital hardware

design flow. Users were expected to learn behavioral HDLs to describe their hardware, but they could not rival with handwritten RTL designs. In fact, they provided poor results for control-oriented applications. EDA tools could not support HLS tools, yet good enough, to iterate over the area, power, and performance parameters of final ASICs [CLN+11]. The timing closure problem between logic and physical design only matured in the late 2000s. Very little or no support is given for interface synthesis, validation, and system integration. Despite all these limitations, HLS tools were overpromoted, causing high expectations, thus high disappointments [MS09].

Interest in HLS has been piqued again in the last two decades. The improvements in FPGA technology and the spread of their use have been two of the factors that derive this interest as well as the critical need for energy efficiency gains in the post-Dennard scaling era (see Section 1.2), widespread use/need of heterogeneous SoCs, and the increased design complexity caused from huge silicon capacity. HLS has been seen as a great way to map algorithms into hardware, especially for FPGAs, which requires less validation compared to designing ASICs and is significantly cheaper. In fact, a number of HLS tools solely target FPGAs and generate device-specific code to achieve even a higher performance [CLN+11].

The latest generation of HLS tools seem to have higher commercial success than the previous ones [CLN+11; MS09; BRS13; NSP+15]. These tools aim to attract programmers with very little or no hardware expertise by offering a programming language that is more familiar, mostly based on C or C++. They have become capable enough to provide high-quality results, especially for DSP and datapath-oriented applications. Learned from the past, tool developers are more careful at promoting their tools while focusing on domain-specific applications where HLS are more successful and FPGAs provide great benefits [CLN+11; NSP+15]. Furthermore, they offer system-level integration tools (e.g., Intel OpenCL SDK for FPGAs, Xilinx Vitis) for integrating generated hardware into a heterogeneous system. For instance, existing HLS tools decrease the time spent communicating a hardware implementation via a peripheral component interconnect express (PCIe) interface to an order of minutes for a non-experienced developer from months of work at RTL. They even offer system synthesis to map program parts to either software or hardware. This enables software-like development for library design and verification.

Most of the existing HLS tools now support integrating RTL code descriptions into a higher-level description (often as a black box). This allows using high-detailed low-level descriptions and legacy code as part of a high-level description. Furthermore, existing HLS tools allow users to simulate and verify their code at every level of design (i.e., algorithmic level, RTL, post-synthesis) by automatically generating simulation files from a high-level description. This increases the productivity of hardware system design significantly.

What is more, HLS tools significantly ease design space exploration. This is significantly important because hardware designers expect different design specifications to

result in different hardware implementations (with different area, power, and speed parameters) even when the functional description is the same. For instance, current tools support modifying the description of a hardware architecture for a faster or a slower clock frequency, which mostly requires changing a parameter defined by a pragma. They mostly fail at finding the implementation variations when a different memory implementation or a complex control structure is required. Yet, the required code modifications are significantly less compared to applying similar changes to a hardware description at RTL. It must be noted that support for writing generic code in programming languages such as C++ is significantly better than languages at RTL, e.g., Verilog, VHDL. Therefore, providing generic descriptions that support different design parameters without modifying the code is way easier when an HLS tool is used.

In summary, improvements in EDA technology as well as HLS techniques (to a level that rivals with handwritten implementations), higher motivation to embrace a new design methodology (e.g., caused by the need for energy efficiency, the rise of FPGAs, and increased complexity of huge silicon capacity), focusing on domain-specific applications, offering system-level integration tools along with a familiar language and a mechanism to explore design space have been seen the key features that the latest generation of tools to achieve more commercial success.

1.4.2 Limitations of Existing HLS Tools

Most of the algorithm developers are not familiar with hardware design at all. However, today's HLS tools require a description of hardware behavior (how the circuit works) rather than its functionality (what the circuit does), as shown in Figure 1.9. Even when a program optimized for an ISA (e.g., a CPU) is successfully compiled through current tools, the generated circuits deliver unacceptably bad performance or use too much area, demolishing the benefits of using specialized hardware. Therefore, users have to deal with hardware design tasks such as exploiting the parallelism of an algorithm for its hardware implementation, coming up with an application-specific memory architecture reducing off-chip communication, and minimizing bit-precision in order for HLS tools to synthesize hardware circuits delivering expected performance within an acceptable resource budget. Then, they are expected to describe the hardware implementation of an application using the language abstractions of software (e.g., registers and arrays to specify a memory hierarchy, and loops to describe the execution of a hardware pipeline). This makes the learning curve steep for programmers with no hardware design experience.

Most of the existing HLS tools take as input a behavioral description written in a popular programming language, such as C, C++, or OpenCL. However, they support only a subset of these programming languages. That is, users are not allowed

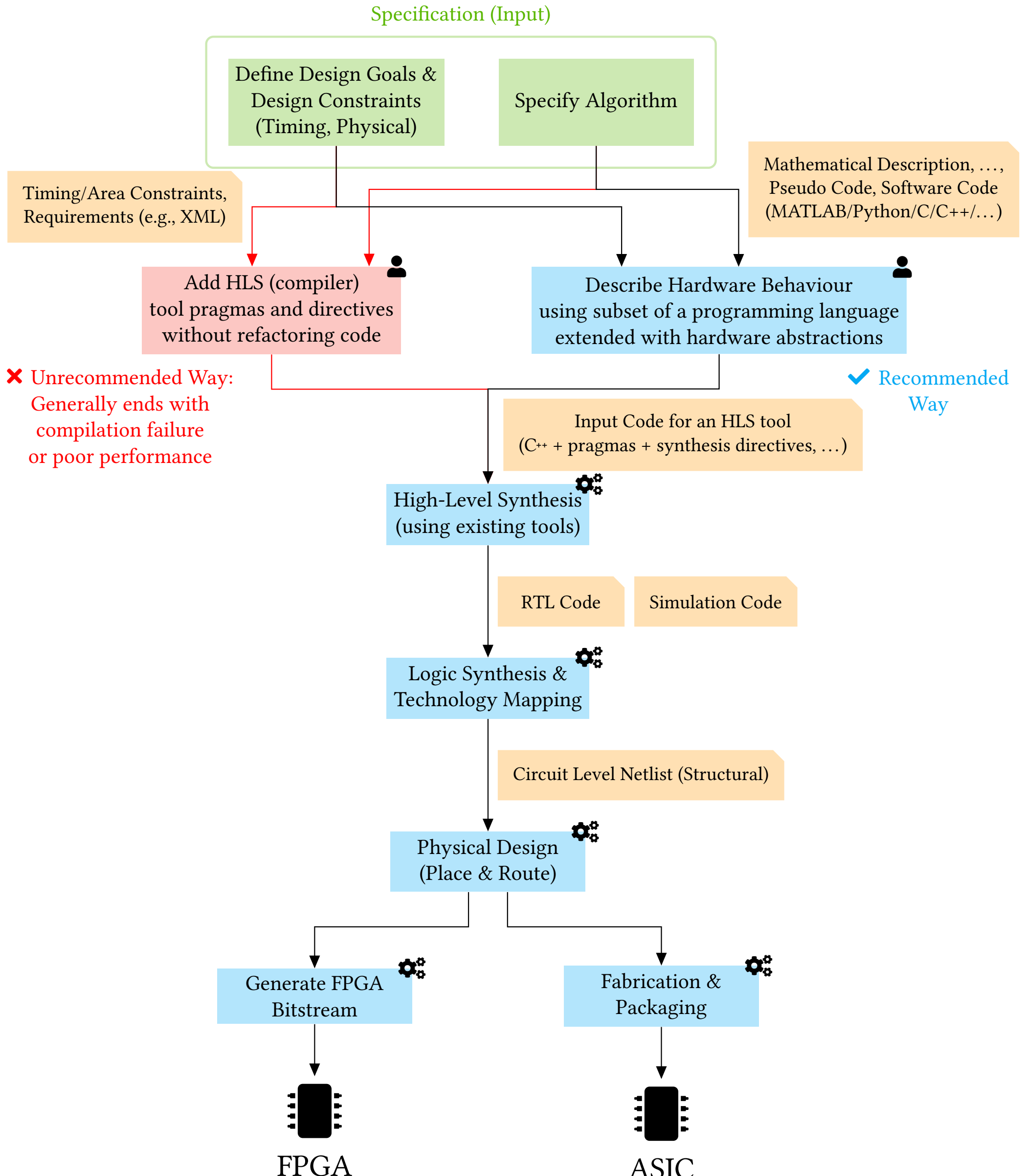


Figure 1.9: Hardware design flow using existing HLS tools. It takes as input the specification of the algorithm, design specifications, and design constraints (denoted by green). Hardware design and synthesis tasks are colored blue, whereas inputs to these tasks are colored beige. The 👤 denotes the manual work of a hardware designer, and ⚙ denotes the tasks automated by synthesis tools.
In order to synthesize circuits providing good quality results in terms of resource usage and performance, users have to describe the behavior of hardware (how the circuit works) rather than its functionality (what the circuit does). Therefore, current HLS tools do not eliminate the need for hardware design knowledge. HLS users can use a common programming language, such as C, C++, or OpenCL, extended with tool-specific language features and pragmas. This mixed-use of hardware and software abstractions makes the input description cumbersome, tool-specific, and not portable.

to use language features such as pointers, dynamic memory allocation, threads, and recursion. HLS tools such as Vivado HLS, LegUp, Bambu, and Catapult-C usually use different C dialects [NSP+15]. They extend the input language with tool-specific language features and/or pragmas since these programming languages are not designed for hardware description (i.e., they are tailored for execution on ISAs). However, pragmas cannot be used in a modular way because the preprocessor already resolves them (e.g., pragmas cannot be passed as function parameters), and they are tool-specific [ÖPM+20a].

Another problem is that syntactic variations affect the performance of HLS. That is, for instance, using *if-else* statements instead of a *switch* statement leads to a completely different hardware implementation even when the same hardware is described[3]. These limitations force users to tune their code for one specific HLS tool. Similarly, an input code tuned for an HLS tool performs poorly even when it is successfully compiled and executed on a CPU or on another ISA. Furthermore, the lack of standardization in HLS languages and compilers hinders the portability of code across them.

1.5 Our Approach: Raising the Abstraction Level in HLS for a Restricted Application Domain

We believe that the next step for HLS requires innovations at the language level. More specifically, the abstraction level in HLS must be raised to the algorithm description (what the circuit does?) from its implementation (how the circuit works?) to deliver the promise of bridging the productivity gap between algorithm development and its hardware design.

As discussed in Section 1.4.2, modern HLS tools offer a familiar language for HLS, but they expect users to write tool-specific and cumbersome code, mixing the programming abstractions of software and hardware descriptions. Writing such a description is only possible by understanding both hardware design principles and software programming languages. That is, even experienced hardware designers have to learn how to describe the behavior of hardware expected by the selected HLS tool. They should consider hardware design techniques and optimizations despite using a software programming language. Further software engineering skills are required to develop a readable, modular, extensible, and maintainable code [ÖRH+17a; dSBL19; EZI+19; RAK18].

[3]Note that syntactic variations affect the performance of RTL synthesis as well. However, the effects on the quality of synthesis are not as significant as it is in HLS. Therefore, it is not seen as a big problem by the hardware design community

Synthesizing *good* hardware circuits (delivering good performance and resource usage trade-offd from a purely behavioral description that contains no information about its implementation has been proven to be a challenging task when a programming language designed for an ISA is used for design entry. Programming languages such as C, C++, OpenCL, CUDA, and OpenMP are designed for a fixed memory and execution model of sequential, multi-threaded, and/or multi- and many-core execution. This means that an algorithm described in a language such as OpenCL is specific to a processor architecture in the first place. Furthermore, to maximize performance, programmers must tune their implementation with low-level optimizations specific to the target device [SFL+15]. Extending one of these languages with hardware description-centric features, thus creating an ambiguous language as design entry, burdens not only users but also HLS compilers. That is, an HLS compiler must transform the execution model of a fixed computer architecture (e.g., memory model of a many-core processor architecture such as GPU) to an application-specific circuit. Furthermore, a specialized software code must be generated for the synthesized application-specific hardware in the case of system generation tools.

We suggest decoupling the description of algorithm behavior from its implementation as shown in Figure 1.10. We avoid having a heroic approach of providing a general-purpose language for design entry. Instead, we advocate providing a set of algorithm-level, declarative, and domain-specific abstractions (in the form of a DSL or a function library) for the description of an application. Then, we use code generation techniques (i.e., source-to-source compilation, metaprogramming, or partial evaluation) to generate a concise description of the hardware behavior. We leverage decades of research on HLS techniques by using a modern HLS tool for synthesizing an RTL circuit from an untimed but application-specific, low-level, target-specific, highly-optimized hardware description. In this way, we use domain knowledge to capture memory-access patterns as well as the intrinsic parallelism from the algorithmic abstractions. Then, we solve the allocation, scheduling, and binding problems at a higher level, e.g., allocating registers, on-chip memory blocks for the described memory hierarchy, parallelization and software pipelining of loops according to latency and throughput constraints.

In this thesis, we focus on the domain of image processing applications as a proof of concept solution since image processing applications are susceptible to leveraging the benefits of designing application-specific circuits. A large portion of image processing applications are computation-intensive algorithms that have high data locality. Often these algorithms apply many operations on the same data through dependent algorithmic steps having temporal locality. Therefore, designing deeply pipelined circuits reducing external memory access allows for achieving higher throughput and lower power consumption.

Our approach has the following main steps:

1. identify performance-relevant abstractions [ÖOQ+21; ÖPM+18]

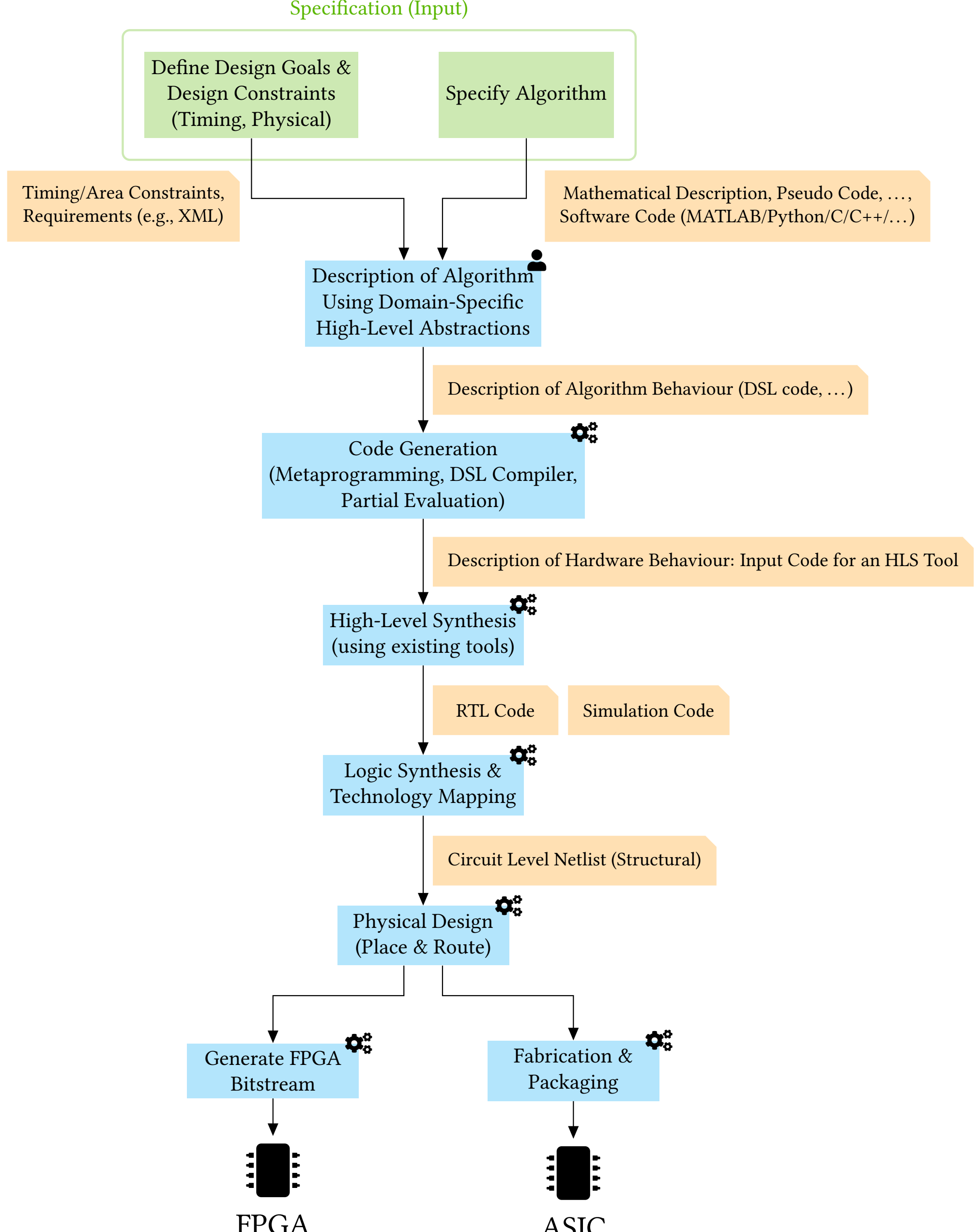


Figure 1.10: The design flow proposed in this thesis, which takes as input the specification of the algorithm, design constraints, and design specifications (denoted by green). Hardware design and synthesis tasks are colored blue, whereas inputs to these tasks are colored beige. the 👤 denotes the manual work of a hardware designer, and ⚙ denotes the tasks automated by synthesis tools.
Unlike existing HLS tools (see Figure 1.9 for their design flow), our approach eliminates the need for describing hardware behavior. It takes a description of the algorithm behavior as input, specified by a set of high-level domain-specific abstractions. Then, a code generation mechanism uses the inherent parallelism information captured from the input application and the domain knowledge (e.g., image processing) to transform user code to highly-optimized input code for HLS.

2. investigate efficient implementation techniques for the considered abstractions and optimize the code at low level [ÖRH+17b; ÖRH+17a; ÖRH+16; ÖPM+18; RÖH+18]
3. develop libraries, DSLs, and/or compilers to increase productivity, modularity, and portability without sacrificing the performance of the generated circuits [ÖRH+16; ÖRH+17a; RÖM+17a; ÖPM+18; ÖPM+20a; ÖOQ+21]

1.6 Contributions

The idea of using a DSL (or a set of high-level abstractions) to eliminate device-specific control flow, and thus, provide productivity, performance, and portability, is not new. Writing so-called *high performance* code is also a challenging and tedious task for modern CPUs as well as domain-specific computing platforms such as GPUs, DSPs, and TPUs [HP19; CCF+10; LTE+20; RBA+13; SMB+16; BRR+19]. DSLs have been seen as a promising solution in this field to circumvent readability, portability, and modularity deficiencies of device-specific program optimizations and eliminate the need for device knowledge to achieve high performance. Examples include SQL, MATLAB as well as the recently built image processing DSLs such as Hipacc [MRH+16], Halide [RBA+13], and PolyMage [MVB15].

However, this thesis includes pioneering work that targets FPGAs by using state-of-the-art HLS tools from a DSL designed for **productivity**, **performance**, and **portability**. As a result, we aim to use the same application description to provide portability of performance across different computing platforms, including FPGAs. Furthermore, we investigate modern code generation techniques such as partial evaluation, metaprogramming, and source-to-source code compilation for this task.

More specifically, the main contributions of this thesis are summarized as follows:

- In Chapter 3, we propose novel hardware implementation techniques for loop coarsening and image border handling [ÖRH+17b]. These are crucial techniques for accelerating stencil-based image processing applications and are also relevant for RTL designers. They require drastic modifications in the control path and memory architecture, as well as in the datapath. Therefore, they are hard to be inferred by HLS compilers unless the input code is written in a so-called *hardware design manner* [ÖRH+16], an approach we introduced for concisely describing hardware for modern HLS tools and provides good QoRs that rivals with RTL descriptions [ÖRH+17b; ÖRH+17a]. This makes them good test cases to show the effectiveness of our approach explained in Section 1.5.
- In Chapter 4, we present a novel source-to-source compiler that generates input code for Intel FPGA SDK for OpenCL from a DSL (embedded into C++)

without sacrificing QoR [ÖRH+16]. To the best of our knowledge, this is the first scientific paper targeting Intel HLS tools by generating OpenCL code from a DSL. It uses several hardware-centric optimizations, including our proposed loop coarsening techniques. We show that our approach can achieve five times higher throughput while using 60 % fewer hardware resources compared to the parallelization intrinsics provided by the Intel compiler. Developing or modifying a source-to-source compiler is a complex task. In order to alleviate the task of our DSL compiler, we used metaprogramming techniques to develop a modular and highly parameterizable function library that allows users to describe image processing algorithms as stream-based data flow graphs [ÖRH+17a]. It is highly optimized with hardware-centric design techniques such as bit-level optimizations, deep pipelining, loop coarsening, and with best practices for Xilinx Vivado HLS. This library has the following novel features compared to previous work, such as Xilinx' OpenCV implementation: (i) Users are not restricted to complete algorithm calls, such as Harris Corner. Instead, our approach provides domain-specific abstractions (similar to Hipacc [MRH+16]) as well as crucial hardware design elements, such as line buffers and sliding window. (ii) It provides multiple implementations for the domain-specific abstractions such that users can select speed or area as an optimization goal.

- In Chapter 5, we demonstrate the benefits of using a DSL-based code generation from OpenVX, a royalty-free industrial standard based on a graph-based execution model, by providing a novel implementation called HipaccVX [ÖOQ+21]. HipaccVX is able to generate highly-optimized target-specific code for FPGAs as well for CPUs and GPUs. Our approach allows accelerating user-defined kernels for the selected computing platform as part of an application graph that includes OpenVX' CV functions, a feature that is not supported in the standard [The19]. It also enables additional optimizations that cannot be applied from a typical OpenVX backend that solely includes implementations of OpenVX' CV functions.

- In Chapter 6, we present AnyHLS, a novel approach to raise the abstraction level in HLS by using partial evaluation as a core compiler technology [ÖPM+20a]. It provides significantly higher productivity for developing higher-order *zero-cost* functions and an unprecedented level of portability across different HLS tools. AnyHLS uses solely *one* language and *one* function library to generate target-specific highly optimized input code for two commercial HLS tools, namely Xilinx Vivado HLS and Intel FPGA SDK for OpenCL. Unlike metaprogramming, it guarantees the well-typedness of the generated program. Furthermore, extending AnyHLS with new functionality does not require modifications to a compiler or a code generator written in a different (host) language. We showed the productivity, modularity, and portability gains by presenting an image

processing library as a case study.

Before introducing our contributions listed above, Chapter 2 briefly gives the background information necessary to understand this thesis and provides a motivational example (in Section 2.3.2) to outline optimization challenges required for C-based HLS tools.

Part I

Mapping Image Processing Algorithms to Hardware

2 Image Processing with Hardware Pipelines

This chapter provides the fundamental knowledge necessary to understand the contributions of this thesis. First, Section 2.1 gives an overview of image processing applications and image processing operators that we use as performance-relevant abstractions in our proposed declarative programming techniques. Then, Section 2.2 briefly explains the background knowledge and necessary optimization techniques for C-based HLS tools. Finally, Section 2.3 present implementation techniques for image processing operators and challenges of using C-based HLS for this task by showing a motivational example.

2.1 Image Processing Applications

Image processing is the manipulation and analysis of images using mathematical algorithms. It can be used to improve the quality of an image, extract useful information, or compress the image for storage or transmission. Image processing is a broad field that encompasses many different techniques and applications, some of which are listed below.

- *Image enhancement* refers to techniques used to improve the visual quality of an image. Examples include adjusting the brightness and contrast, removing noise, and sharpening the edges of objects in the image.

- *Image restoration* refers to techniques used to remove distortions or defects from an image. Examples include removing blur caused by camera shake or atmospheric turbulence and filling in missing pixels caused by image compression or transmission errors.

- *Image analysis* refers to techniques used to extract useful information from an image. Examples include identifying objects or features in an image, measuring their properties, and tracking them over time.

- *Computer vision* uses computers to understand and interpret images and video. Applications include object recognition, facial recognition, and gesture recognition.
- *Photography*, where image processing algorithms are used to improve the visual quality of images captured by a camera. These algorithms include image enhancement, color correction, and white balance to produce a final output image that is ready to be shared or printed.
- *Medical Imaging* techniques like X-ray, CT, MRI, and ultrasound are used to create detailed pictures of the inside of the body. Doctors can then analyze these images to diagnose and treat injuries and illnesses.
- *Autonomous vehicles* rely on image processing to understand and navigate their environment. Cameras mounted on the vehicle capture images of the road, and image processing algorithms are used to detect and track other vehicles, pedestrians, and road signs.
- *Robotics*, where image processing is used to give robots the ability to "see" and understand their environment. Applications include navigation, grasping, and manipulation.
- *Remote sensing*, where image processing is used to extract information from images and videos captured by satellites and drones. Applications include monitoring crop growth, detecting forest fires, and mapping urban areas.
- *Quality control,* where cameras and image processing are used to inspect products on assembly lines, looking for defects such as scratches, dents, or misaligned parts.
- *Biometrics,* where image processing is used to identify and authenticate individuals based on their unique physical characteristics, such as fingerprints, facial features, or iris patterns.
- *Surveillance,* where cameras are used to monitor public spaces for security purposes. Image processing algorithms analyze the video feed in real time, detecting and tracking people and vehicles, and alerting security personnel to suspicious activity.
- *Entertainment,* where special effects and realistic graphics are created in movies and video games. Image processing algorithms are also used in video editing and post-production.
- *Art and design,* where image processing is used to create and manipulate digital images, animations, and videos.

The field of image processing is broad and constantly evolving, where new applications are being developed all the time. The emergence of cheap, low-power cameras and embedded platforms have boosted the use of intelligent systems with computer vision capabilities in a broad spectrum of markets, ranging from consumer electronics, such as mobile, to real-time automotive applications and industrial automation, e.g., semiconductors, pharmaceuticals, and packaging. The global machine vision market size was valued at $16.0 billion already in 2018, and yet, is expected to reach a value of $24.8 billion by 2023 [BCC18]. Many of these systems are associated with stringent requirements regarding performance, energy efficiency, and power, where image processing is used for preprocessing, feature extraction, and/or post-processing operations.

2.1.1 The Power and Computational Requirements

Image processing algorithms are often computation hungry since they mostly consist of multiple stages of computationally intensive mathematical operations (e.g., convolutions at preprocessing followed by matrix multiplications and thresholding at a feature extraction) dealing with large amounts of pixels. Processing a high-resolution image requires applying an input algorithm to thousands of pixels, whereas processing a video with many frames requires processing millions of pixels.

Many image processing systems are required to work in real-time, which refers to capturing an image and processing it immediately in a way that the output is produced as soon as the image is captured. The output is produced with minimal delay, so the image processing system can promptly act upon the information from the image. For instance, in autonomous vehicles, cameras mounted on the vehicle capture images of their environment, and image processing algorithms are used to detect and track other vehicles, pedestrians, and road signs. This system allows the car to safely navigate the road and make decisions based on the information from the image. In medical imaging modalities such as CT, MRI, and PET scans, real-time image processing allows for faster, more accurate diagnoses and improved patient outcomes. In these scenarios, millions of pixels should be processed every second.

Furthermore, power and energy efficiency are essential considerations for image processing systems used in battery-powered devices, such as mobile phones, tablets, and drones. In embedded systems and IoT devices, power and energy efficiency are important to minimize power consumption, reduce heat dissipation, and extend the device's lifetime. In robotics and drones, power is a critical resource to enable the devices to function for a long time, where these devices are often used in remote or hard-to-reach locations. Furthermore, these systems often need to work in real-time.

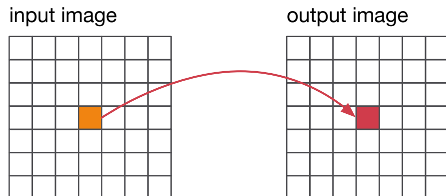
FPGAs are often a good choice for implementing a wide variety of image processing algorithms since they provide a high degree of flexibility and can be reprogrammed to adapt to different image processing tasks. Often traditional processors are not

able to deliver the required power and speed goals of real-time image processing systems. GPUs and DSPs can exploit the data-level parallelism in image processing thanks to their multicore architectures and/or vector units. However, designing specialized circuits allows utilizing only the necessary amount of resources, thus providing better efficiency compared to ISA (as explained in Section 1.2.3). Similar to ASICs, FPGAs allow designing an application-specific memory architecture to reduce off-chip communication, hence providing significantly better power efficiency and speed for an algorithm that deals with millions of pixels every second. Furthermore, designing application-specific deep pipelines consisting of thousands of stages allows for exploiting temporal locality and data locality, thus providing high throughput. Unlike GPUs, FPGA implementations can deliver deterministic latencies and satisfy the reliability requirements of critical applications, for instance, medical devices, military, and space applications.

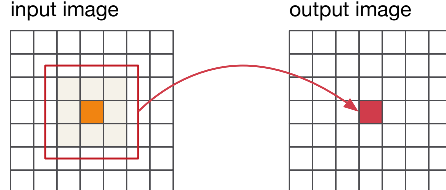
2.1.2 Performance-Relevant Abstractions for Domain-Specific Code Generation

Image processing algorithms can be classified into three categories based on their scope of operation in the spatial domain [Bai11b; Ban08]: point operators, local operators, and global operators (see Figure 2.1).

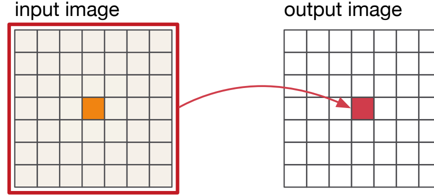
- *Point operators* apply a mathematical function to each pixel independently, without considering the values of neighboring pixels. Examples of point operators include the following:
  - Thresholding: Converting an image to black and white by setting a threshold value and converting all pixels with intensity values above the threshold to white and all pixels below the threshold to black.
  - Brightness adjustment: Increasing or decreasing the intensity of each pixel in the image by a fixed amount.
  - Contrast adjustment: Scaling the intensity of each pixel in the image by a fixed factor.
  - Quantization: Each pixel is mapped to a predefined set of discrete values.
  - Negation: The value of each pixel is inverted.
- *Local operators*, also known as neighborhood operators, apply a specific operation to each pixel based on the values of that pixel and its neighboring pixels. Example applications include:



(a) Point operators produce an output from an input pixel.



(b) Local operators calculate an output from the neighboring pixels in a local window.



(c) Global operators use the whole image as input to calculate an output.

Figure 2.1: Image processing operators can be classified into three categories based on their scope of operation in the spatial domain. Based on these operators, in this thesis, we provide high-level abstractions for the description of image processing algorithms. This approach allows our code generation techniques to capture an application's memory access patterns and generate descriptions of highly-optimized dedicated circuits as input to HLS.

- Image smoothing, the value of each pixel is replaced by the average value of its neighboring pixels, which reduces noise and preserves the overall texture of the image.
- Edge detection: Detecting the boundaries of objects in an image by taking the gradient of the intensity values of the pixels in the neighborhood of each pixel.
- Sharpening filter: Enhancing the edges in an image by subtracting the average intensity of the pixels in the neighborhood of each pixel from the intensity of the target pixel.
- Erosion and dilation: Morphological image processing operations that remove and add pixels to the objects' boundaries, respectively. These are commonly used as part of image segmentation and feature extraction algorithms.
- Median filter: Reduces noise in an image by replacing each pixel with the

median value of its neighboring pixels.

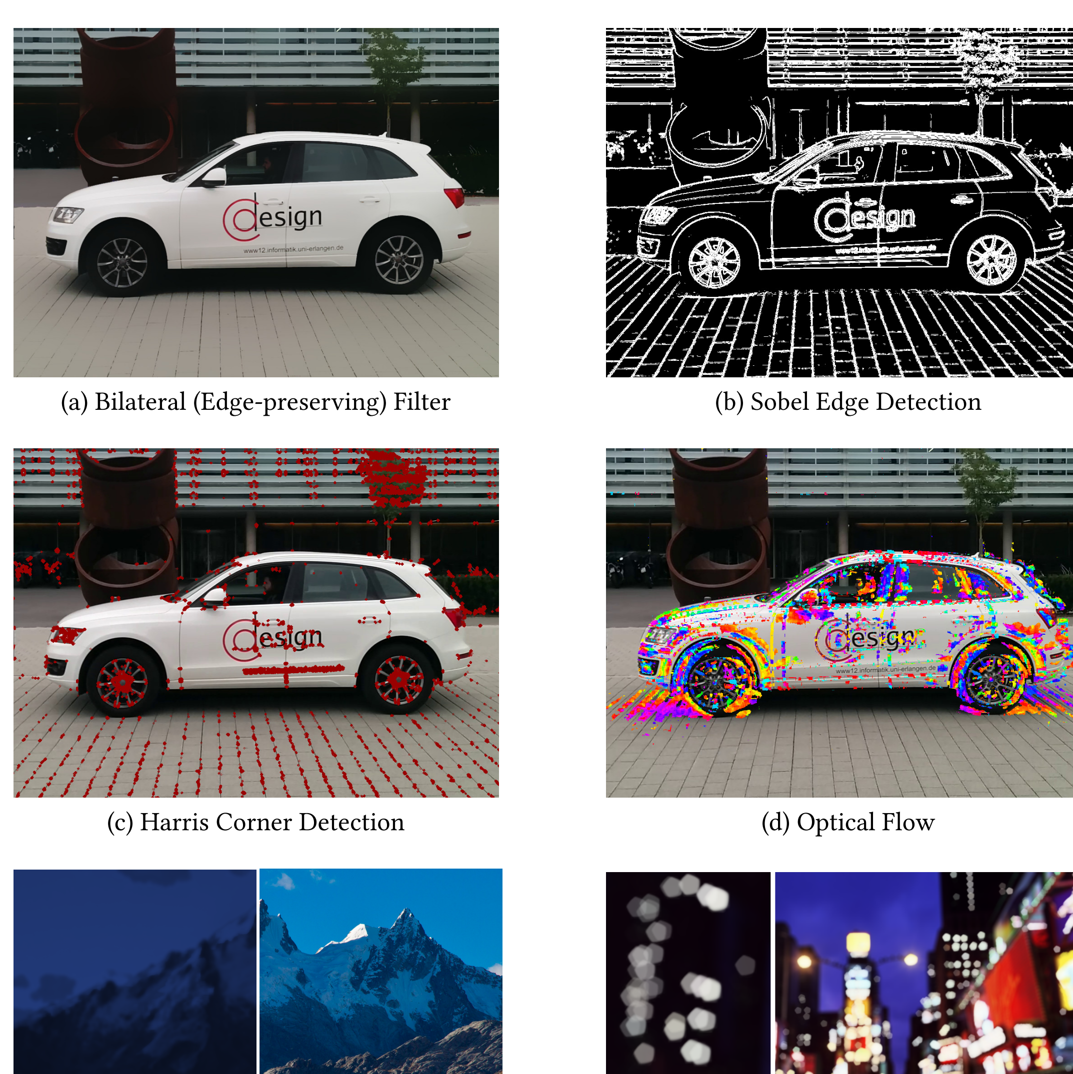
  - Opening and closing: Morphological operations are used to remove small objects or fill small holes in an image.

  Stencil functions are local operators, where the neighborhood (local window) shape defines the *stencil* pattern. Typically, a stencil pattern contains a matrix of values used by the local operator function (also called *stencil function*).

- *Global operators*, also known as spatial-domain operators, are applied to the entire image as a whole rather than to individual pixels or small groups of pixels. Example applications include:

  - Reduction: A category of global operator application that calculates an output value from the whole image, such as finding the maximum or minimum pixel value.
  - Histogram: A global reduction algorithm that creates the frequency of a mapping function calculated from input pixels of an image. For instance, it can be used to analyze an image's overall brightness, contrast, and color distribution.

Existing compilers are not able to deliver high performance for modern ISAs from an input application written by a general-purpose language such as C++ [CCF+10; HP19; RBA+13; SFL+15] unless the input code is manually optimized at low-level. These optimizations are tedious, error-prone, and not portable. Overcoming this limitation requires capturing the inherent parallelism of the input application to optimize memory operations and parallelize selected operations on the target platform. Halide [RBA+13], PolyMage [MVB15], and Hipacc [MRH+16] are modern image processing DSLs that show decoupling an algorithm description from its implementation by providing a set of domain-specific language constructs allows generating target-specific highly-optimized code for various ISAs (such as CPUs and GPUs) from the same input description. These DSLs primarily focus on stencil-based image processing applications, which are power-hungry and are heavily used in many applications from photography to medical imaging, for preprocessing, feature extraction, and post-processing purposes [RBA+13]. Our work follows the same philosophy to raise the abstraction level in HLS for image processing. We provide high-level abstractions to express an application as a dataflow graph of point, local, and global operators. Similar to previous work, we support only global reductions since they consist of many applications with highly different characteristics. Figure 2.2 shows example image processing applications that can be described by design flows presented in this thesis. Identification of these performance-relevant abstractions and the idea of



(a) Bilateral (Edge-preserving) Filter

(b) Sobel Edge Detection

(c) Harris Corner Detection

(d) Optical Flow

(e) Night Filter

(f) Bokeh Effect

Figure 2.2: Example image processing applications that can be described using the frameworks presented in this thesis. A bilateral filter is a *preprocessing* algorithm that smooths an input image while preserving the edges. Sobel and Harris are *feature detection* algorithms that find edges and corners, respectively. The optical flow algorithm detects motion vectors between two input images. Night filter and Bokeh Effect are *post-processing* algorithms used in computational photography. We presented FPGA, CPU, and GPU implementation of these applications at the university booth of the DATE conference in 2019 [ÖRQ+19]. Thereby, the applications are described at a high-level using Hipacc DSL, and the FPGA implementations are automatically generated from the compiler backend explained in Chapter 4.

expressing image processing applications in terms of image processing operators is not new and not one of the main contributions of this thesis. However, we present novel approaches and frameworks that automatically generate highly-optimized hardware implementations by leveraging modern high-level synthesis tools and the parallelism information captured from a declarative application description based on image processing operators.

2.2 Mapping Algorithms to FPGAs using C-based HLS

In this section, we discuss architectural features relevant to HLS and briefly explain fundamental optimization techniques for C-based HLS tools, specifically for Xilinx Vivado HLS and Intel FPGA SDK for OpenCL.

2.2.1 An Introduction to FPGAs

Modern FPGAs comprise logic cells that can be configured to implement user functions and have a fixed number of inputs and outputs. Most of the FPGAs use static random-access memory (SRAM) for programming routing interconnects and logic cells. Typically, logic cells in these FPGAs are composed of a LUT, flip-flop (FF) (also referred to as registers), and a MUX.

A LUT is essentially a memory element programmed at the FPGA configuration time to implement a Boolean function as a truth table. This mimics the functionality of digital circuit design. A typical modern FPGA is built from 6-input LUTs technology [Xil19], which allows implementing $(2^6)^6$ functions for its 6 inputs. More complex Boolean functions are realized by cascading multiple LUTs. Typically, FFs are paired with LUTs. This design decreases the logic delay of consecutive LUTs by storing intermediate results in FFs, a logic design technique known as pipelining. Furthermore, registers and LUTs enable fast-access data storage around an arithmetic datapath.

In addition to the functionality mentioned above, modern FPGAs include a large number of hard blocks that embed common functionality at a higher granularity (word-level instead of bit-level) into the generic silicon blocks for DSP, on-chip memory blocks, off-chip memory controllers and high-speed transceivers. DSPs are optimized at the ASIC level for a limited range of functionality, including multiplier, multiplier–accumulator, barrel shifter, or multipliers. FPGA designers can program the DSP blocks, and accordingly, wire them with the logic cells. Using DSP blocks often reduces the required resources and improves the implementation performance.

FPGAs are not restricted with a cache or a unified memory space. Instead, a fast application-specific memory system can be designed using on-chip memory blocks,

registers, and LUTs. A FF is the smallest storage unit in an FPGA. FPGA synthesis tools can configure several LUTs and registers to implement distributed random-access memory (RAM) that is localized and fast. However, the storage offered by FFs and LUTs is limited. This constraint makes them expensive to use for large data. The on-chip memory blocks provide a larger storage capacity compared to registers. Furthermore, data stored in them can be read and written one clock cycle after the request, which is as fast as a FF. On-chip memory blocks can be configured for parallel access. However, on-chip memory blocks can be accessed via, at maximum (for a typical FPGA), two memory access ports that can be used as either read or write. Therefore, the required number of parallel access directly affects the number of on-chip memory blocks utilized for the same amount of data. Often, the storage space offered by FFs, LUTs, and on-chip memory blocks is insufficient to hold all the intermediate results, thus not entirely eliminating the need for external memory. For this reason, designing an application-specific memory system to reach optimal performance is not an easy problem for non-experts.

Xilinx[1] and Intel[2] are the leading FPGA vendors. While they use different terminologies for describing their FPGA architectures, implementation results reported from their tools can be interpreted similarly: Logic cells of Xilinx FPGAs reside in *configurable logic blocks (CLBs)*, where each CLB contains multiple (typically two or four) *slices*. These slices include a fixed number of logic cells, thus LUTs, FFs, MUXs, and latches. Intel uses the term logic array block (LAB) for a CLB and reports the number of used logic elements (LEs) instead of slices. Xilinx refers to on-chip memory blocks as block random access memory (BRAM)[3] while Intel refers to them as M10K, M20K memory blocks according to their storage capacity. Finally, Intel synthesis results inform the number of used adaptive logic modules (ALMs), which are used to implement 6-input or 8-input LUTs in their technology. We will mainly use Xilinx' terminology wherever it is correct to use.

2.2.2 Area and Speed Considerations in FPGA Design

An FPGA has a limited number of programmable logic blocks and connections, and a limited amount of memory. These resources are used by a synthesis tool to implement hardware.

From the design perspective, FPGAs can be seen to have two types of resources:

i) *Memory resources*: Data is mainly stored in registers and BRAMs whereas LUTs can be used to implement a distributed memory. FPGA designers should be

[1] acquired by AMD
[2] formerly known as Altera
[3] Xilinx also introduced UltraRAMs with UltraScale+ FPGA series. These on-chip memory blocks have a fixed configuration of 72 bits wide and 4,096 bits deep.

careful not to waste a significant amount of LUTs or registers instead of a small number of BRAMs to store data.

ii) *Computational resources*: Implementing an arithmetic operation or a boolean logic mainly requires LUTs and DSPs. Additionally, registers are partitioned around an arithmetic datapath for highly parallelized computation and pipelining. As a good practice, DSPs should be preferred over LUTs for implementing expensive operations such as multiplication.

The following three are the primary metrics measuring the performance of a digital circuit:

i) *Maximum Achievable Speed (Clock Frequency)*: The maximum achievable frequency of a circuit is determined by many factors, including the propagation delay of the longest combinatorial logic path between two clocked elements (*critical path*), the technological characteristics (such as setup hold times of FF), the performance of input/output (IO) interfaces, power supply noise, and operating temperature. Often the speed of a circuit refers to the maximum operating clock frequency that a circuit can reliably operate, typically in a range from a few MHz to several GHz. FPGA vendors inform the maximum clock frequency of a device. However, the speed of a circuit heavily depends on the input hardware description and the optimizations reducing critical path (e.g., pipelining) [4]

ii) *Latency*: In FPGA design and HLS, latency refers to the total number of clock cycles required for producing a result from an input. The term *Overall Latency* (of a loop or an application) often refers to the total number of clock cycles required for processing all the inputs.

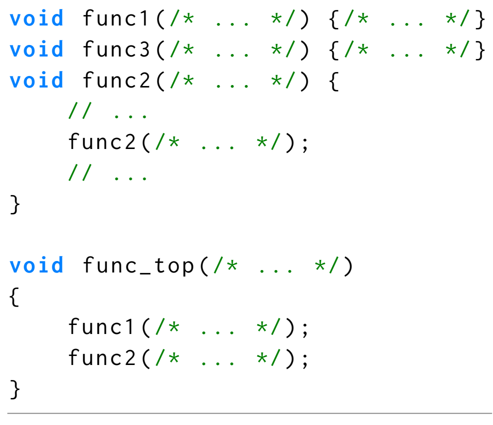
iii) *Throughput*: The number of data processed (or tasks finished, operations performed) within a time unit. Throughput is often measured in terms of samples (or bits) per second (or in a clock cycle). Often, the initial delay spent to produce the first output (called *initial latency*) is ignored when the throughput of a circuit is measured.

In image and video processing (and many applications where FPGAs are favorable), throughput is important since thousands to millions of pixels are processed every second. The clock frequency is another key factor that linearly affects the processing time.

[4]Hardware synthesis tools estimate the delay of the critical path and the clock uncertainty to calculate the minimum period that would reliably drive a sequential circuit without creating metastability issues. HLS users can simply analyze the static timing analysis of RTL synthesis and validate the final circuit with post-place and route timing results to see if the generated hardware satisfies the input timing constraints.

Table 2.1: Typical mapping of C-based language constructs to RTL.

C/C++ Constructs	OpenCL	RTL Mapping
Functions	Kernels	HDL Modules
Func. Arguments	Kern. Arguments	IO Ports
Operators	Operators	Functional Units
Scalars	Scalars	Wires or Registers
Arrays	Arrays	Memories
Control Flows	Control Flows	Control Logics



```
void func1(/* ... */) {/* ... */}
void func3(/* ... */) {/* ... */}
void func2(/* ... */) {
    // ...
    func2(/* ... */);
    // ...
}

void func_top(/* ... */)
{
    func1(/* ... */);
    func2(/* ... */);
}
```

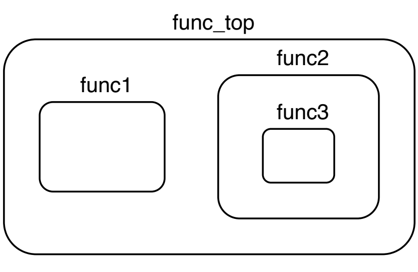


Figure 2.3: C/C++ functions and OpenCL kernels are mapped to RTL modules unless inlined. HLS tools create a hierarchy of these functions/kernels according to their call hierarchy.

2.2.3 Writing Software for C-based HLS

This section briefly overviews key optimization strategies required for describing hardware circuits that deliver good synthesis results by using modern C-based HLS. In this thesis, we use Xilinx Vivado HLS and Intel OpenCL SDK for FPGA, whereas alternative commercial/academic C-based HLS tools benefit from similar optimizations. We refer to [NPP+20] for a survey of these HLS tools.

As summarized in Table 2.1, HLS tools generate RTL modules from C functions and OpenCL kernels. Unless the functions are inlined, the function call hierarchy is sustained in the generated circuit description, as shown in Figure 2.3. Inlining can explicitly be requested by using C language syntax or HLS tool pragmas, but the HLS compiler will make the final choice. Xilinx Vivado HLS requires users to define one top function that calls all the other accelerator functions, whereas this is not a requirement for Intel FPGA SDK for OpenCL.

Function arguments are interpreted as input or output parameters of the generated

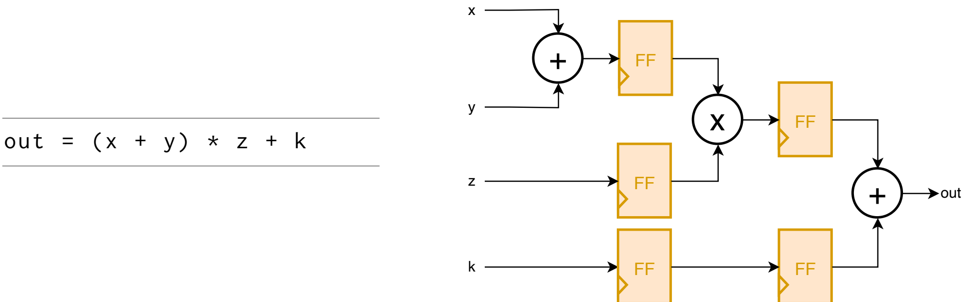


Figure 2.4: Data path circuits are mostly generated from expressions consisting of arithmetic operations. Users do not have to deal with low-level timing and scheduling problems. The HLS compiler decides the degree of registering according to timing constraints, e.g., circuits doing arithmetic operations can further be pipelined to achieve higher speed. Correspondingly, a control path is created to schedule the operations according to the input program.

circuits. Users can define the interface communication protocol (such as AXI, AXI-Lite, or bit vector) by writing tool-specific pragmas. When required, HLS tools automatically add control signals to the interface, such as clock and reset. HLS tools create a data path and a control path for the input description after scheduling, allocation, and binding steps (see Section 1.4 for more details). Data path circuits are mostly generated from arithmetic operations, as illustrated in Figure 2.4. HLS users do not have to deal with low-level timing and scheduling details. The degree of registers is automatically changed according to a given timing target. For instance, HLS compilers can utilize arithmetic circuits with several pipeline stages for a higher speed target. In this case, a control path for the new schedule would be generated automatically. For RTL designers, the same task would require manual scheduling of the new circuit and severe modifications in the HDL code.

Scalars (constants and variables) at the input program are implemented by wires and registers. Arrays are mapped to memory elements. Intel FPGA SDK allows using external memory (DDR Memory) on the FPGA board using OpenCL's *external* memory. Xilinx Vivado HLS allows reading global arrays only by using a function interface, which does not always represent external memory. HLS compilers map OpenCL's *local* memory and C arrays with automatic storage class to on-chip memory resources, which can be LUTs, registers, or on-chip memory blocks unless explicitly defined by tool-specific pragmas. Often, the large arrays are implemented with on-chip memory blocks, not to consume registers and LUTs for storing large data.

HLS users are encouraged to write loop iterations (e.g., using `for`, `while`) with constant bounds to describe the execution behavior of a circuit. Writing all the functionality of a circuit within a main loop allows HLS tools to analyze the latency

of the whole function and data dependencies between functions. The number of loop iterations represents the amount of data and amount of operations for the described task. For instance, the following code tells the HLS compiler to generate a circuit executing 100 iterations, multiplying the data stored on `a` and `b` arrays.

```
void func(/* ... */)
{
    // ..
    for(int i = 0; i < 100; ++i)
    {
        c[i] = a[i] * b[i]
        // ..
    }
}
```

The latency of the generated circuit depends on the scheduling of the HLS compiler. The FSM at the generated control path runs as long as a clock signal is supplied, where a typical implementation reads and produces meaningful data according to *valid* and *done* signals at the interface.

2.2.4 Optimizing Software for C-based HLS

This section briefly summarizes the critical optimization strategies crucial to achieving high-quality synthesis results using current HLS tools.

Increasing Spatial (Data-Level) Parallelism

Designing custom hardware allows exploiting the data locality of an algorithm by replicating a functional unit multiple times to execute multiple operations in parallel. For instance, Figure 2.5 shows 3 alternative hardware implementations for the same expression, including 9 addition and 1 multiplication, where the multiplication has to be applied after the additions because of data dependency. The *HW-3* exploits the DLP of the summation expression by utilizing 9 adders in parallel to calculate the result in 1 clock cycle. The improvement in latency comes at the cost of a higher resource usage.

HLS tools generate sequential circuits for loop iterations. However, they offer loop unrolling pragmas, where users can define an unrolling factor to specify the level of spatial parallelism. For instance, *HW-1* in Figure 2.5 consists of a sequential loop for the additions, whereas this loop is fully unrolled in *HW-3*. Restructuring the input code by manually unrolling a loop describes the parallelized functional units at the input program more concisely compared to using tool-specific loop unrolling pragmas.

Increasing the spatial parallelism of a hardware implementation should be thought of as designing a circuit with a different area/performance trade-off (i.e., replicating

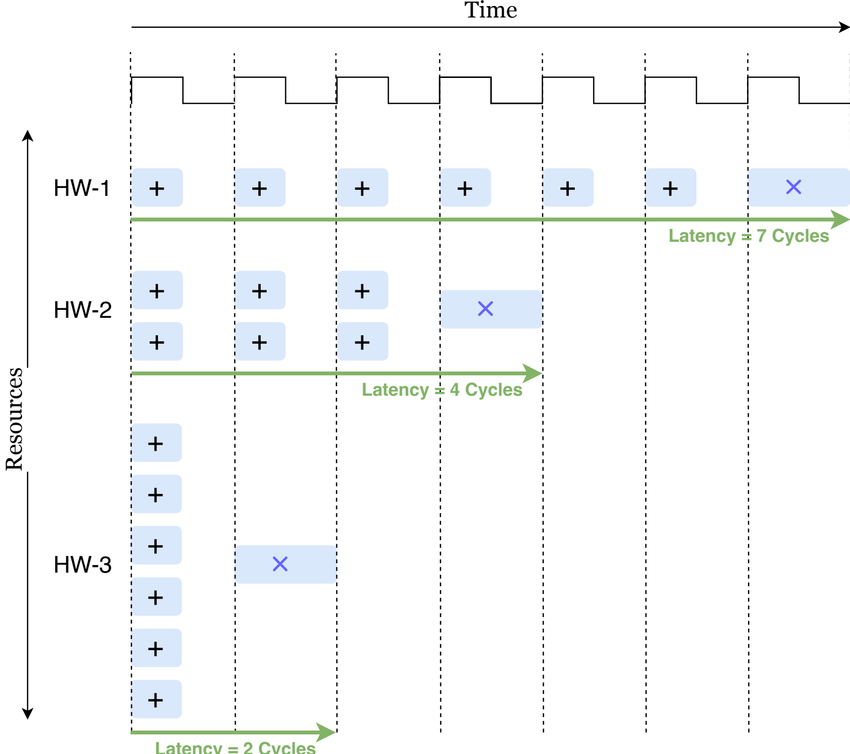


Figure 2.5: Utilizing several functional units in parallel to exploit data-level parallelism (DLP) improves the latency at the cost of using more resources. *HW-1* uses 1 adder to execute 9 additions sequentially while *HW-2* and *HW-3* use 2 and 9 adders, respectively. HLS tools generate sequential circuits for loops, where unrolling the loops means utilizing parallel resources for the body function.

a hardware accelerator to increase throughput increases resource usage). However, exploiting the resource sharing between replicated hardware accelerators could significantly decrease the resource usage cost of DLP, as presented in Chapter 3. Furthermore, in most applications, an FPGA implementation can achieve a throughput close to the memory bandwidth of modern external memories (e.g., double data rate synchronous dynamic random-access memory (DDR SDRAM)) only by processing multiple inputs simultaneously due to current FPGA's low logic speed limitations.

For this purpose, the DLP of the hardware implementation must be increased (to the compute-bound at best) according to the Roofline model [WWP09]. However, this parallelization can be applied to the resource budget.

Pipelined Scheduling

Temporal locality eliminates the data communication overhead between different processor cores, and also introduces a different type of parallelism called *pipelining* for the algorithms composed of sequentially dependent functional blocks.

Instruction-Level (Structural) Pipelining The clock speed of a hardware implementation is limited by the slowest combinational path, which can be divided into stages via registers until the desired logic speed is reached. This technique is called *pipelining* and mostly improves the throughput despite the increased latency in terms of clock cycles. For instance, the operations (denoted as *Op*) in Figure 2.6 represent smaller combinational circuits, similar to the instructions of a software program. Structural pipelining is applied in Figure 2.6b to divide the combinational circuit in Figure 2.6a into 3-stages, thus the critical path is decreased to *Op-2*'s propagation delay.

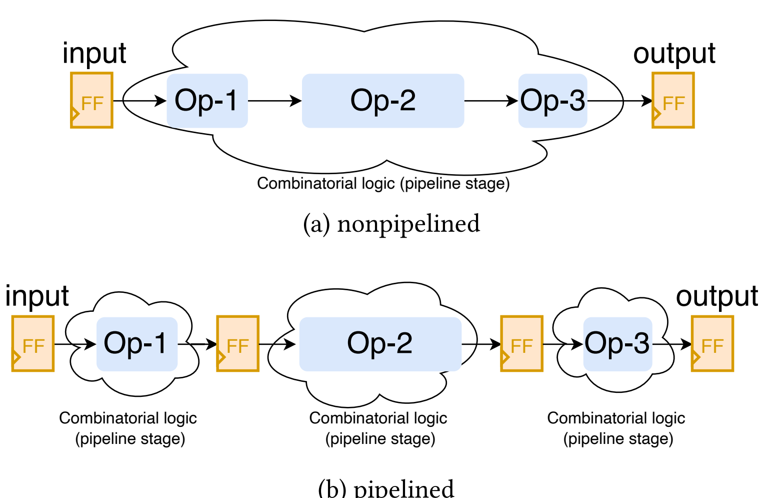


Figure 2.6: Pipelined schedule

HLS tools automatically apply structural pipelining for a given clock speed target. For instance, the multiplier block in Figure 2.5 has a considerably larger propagation delay than adders. When run with a higher speed target, HLS compiler would automatically pipeline this multiplier circuit to multiple stages (or replace it with a faster multiplier block). RTL developers have to describe structural pipelining by hand, thus needing to rewrite their circuit description when the speed target is changed.

Loop (Instruction-level) Pipelining An algorithm can be pipelined at the functional level, similar to structural pipelining. Thereby, operations of a function (or

an iteration block of a loop) are divided into stages and overlapped according to functional dependencies. Consequent stages process the previous stage's output, allowing hardware to read a new input even before the previous result is calculated. In this way, a fixed throughput is achieved after the first output is produced, although the latency of an individual functional block increases or stays the same. For instance, the iteration block in Figure 2.7a is structurally pipelined to 3 stages in Figure 2.7b. The next iteration can be calculated 1 clock cycle after the previous one has started (e.g., *iteration 1* can start 1 cycle after *iteration 0*). In this way, a new result is produced in every cycle after an initial latency. Another example is given in Figure 2.8, where functions of an algorithm are pipelined according to their dependencies. Often, HLS tools provide a special pragma (e.g., dataflow pragma in Xilinx Vivado HLS) to apply this optimization.

Subsequent iterations of a functional pipeline can produce correct results only if all the dependency data is available for the subsequent calculation. This is indicated by the term initiation interval (II), which shows the longest waiting time between any of two subsequent iterations to acquire correct results. For instance, II equals to 1 clock cycle in Figure 2.7b but would be 3 if *iteration-2* depends on the results of *iteration-1* (i.e., if *Op-1* would depend on the result of *Op-3* calculated in the previous iteration).

The latency L of a functional pipeline can be formulated as below, where $N_{iteration}$ denotes the number of required iterations to process all inputs:

$$L = L_{\text{initial}} + L_{\text{process}} = L_{initial} + II \cdot N_{iteration} \text{ cycles} \tag{2.1}$$

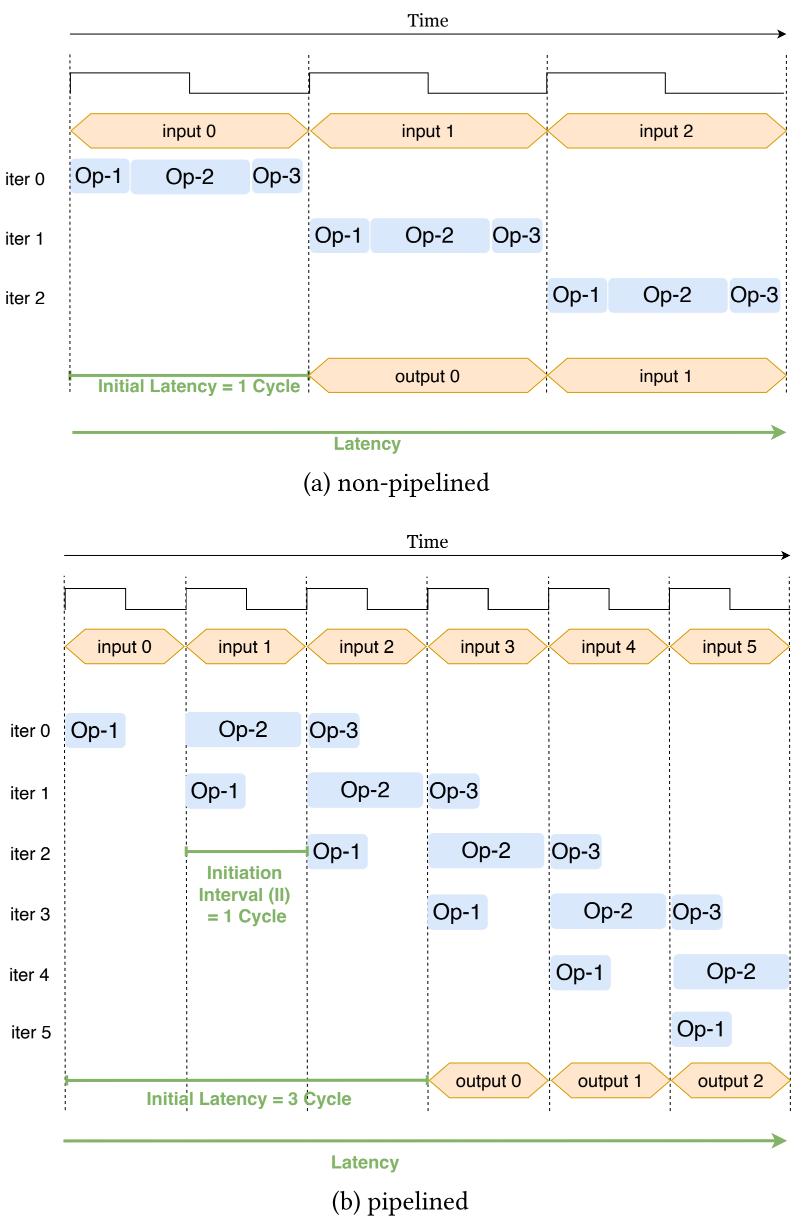
Memory design

As explained in Sections 1.2.2 and 1.2.3, designing an application-specific on-chip memory is crucial to reduce off-chip communication and achieving high performance. Similar to RTL designers, HLS users have to describe a caching mechanism using arrays and registers in the input program. The tools provide pragmas for mapping arrays to specific resources, such as on-chip memory blocks.

HLS tools often fail to pipeline loops unless the arrays that are read and written are not restructured to prevent *loop-carried dependencies*, a term used for functional dependencies between loop iterations. Users can reshape, merge, and partition arrays using tool-specific pragmas (or by hand) to explicitly describe the bitwidth, size, and number of ports on an on-chip memory block.

Stream Processing

Inter-kernel dependencies of an algorithm can mostly be accessed on the fly in combination with fine-granular communication (i.e., once data is produced, the next block can consume) so that the whole FPGA implementation can be pipelined to



(a) non-pipelined

(b) pipelined

Figure 2.7: Loop pipelining increases the throughput by dividing the operations of a loop into stages and overlapping their executions according to their functional dependence. The initiation interval (*II*) refers to the number of clock cycles between two successive loop iterations.

have a fixed throughput. In the best case on a per-datum basis (single register or small buffer) instead of reading or writing the whole intermediate images as shown in Figure 2.9. In this case, off-chip communication can be reduced, and the whole

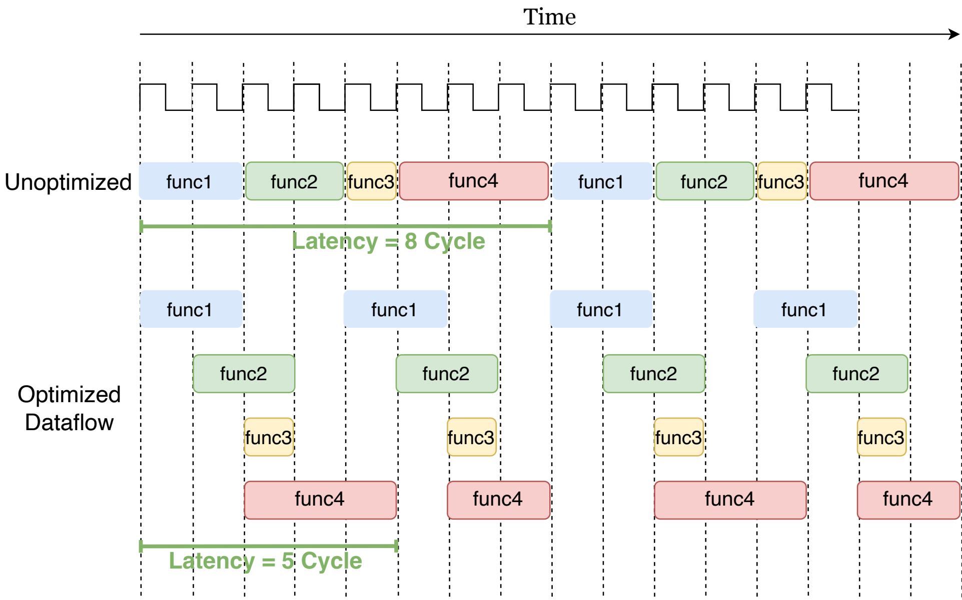


Figure 2.8: An example of task-level pipelining at the function level (also known as dataflow optimization). Executions of the function are overlapped according to their dependencies.

latency can be decreased significantly.

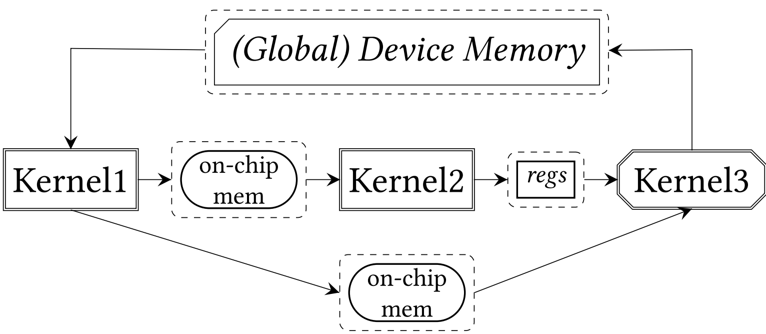


Figure 2.9: FPGA implementations can reduce off-chip communication with their reconfigurable application-tailored on-chip memory. Stream processing exploits this for data transmission between operators.

This is most effective when the whole pipeline has a constant reading and processing speed for every input. Yet, such an implementation has two challenges: (1) determination of the memory types and sizes of the buffers between the kernels, (2) synchronization between kernels. The design of such a system needs integer programming [HBD+14] for calculating the dependency pixels' size and designing a generic and lightweight interface. HLS tools provide special data structures (i.e.,

`stream` in Xilinx Vivado HLS and `channel` in Intel FPGA SDK for OpenCL) for this purpose.

2.3 The Challenge of Designing Image Processing Circuits with C-based HLS Tools

In this section, we briefly explain the hardware implementation of image processing operators, the implementation of a local operator, and finally provide an overview of the approach developed in this thesis.

2.3.1 Hardware Implementation of Image Processing Operators

Image processing applications are favorable to FPGA implementations as they have high spatial locality. Pixels stored in external memory can be accessed in burst mode by reading and writing images in raster order as shown in Listing 2.1. The nested for loop above scans the two-dimensional input image of size `width` × `height` across each row of pixels from left to right and from top to bottom, where an output pixel is calculated for every input pixel by using the function `f`. Pipelining these loops allows exploiting the temporal locality of the image processing operators, at best producing a result in every iteration where II = 1 clock cycle. On a modern computing platform, external memory access requires 130x to 260x more energy than accessing to a large SRAM scratchpad, as discussed in Section 1.2.2. Hence, decreasing external memory usage by designing an on-chip memory structure is crucial to leverage the benefits of designing custom hardware. Similarly, data between image processing operators should be transferred using HLS streams (which hold the data in on-chip memory resources as shown in Figure 2.9), and the size of these streams must be minimized for efficient use of memory resources.

Listing 2.1: Raster order scan of an image processing operator

```
1 for (size_t row = 0; row < height ; row ++) {
2     for (size_t col = 0; col < width ; col ++) {
3         output_image[row][col] = f(input_image[row][col]);
4     }
5 }
```

The DLP of the implementation can be increased by unrolling (coarsening) the deepest-level horizontal scan loop by a factor v similar to vectorization such that multiple consecutive v datum is packed as a *data-beat* and processed in parallel,

as shown in Listing 2.2. Accessing DDR memory through *data-beats* (increasing the bit width of read/write requests) mostly improves the bandwidth rate between programmable logic and accelerator device (depending on the interface and v). Hence, the memory access remains in burst mode, and the throughput is increased by v as follows:

$$Throughput = v/II \text{ pixels/cycle} \tag{2.2}$$

This loop coarsening transformation is often subject to further optimizations, such as utilizing fewer resources for data paths with subexpression optimizations. However, it requires a careful modification of the on-chip memory architecture to prevent creating loop-carried dependencies. Chapter 3 presents novel implementation techniques for loop coarsening of stencil-based image processing applications.

Listing 2.2: Coarsening horizontal scan loop of an image processing algorithm by a factor of v to increase DLP (width is a multiple of v)

```
1 for (size_t row = 0; row < height ; ++row) {
2     for (size_t col = 0; col < width ; col += v) {
3         output_image[row][col] = f(input_image[row][col]);
4         output_image[row][col + 1] = f(input_image[row][col + 1]);
5         // ...
6         output_image[row][col + v - 1] = f(input_image[row][col + v - 1]);
7     }
8 }
```

Point operators (e.g., image scaling and color transformation) calculate an output result pixel for every pixel of the input image (see Figure 2.1). Therefore, a special on-chip memory architecture is not needed. Correspondingly, the total latency L is given as below, where L_{arith} is the latency of the datapath and $N_{process}$ is the image size:

$$L = L_{\text{initial}} + L_{\text{process}} = L_{arith} + (II \cdot N_{process}) \text{ cycles} \tag{2.3}$$

Reading and writing from streams, pipelining the raster order scan, and unrolling the column iteration by v provide a throughput of v pixels per cycle. Typical optimizations such as bit precision, subcommon expression reduction, strength reduction, and reducing the number of expensive operations (such as multiplications) reduce resource usage.

Local operators, such as Gaussian smooth and Sobel filtering (edge detection), calculate an output using the neighboring pixels in a local window. For instance, a convolution description is shown in Listing 2.3[5].

[5]A local operator depends on the pixels outside the image at the borders. The `border_handling` in Listings 2.3 and 2.4 prevents out-of-border access by one of the well-known algorithms (e.g., mirroring). Chapter 3 explains these techniques in detail.

Listing 2.3: Sequential implementation of a convolution function.

```
1  // input/output  images: arr, out
2  // coefficients: mask
3
4  for(int y = 0; y < IMAGE_HEIGHT,  y++)
5      for(int x = 0; x < IMAGE_WIDTH, x++)
6          for(int j = 0; j < MASK_HEIGHT; j++)
7              for(int i = 0; i < MASK_WIDTH; i++)
8                  out[j][i] = mask[j][i] * border_handling(arr, x, y, i, j);
```

A local operator similar to Listing 2.3 requires reading the neighboring pixels of the window MASK_WIDTH $\times$ MASK_HEIGHT for every input pixel (where the input image size is IMAGE_WIDTH $\times$ IMAGE_HEIGHT. This costs hundreds of clock cycles delay if the access is from an external memory. However, applying a local operator to two adjacent windows of size MASK_WIDTH $\times$ MASK_HEIGHT requires the same (MASK_WIDTH$-$1) $\cdot$ MASK_HEIGHT pixels to calculate results at the image coordinates (x, y) and $(x + 1, y)$. This locality can be exploited by processing the image in raster order for burst mode and processing the next pixels sequentially, thus holding the overlapping (MASK_WIDTH$-$1) $\cdot$ MASK_HEIGHT pixels on registers with a sliding window that only reads MASK_HEIGHT new pixels in every shift, as shown in Figure 2.10. In this way, a throughput of one pixel per II cycle can be achieved at the cost of an initial latency (L_{initial}), spent for caching neighboring pixels of the first window (N_{initial}). The total latency of this implementation is given below:

$$L = L_{\text{initial}} + L_{\text{process}} = L_{\text{arith}} + II \cdot (N_{\text{initial}} + N_{\text{process}}) \text{ cycles} \tag{2.4}$$

Literature provides many *global operators* with highly different memory access patterns. We refer to [Bai11a; KMN18] for their hardware implementations. In this thesis, we develop methods that support global reduction. However, our approach can be extended for other global operators. Hardware implementation of a *reduction* is very similar to point operators, where a streaming pipelined implementation without a special memory design provides good performance. Intermediate results are stored in on-chip memory and updated in every iteration. Unlike point operators, the output of a reduction is updated only when the whole image is traversed.

2.3.2 Motivational Example

This section presents a motivational example to show that hardware design knowledge is crucial for using C-based HLS tools. The *unoptimized* implementation in Figure 2.11 shows the resource usage and throughput results of the convolution function shown in Listing 2.3. Listing 2.4 optimizes the convolution in Listing 2.3, solely by using Xilinx Vivado HLS pragmas. It is shown in Figure 2.11 that these optimizations

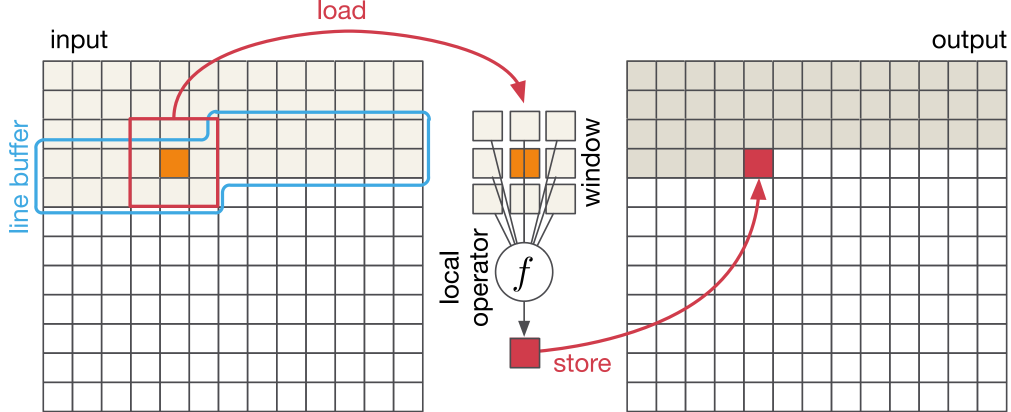


Figure 2.10: Line-buffered pipeline implementation of a local operator reduces the external memory read by storing dependency pixels of a local neighborhood in on-chip memory. The image is read in raster order. Line buffers contain the dependency pixels in on-chip memory blocks and act like multiple first in first out (FIFO) buffers, where line buffers read a new pixel, and sliding window reads a new column in every iteration. The sliding window holds the local window in registers and thus provides parallel access for the local operator function. Ultimately, exploiting the raster scan's temporal locality, a throughput of 1 pixel per II cycles is provided. (Figure reprinted from [RSH+14], © 2014 IEEE)

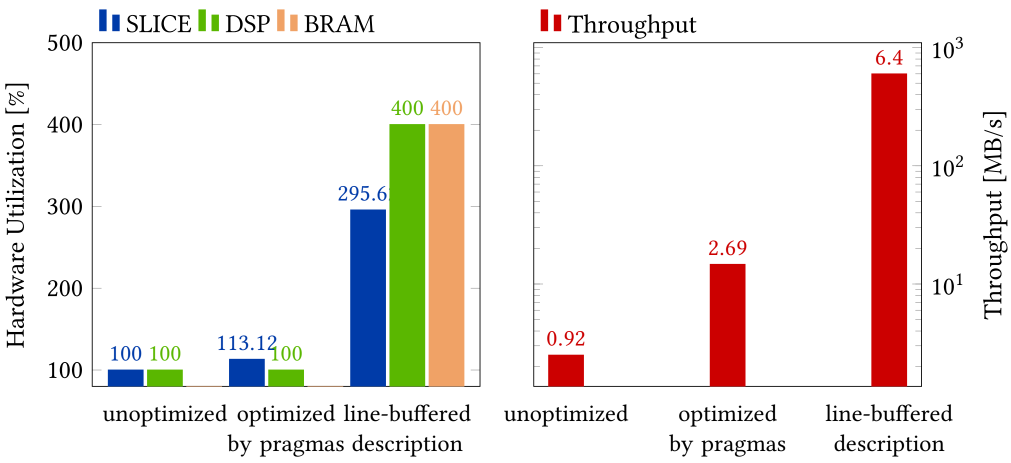


Figure 2.11: Different implementations using Xilinx Vivado HLS

increase the throughput by 29.2×, at the cost of almost 13% more resource usage. However, describing a line-buffered pipeline implementation (similar to Listing 2.5)

Listing 2.4: Optimization of a convolution description using pragmas for Vivado HLS.

```
1  // input/output  images: arr, out
2  // coefficients: mask
3
4  for(int y = 0; y < IMAGE_HEIGHT,  y++)
5      for(int x = 0; x < IMAGE_WIDTH, x++)
6          #pragma HLS pipeline II=1
7          for(int j = 0; j < MASK_HEIGHT; j++)
8              #pragma HLS unroll factor=MASK_HEIGHT
9              for(int i = 0; i < MASK_WIDTH; i++)
10                 #pragma HLS unroll factor=MASK_WIDTH
11                 out[j][i] = mask[j][i] * border_handling(arr, x, y, i, j);
```

increases the throughput by 237.9%, at the cost of 182.5% more FPGA slices. The line-buffered implementation uses on-chip memories (i.e., 4 BRAMs for a 5-by-5 window), whereas the other versions do not leverage their benefits.

Listing 2.5: Line-buffered pipelined implementation of a convolution function. The synthesized circuit provides 1 pixel per clock cycle throughput, holding dependency pixels on an application-specific memory hierarchy and pipelining the algorithm. It exploits the locality of the convolution algorithm at the cost of writing non-portable, tedious C++ code optimized for Vivado HLS. Only the lines 72 to 77 describe iteration over the local window (behavior of the algorithm), which only corresponds to approx. 6 % of the whole implementation.

```
1  // input/output images: arr, out
2  // coefficients: mask
3  // compile-time constants: MAX_IMAGE_WIDTH, MAX_MASK_HEIGHT,
      MAX_MASK_WIDTH, ROW_BW, COL_BW
4
5  // Further optimization is possible when these are compile-time constants
6  const int image_size = image_height * image_width;
7  const int initial_latency = image_width * mask_height / 2 + mask_width/2;
8
9  in_data_t lbuf[MAX_MASK_HEIGHT - 1][MAX_IMAGE_WIDTH];
10 #pragma HLS array_partition variable=lbuf dim=1 complete
11 #pragma HLS dependence variable=lbuf inter false
12 #pragma HLS dependence variable=lbuf intra false
13
14 in_data_t swin[MAX_MASK_HEIGHT][MAX_MASK_WIDTH];
15 #pragma HLS array_partition variable=swin dim=0 complete
16
17 // Registers to keep memory access addresses
18 ap_uint<ROW_BW> row = 0;
19 ap_uint<COL_BW> col = 0;
20
21 for(ap_uint<ROW_BW + COL_BW> clock_tick = 0;
22   clock_tick < initial_latency + image_size;
```

Annotations: "on-chip memory (BRAM) to hold dependency pixel" (lines 9–12); "on-chip memory (registers) to hold dependency pixel" (lines 14–15); "Bit-level precision to avoid unnecessary resource usage" (lines 18–19); "Life time" (line 22).

```
23      ++clock_tick)
24  {
25    #pragma HLS pipeline II=1
26
27    //**********************************************************
28    // Read new pixel
29    //**********************************************************
30    in_data_t newPixel = 0;
31    if(clock_tick < image_size)
32    {
33      newPixel = in[clock_tick];
34    }
35
36    //**********************************************************
37    // Shift line buffers
38    //**********************************************************
39    in_data_t new_swin_row[mask_height];
40    for(int i = 0; i < mask_height; ++i)
41    {
42      #pragma HLS unroll
43      if (i == 0)
44          {
45        new_swin_row[i] = lbuf[i][col];
46          }
47      else
48      {
49        data_t temp = (i == mask_height - 1) ? newPixel : lbuf[i][col];
50        lbuf[i - 1][col] = temp;
51        new_swin_row[i] = temp;
52      }
53    }
54
55    //**********************************************************
56    // Shift sliding window
57    //**********************************************************
58    for(int i = 0; i < mask_height; ++i)
59    {
60      #pragma HLS unroll factor=MAX_MASK_HEIGHT
61      for(int j = 0; j < mask_width - 1; ++j)
62        #pragma HLS unroll factor=MAX_MASK_WIDTH
63        swin[i][j] = bh(&swin[0][j + 1], i, col, MAX_MASK_WIDTH);
64
65      // store the last column of line buffers to swin
66      swin[i][mask_width - 1] = bh(new_swin_row, j, row, MAX_MASK_HEIGHT);
67    }
68
69    //**********************************************************
70    // Compute the convolution (actual algorithm)
71    //**********************************************************
72    out_data_t result = 0;
73    for(int j = 0; j < mask_height, ++j)
74      #pragma HLS unroll factor=MAX_MASK_HEIGHT
```

Update strategy for On-chip memory (BRAMs)

Update strategy for on-chip memory (registers)

Computation of convolution

```
75        for(int i = 0; i < mask_width, ++i)
76          #pragma HLS unroll factor=MAX_MASK_WIDTH
77          result += swin[j][i] * mask[j][i];
78
79        if(clock_tick >= initial_latency )
80        {
81            //*********************************************************
82            // Write Result
83            //*********************************************************
84            out[clock_tick - initial_latency] = result;
85
86            //*********************************************************
87            // Update image indexes
88            //*********************************************************
89            ++col;
90            if(col == image_width)
91            {
92                col = 0;
93                ++row;
94            }
95        }
96    }
```

2.3.3 Proposed Approach

As explained in Section 2.2, current HLS tools can generate RTL circuits from untimed C-based programs but deliver a good quality of synthesis results only when users are able to adequately describe the behavior of the hardware implementation for their application. Optimization strategies include describing an application-specific on-chip memory hierarchy, pipelining, increasing spatial parallelism, strength reduction, and decreasing the bit-width of variables and operations. Triggering these optimizations requires hardware design knowledge and writing tedious, unportable code that is tool-specific and error-prone. Furthermore, a C++ code or an OpenCL program that is optimized for an HLS tool would perform very poorly when executed on a CPU or a GPU. For instance, the convolution implementation in Listing 2.5 can only be written by understanding hardware design principles and performs poorly on other platforms. What is more, this description ignores image border handling and can further be improved with loop coarsening. These extensions would significantly increase the code's length (e.g., approximately 10× longer code depending on the description style and implementation technique).

This thesis mitigates these problems by domain-specific high-level abstractions to express applications as a dataflow graph consisting of point, local, and global operators. The data is expressed by special types, such as images. Users describe the mathematical function of these operators in a declarative way. Then, a domain-specific code generation mechanism (e.g.,, a source-to-source compiler) captures

the inherent parallelism of the input application to generate a highly sophisticated specification of an efficient application-specific circuit as input to HLS. Our approach still requires the developer of the code generation method (e.g., DSL developer) to understand hardware design techniques for the target image processing applications. However, thanks to code generation, it does not expose these low-level details to application developers. For instance, only Line 77 in Listing 2.5 describes the mathematical function of the convolution operator. Hence, this one-line description would be enough to describe the convolution function when described in a declarative way.

3 Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing

FPGAs and ASICs excel at exploiting ILP by allowing to implement deeply pipelined circuits, where the execution of successive loop iterations are overlapped to provide high throughput. In addition, exploiting DLP by processing multiple pixels in every clock cycle is crucial to increase throughput for high-speed execution (to the memory-bound at best) according to the Roofline model [WWP09]. This optimization is especially critical for FPGA implementations since a typical modern FPGA's programmable logic is slower than the frequencies of modern external memory technologies. While existing HLS tools are good at exploiting ILP from the application description, they provide poor results or very little support for extracting DLP.

In this chapter, we propose novel techniques that exploit DLPs in stencil-based image processing applications for FPGA and ASIC implementations [ÖRH+17b]. Our implementations are based on a technique called *loop coarsening*, where outer loops of an image processing algorithm are coarsened to read and write multiple pixels at a time. Correspondingly, we develop hardware implementations for image border handling that supports loop coarsening. These implementation techniques are automated through a code generation mechanism in Chapters 4 to 6. The FPGA synthesis results show that the proposed coarsening architecture uses 32 % fewer registers for a 5-by-5 convolution with a coarsening factor of 64 compared to previous works, whereas the proposed border handling architectures decrease the LUT usage by 36 %.

Similar to many other hardware design problems, it is hard to find a *one-size-fits-all solution*, i.e., whether an implementation strategy using fewer resources and providing higher performance than others depends on application parameters such as image width and stencil size. Therefore, we provide an analysis of the investigated implementation techniques for loop coarsening and border handling. This allows us

to develop an algorithm that selects the most resource-efficient technique from a set of implementations of a stencil-based image processing application for a given set of input parameters.

3.1 Introduction

In many application domains, the bit width of input and output data elements is much smaller than what modern memory technologies and communication interfaces offer. Furthermore, the logic speed of a typical "modern" FPGA is much lower than the memory technologies. For instance, AXI4 interfaces used in the Xilinx implementation for direct memory access (DMA) supports transferring $1,024$ bits of data concurrently into the reconfigurable logic [Xil17b]. Each DDR3 channel on a "modern" FPGA (Xilinx Zynq zc706) supports 12 GBytes/s memory bandwidth, allowing to have 512-bit wide interfaces for around 200 MHz logic frequency. User logic in Xilinx Alveo boards can access each high bandwidth memory (HBM) channel by 256-bit wide interfaces at 400 MHz [CCW+20]. What is more, high-speed serial transceiver technology enables higher data rate communication channels between FPGA and external devices [Xil17a].

In order to read and write at the speed of a communication interface, multiple data elements can be streamed in parallel into FPGA logic as shown in Figure 3.1. To also run an algorithm at the memory bandwidth limit by using an FPGA (or ASIC), data path of its hardware implementation must sufficiently be parallelized. For instance, the data bit width of a typical image ranges between 8 bits (grayscale) and 32 bits (RGBA). This means that a hardware implementation operating on a grayscale image at 200 MHz must be parallelized by a factor of 16 to read, write, and process at the memory speed of a DDR3 channel on a Zynq FPGA. A naive approach is to clone the entire accelerator by a factor of 16, but this is not resource-efficient. First, it is often possible to use one optimized control structure for all parallelized data paths, arithmetic calculations operating on input data. Second, and more importantly, redundant data is stored when a hardware accelerator circuit is replicated with its memory architecture (e.g., holding dependency pixels) to process multiple input in parallel. This is the case for local operators.

For instance, assume that the output of a local operator at image coordinates i and j depends on the $w \cdot h$ pixels at the input image, where the rectangular region is bounded by the coordinates $(i - \lfloor w/2 \rfloor, j - \lfloor h/2 \rfloor)$ and $(i + \lfloor w/2 \rfloor, j + \lfloor h/2 \rfloor)$. Then, outputs of v consecutive pixels depend on $(w + v - 1) \cdot h$ input pixels. Replicating the line-buffered implementation of a local operator by a factor of v, as shown in Figure 3.2b, requires holding $v \cdot w \cdot h$ pixels, where each of the replicate has its own line buffers and sliding window. Instead, resource and memory usage (thus power) can be reduced significantly through better resource sharing where a specially designed control structure as well as memory architecture is used for all the

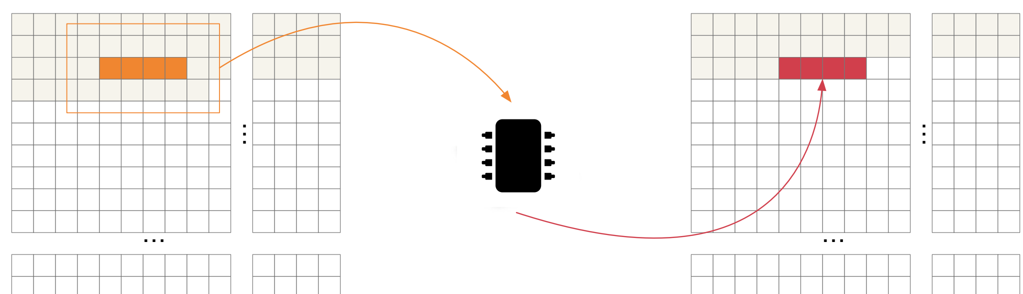


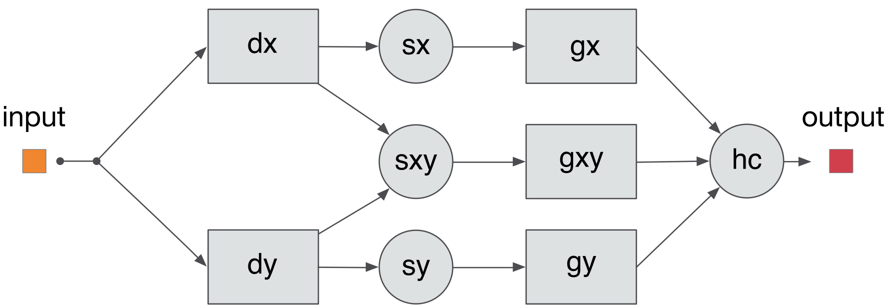
Figure 3.1: FPGAs have a lower logic speed compared to modern memory interfaces. Yet, memory can be read at a higher speed, and multiple pixels can be streamed in parallel into the reconfigurable logic. Therefore, exploiting DLP in image processing algorithms allows hardware implementations to compute at memory speed. In the example, 4 iterations of the loop nest are processed at a time and 4 result pixels are processed as a result.

arithmetic function replicates, as illustrated in Figure 3.2c. This chapter presents novel resource-efficient implementation techniques for parallelizing stencil-based image processing applications by processing multiple consecutive pixels at a time as shown in Figure 3.1.

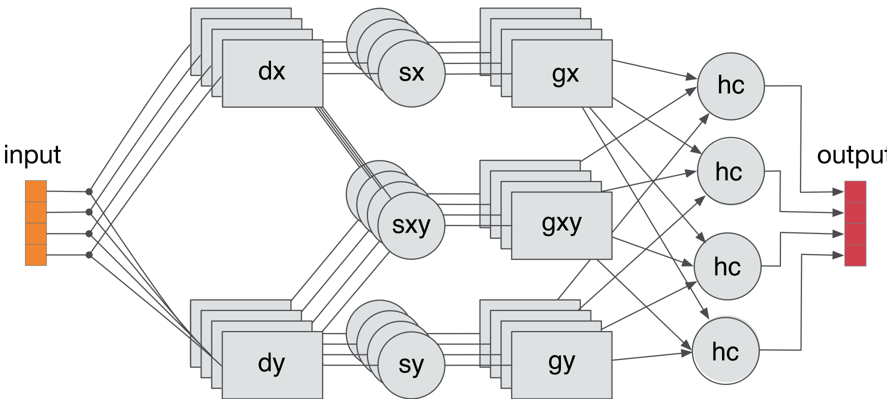
Our contributions of this chapter are summarized as follows[1]:

- Two novel hardware design techniques for loop coarsening that use significantly fewer registers than previous work (see Figure 3.3). Depending on the choice of operator parameters, either the first or the second architecture requires fewer resources. These are automatically generated from a DSL or a high-level function library presented in Chapters 4 to 6.
- The analysis and design of hardware architectures that support image border handling with a guaranteed throughput of the input stream for the processing of local image operators in combination with loop coarsening (see Figure 3.3).
- A systematic analysis of the investigated loop coarsening and border handling techniques. We formulate the resource usage of all the considered implementation techniques in terms of registers and MUX. In this way, we provide an algorithm for suggesting the most efficient hardware implementation for a given coarsening factor and input parameters of an algorithm, such as image width and stencil size.

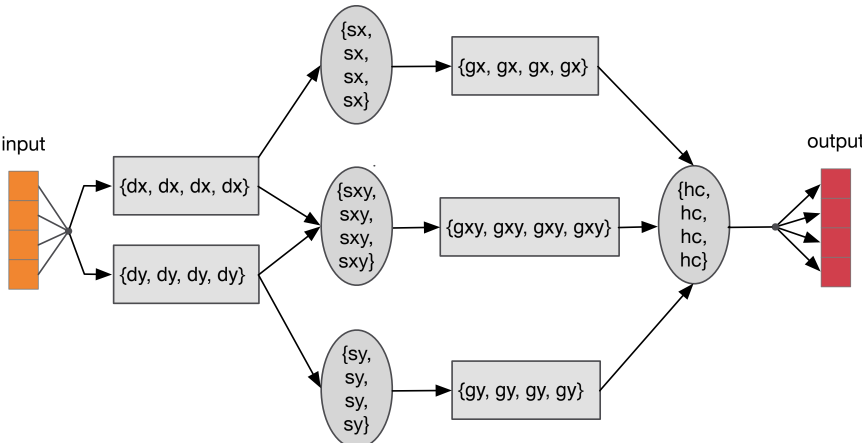
[1]The contents of this chapter are based on and partly published in [ÖRH+17b], which has appeared in the Proceedings of the 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), and [RÖH+18], which has appeared in Journal of Signal Processing Systems.



(a) Data flow graph (DFG) representation of Harris Corner based on image processing operators. Circle nodes represent point operators, while rectangular nodes represent local operators.



(b) A naive approach for parallelizing Harris Corner would be replicating the accelerator hardware. This requires utilizing replicates of the memory hierarchy (e.g., line buffers, sliding window) as well as the control and data paths.



(c) A resource-efficient approach to exploit DLP in the Harris Corner algorithm is to modify its implementation to handle multiple pixels. This requires understanding the algorithm and implementing an application-specific memory hierarchy, data path and control path.

Figure 3.2: The hardware implementation of an algorithm must be parallelized to reach the speed of modern a communication and memory interface designed for data transfer. This is illustrated for data flow graph (DFG) representation of a Harris Corner application.

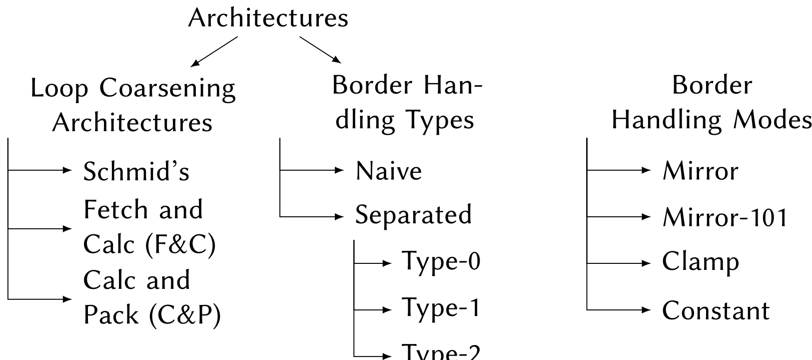


Figure 3.3: Suggested loop coarsening architectures, border handling types, and border handling modes considered.

- An analysis of the results obtained by Vivado HLS for both proposed variants and mentioned naive implementations.

3.1.1 Loop Coarsening

Loop coarsening is an implementation technique for parallelizing image processing applications by processing multiple consecutive pixels at a time as shown in Figure 3.1. The name is coined by Schmid et al. [SRH+15] since, as shown in Algorithm 1, the outer loop of an image processing algorithm is coarsened (Line 3) to aggregate multiple consecutive pixels into so-called *superpixels* (Line 3) and then process them in parallel (Line 9). It has the following advantages:

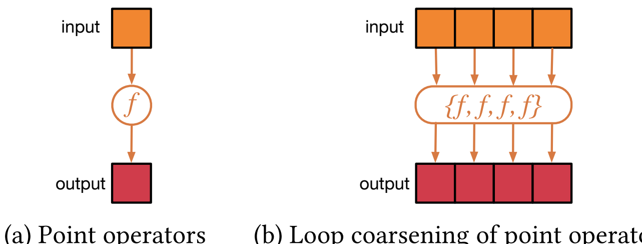
1. Memory access remains in burst mode, which allows highest external memory bandwidth.
2. Loop coarsening solely uses one optimized control structure for all parallelized data paths, which will be the hardware implementation of the arithmetic calculations operating on input data (f in Line 9).
3. One memory architecture is used for all the consecutive pixels, allowing to leverage temporal locality of the algorithm and increase data reuse.

Algorithm 1 considers that the typical implementation of an algorithm (e.g., local operator) uses an on-chip memory architecture (Line 5, e.g., line buffers) to reduce off-chip communication. This is the case for a great portion of algorithms where their hardware implementations deliver high-performance and/or energy efficiency. Implementing such a memory architecture depends on the application, thus the modifications required for loop coarsening.

Algorithm 1: A simplified description of the behaviour of a hardware architecture supporting loop coarsening. It shows that the image read/write schedule as well as on-chip memory architecture is modified to operate on superpixels. Only the arithmetic functions (f) are replicated whereas one control structure and memory is used for all the replicates. This increases resource sharing. However, modifications to on-chip memory and the way `Split`, and `Assemble` functions are implemented depend on the target application. Image border handling is ignored for simplicity, whose implementation also depends on the memory access pattern of the input algorithm.

```
   input  : imageIn, Image Width, Image Height, v, f
   output : imageOut
1  func Coarsening(imageOut, imageIn, Image Width, Image Height, v, f)
2      for y ← 0 to Image Height do
3          for x ← 0 to ⌈Image Width/v⌉ do
               // read new superpixel
4              supPxl ← Read(imageIn, x, y)
               // update on-chip memory
5              onchipMem ← Update(onchipMem, supPxl, x, y)
               // get an accessor for the superpixel
6              supAcc[k] ← Read(onchipMem, x, y)
               // compute result for the input superpixel
7              for k ← 0 to v − 1 do
                   // data separation for subcomputations
8                  subAcc[k] ← Split(supAcc)
                   // subcomputation for the input pixel
9                  subPxl ← f(k);
                   // memory packing
10                 supPxl ← Assemble(subPxl, k)
11             end
               // write the output superpixel
12             Write(imageOut, supPxl, x, y)
13         end
14     end
15 end
```

For this reason, HLS tools provide very little or no support to automatically apply such a parallelization optimization for an algorithm described in a C-based



(a) Point operators (b) Loop coarsening of point operators

Figure 3.4: Implementation of loop coarsening for point operators is straightforward since it is a one-to-one mapping function.

language.[2] This is not a surprise since compiler-based parallelization techniques such as autoparallelization and autovectorization have been researched for over 50 years to leverage multicore acceleration and CPU vector units, yet did not meet the level of performance expectations [CCF+10]. Considering that implementing an application for an FPGA or an ASIC is more complex than compiling an application for an ISA where the underlying hardware is fixed, providing autoparallelization techniques from C-based languages is not a challenging task. Instead, in this dissertation, we raise the abstraction level in HLS by a domain-specific library or a language to extract DLP of an image processing algorithm, thus automatically apply loop coarsening.

Implementation of loop coarsening is rather straightforward for point operators as illustrated in Figure 3.4. The output of a point operator solely depends on the input pixel at the same image coordinates. Therefore, its hardware implementation does not require a complex on-chip memory architecture. Utilizing hardware logic similar to SIMD units by simply replicating the arithmetic functions is enough for the coarsening of point operators.

Loop coarsening of global operators such as reduction and histogram follows Algorithm 1 and requires only small modifications on implementation of their memory architectures. Contrarily, loop coarsening of local operators is complex, and even more, it complicates hardware implementation of image border handling. As nearly all image processing applications include multiple local operators and FPGAs become top-notch platforms in terms of throughput per energy for the implementation of fixed-point applications based on local operators. Therefore, even the smallest improvements in their implementations may provide significant improvements in terms of resource requirements and throughput.

[2] In particularly, Xilinx HLS tools do not offer a mechanism for automatically exploiting DLP. Intel FPGA SDK for OpenCL can parallelize an NDRange kernel with special attributes, which are `num_compute_units` (similar to Figure 3.2b) and `num_simd_work_items` (similar to vector engines of a multicore). Yet the kernel should be described similar to a GPU kernel, e.g., a line-buffered stencil function cannot be parallelized [Int17]. We refer to Chapter 4 for more details.

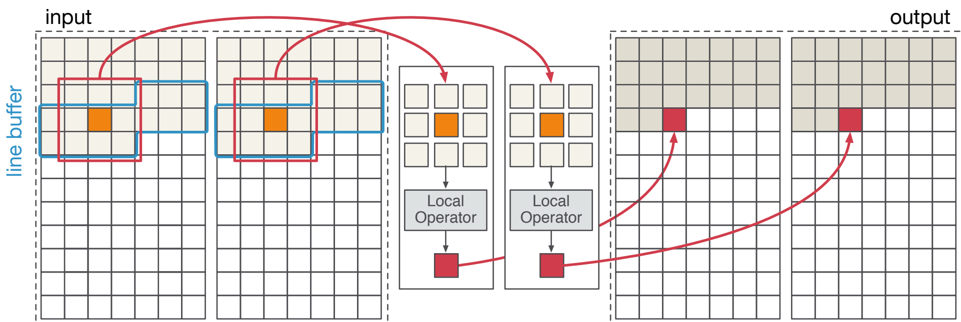


Figure 3.5: Loop tiling for local operators [RÖH+18]. The input image is split vertically into multiple parts. DLP is increased by processing all the image tiles in parallel. In the case of the hardware implementation of local operators, loop tiling replicates the sliding window as well as the arithmetic functions and requires more line buffers. Overlapping regions between the image tiles need to be stored on each of these line buffers. For these reasons, resource sharing is higher compared to the replication approach shown in Figure 3.2b but less efficient than loop coarsening. (Figure reprinted from [RÖH+18], © 2018 IEEE)

Comparison with Loop Tiling Loop tiling is a well-established approach for increasing DLP by splitting an input image into multiple tiles and processing them in parallel as shown in Figure 3.5. This introduces more overhead compared to loop coarsening. First, splitting an input image into tiles requires distributing incoming pixels to multiple buffers, which can be achieved in hardware or software. Second, the same data might need to be transferred multiple times, e.g., overlapping regions must be transferred multiple times for the implementation of local operators. Furthermore, processing all the image tiles in parallel requires replicating the same hardware logic. Therefore, accelerators share fewer resources in loop tiling compared to loop coarsening. We refer to our work in [RÖH+18] for more details and benchmarks.

Another use case of loop tiling is to process each image tile sequentially. This does not increase the DLP, in fact it often slows down the hardware execution. For instance, each image tile in Figure 3.5 could be processed one after another. In this scenario, loop tiling might have the following advantages: i) It can accelerate data transfers between host and device by using one virtual page for each tile and resizing the data transfer size to the optimum. These parameters depend on the communication interface [PBY+17]. ii) Processing a tile that is smaller than the image might reduce the resource usage, e.g., smaller line buffers are able to hold image tiles (as in Figure 3.5) compared to storing the whole image lines. However, this is not the focus of this chapter.

Table 3.1: Notation used in this chapter

v	coarsening factor
W	image width
H	image height
k_{in}	input image bit width
k_{out}	output image bit width
w	width of local operator window
h	height of local operator window
r_w	$\lfloor w/2 \rfloor$, horizontal radius
r_h	$\lfloor h/2 \rfloor$, vertical radius
r_{wv}	$\lceil r_w/v \rceil$, minimum number of data beats larger than or equal to r_w
$\mathrm{MUX}[n]$	n input multiplexer hardware
$C_{\mathrm{FF}}^{\mathrm{arch}}$	register usage estimated for the implementation *arch*
$C_{\mathrm{mux}}^{\mathrm{arch}}$	MUXs usage estimated for the implementation *arch*
$C_{\mathrm{FF}}^{\mathrm{arch}}(\mathrm{type})$	$C_{\mathrm{FF}}^{\mathrm{arch}}$ for the selected *border handling type*
$C_{\mathrm{mux}}^{\mathrm{arch}}(\mathrm{type})$	$C_{\mathrm{mux}}^{\mathrm{arch}}$ for the selected *border handling type*
$T_{\mathrm{CriticalPath}}^{\mathrm{type}}$	estimation of the critical path delay

Differences to Explicit Vectorization ISAs consisting of SIMD units support operations on vector data types. Thereby, one instruction is used to apply the same type of operation on multiple data, i.e., mostly arithmetic operations such as multiplication. One example is to apply convolution on a red, green, blue, alpha channel (RGBA) image, where the same multiplication and addition is applied to all image channels. However, vectorizing a local operator on consecutive pixels is much more complex: new vector data types must be created by extracting pixels from neighbor vectors. Unlikely, superpixel contains a continuous pixel space in loop coarsening.

3.1.2 Notation

This section introduces the notation of important parameters used in this chapter to increase clarity (see Table 3.1). Column and row image coordinates are represented

by x and y. W/H and w/h refer to the width and height of an image and the width and height of a local operator, respectively. Based on that, $r_w = \lfloor w/2 \rfloor$ and $r_h = \lfloor h/2 \rfloor$ represent the integer radius of the operator. Parameter v represents the so-called coarsening factor and k_{in}, k_{out} are input and output bit widths of a local operator.

Moreover, we introduce two types of area cost functions: C_{FF} and C_{mux}. $C_{\text{FF}}^{\text{arch}}$ denotes the number of registers that are estimated to be used for *arch*. Similarly, $C_{\text{mux}}^{\text{arch}}$ indicates the number of MUXs. $C_{\text{FF}}^{\text{arch}}(\text{type})$ and $C_{\text{mux}}^{\text{arch}}(\text{type})$ represent the cost functions for a selected border handling *type*. Furthermore, we denote $\text{MUX}[n]$ to refer to a multiplexer with n inputs. As a speed measure, $T_{\text{CriticalPath}}^{\text{type}}$ denotes the time estimation of the critical path of an architecture that implements border handling of *type*.

3.2 Schmid's Loop Coarsening

DLP of local operators reading data in raster order can be increased by processing groups of data elements, so-called *data beats*, in each cycle. Schmid et al. [SRH+15] denote this technique as *loop coarsening*, since coarsening is applied to the outer loop of a stencil operation, reading data in raster order.

A representation of Schmid's loop coarsening architecture is given in Figure 3.6. The line buffer and sliding window are modified to store data beats consisting of v pixels. The calculation is replicated by a factor of v followed by a unit for packing the results into a single output data beat. Coarsening the loop of a local operator by a factor v provides the throughput (Th) and latency (L) equations given Eq. (3.1).

$$\begin{aligned} Th^{\text{out}} &= Th^{\text{in}} = v \text{ pixels/cycle} \\ L &= (\lfloor h/2 \rfloor \cdot \lceil W/v \rceil + \lfloor \lceil w/v \rceil /2 \rfloor + L_{\text{calc}}) + (\lceil W/v \rceil \cdot H) \end{aligned} \tag{3.1}$$

3.3 Proposed Loop Coarsening

In the following, we propose two novel architectures supporting loop coarsening with equal performance as in Eq. (3.1), but with less area usage [ÖRH+17b]. While the first one, called *Fetch and Calc* (F&C) is an optimized version of Schmid's loop coarsening architecture [SRH+15], the other, named *Calc and Pack* (C&P) uses a different schedule. In Section 3.6, it is shown that which of these is the more efficient architecture depends on the parameters w, h, k_{in}, k_{out}, and the border handling mode. Both architectures have in common that the sliding window is shifted in each cycle by v steps.

For explanation purposes, we group the registers holding the pixels of a sliding window into regions (denoted by R) according to their temporal content as shown

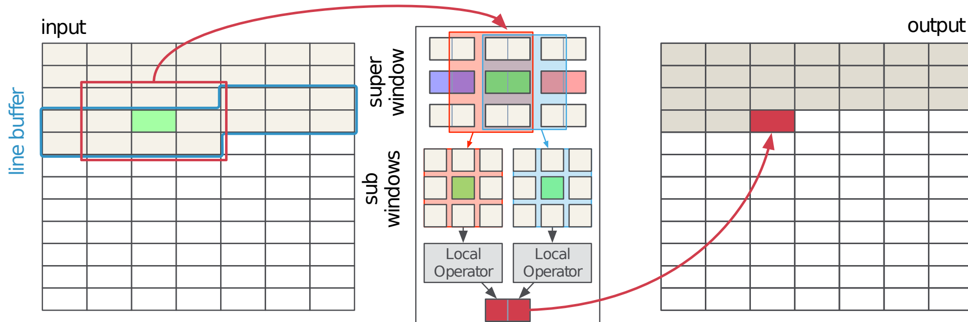


Figure 3.6: Schmid's loop coarsening [SRH+15] with two pixels processed at once ($v = 2$) for a local operator of size 3×3. (Figure reprinted from [SRH+15], © 2015 IEEE)

in Figure 3.8. R_{left} and R_{right} denote the registers that hold the neighboring pixels to process v pixels at R_{mid}. Similarly, R_{fetch} labels the registers that store the newest data beats. We also use colors in the figures for the labels R_{fetch} (red) and R_{mid} (green) since they can overlap.

3.3.1 Fetch and Calc

Ideally, coarsening a (w, h) local operator by v requires a sliding window that holds $(w + v - 1) \cdot h$ pixels. However, this is not the case in Schmid's loop coarsening implementation, where the sliding window always holds $w \cdot h \cdot v$ pixels. In each clock cycle, Schmid's loop coarsening reads (fetches) v new pixels into register region R_{right} and produces v result pixels stored in R_{mid} as shown in Figures 3.7 and 3.8.

In the following cases, Schmid's loop coarsening stores redundant pixels (which are optimized in F&C):

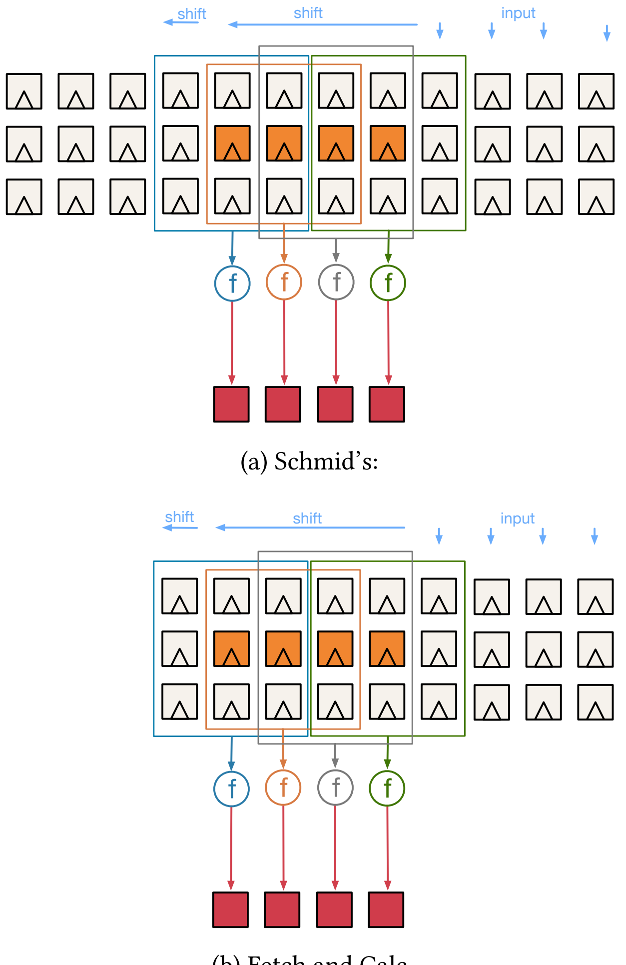
i when $v > r_w$: v number of pixels are stored in R_{right} instead of only r_w, which is the required number of pixels to produce results for v number of pixels stored in R_{mid}.

ii when $v < r_w$: Schmid's architecture requires $v - (v \bmod r_w)$ number of extra FFs in R_{right}, solely to store v pixels that are required in the next iteration.

The number of required FFs in Schmid's sliding window can be expressed by Eq. (3.2).

$$C_{\text{FF}}^{\text{Schmid's}} = k_{\text{in}} \cdot h \cdot v \cdot (2 \cdot \lceil r_w/v \rceil + 1) \tag{3.2}$$

While the unnecessary pixels in R_{right} are necessary for calculations in subsequent cycles, it is not necessary to store more than r_w pixels in R_{left}. Hence, as an improvement to [SRH+15], the number of pixels in R_{left} may be reduced to r_w as shown in



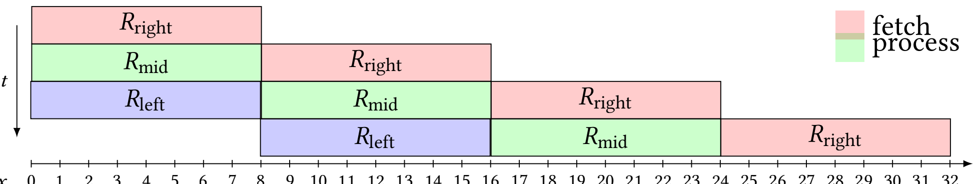
(a) Schmid's:

(b) Fetch and Calc

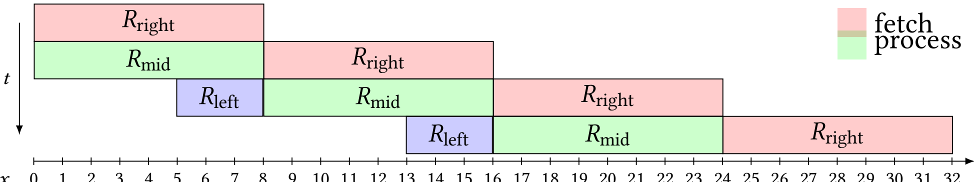
Figure 3.7: F&C optimizes resource usage in Schmid's loop coarsening by eliminating unnecessary FFs from its sliding window. The difference is shown for a $(w = 3, h = 3)$ local operator, and the coarsening factor is $v = 4$. In every iteration (i.e., clock cycle when II=1), the sliding windows of both implementations read (*fetch*) 4 new pixels (which are held in R_{right}) and produce 4 results (for the pixels stored in R_{mid}). Schmid's implementation unnecessarily holds 9 pixels on its left side (R_{left}).

Figure 3.8b. Eq. (3.3) gives the number of pixels that need to be fetched in a row of a coarsened sliding window for any given v and r_w.

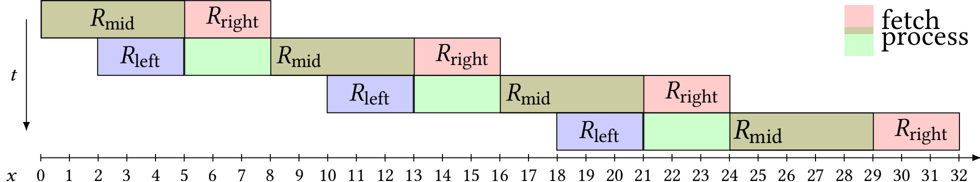
$$C_{\text{FF}}^{\text{F\&C}} = k_{\text{in}} \cdot h \cdot (r_w + v \cdot (\lceil r_w/v \rceil + 1)) + C_{\text{FF}}^{\text{F\&C}}(b) \tag{3.3}$$



(a) Schmid's: Processing starts at the second cycle. Each row holds $v + 2 \cdot (v \cdot \lceil r_w/v \rceil) = 24$ pixels starting with the third cycle for $v = 8$ and $r_w = 3$.



(b) Fetch and Calc: Processing starts at the second cycle. Each row holds $r_w + v + (v \cdot \lceil r_w/v \rceil) = 19$ pixels starting with the third cycle for $v = 8$ and $r_w = 3$.



(c) Calc and Pack: Processing starts at the first cycle. Writing starts at the second cycle. Writing the overlap of *fetch* and *process* is delayed by one cycle. Each row holds $v + 2 \cdot r_w = 14$ pixels starting with the second cycle for $v = 8$ and $r_w = 3$.

Figure 3.8: First 4 iterations (clock cycles) of the raster order image read schedule, where x axis shows horizontal pixel coordinates of an image row, whose width $w = 7$, radius $r_w = 3$ and coarsening factor $v = 8$. For explanation purposes, row of a one dimensional sliding window is divided into three regions, namely R_{left}, R_{right}, and R_{mid} as explained in Section 3.3. The considered local operator calculates v number of output values at the image coordinates for the pixels hold in R_{mid}. Output of R_{mid} depends on the neighboring pixels that stored in R_{left} and R_{right}. Similar to Figure 3.6, *Fetch* (colored by red) and *process* (colored by green) represent pixels read to sliding window and processed in each cycle, respectively. All the considered loop coarsening implementations start writing an *output data beat* (v output) in the second cycle. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

3.3.2 Calc and Pack

C&P reduces the number of FFs required to store new pixels in Schmid's implementation and F&C (in other words, resources used for *fetching*) by using an alternative schedule. In this schedule, the sliding window holds the minimum number of pixels required for the stencil calculation according to Eq. (3.4), so that the newest input *data beat* is not only stored in R_{right} but also in R_{mid}:

$$|R_{\text{left}}| = |R_{\text{right}}| = r_w, \quad |R_{\text{mid}}| = v \tag{3.4}$$

The sliding window of C&P is shifted by v in each cycle. This splits the processing order at R_{mid} into two cycles. Consequently, an output *data beat* is partially calculated in two cycles and written in the second cycle using the memory packing scheme in Eq. (3.5).

$$\begin{aligned}\text{output}([x, x+v], t) = \text{pack}\{&\text{out}([0, v - r_w - 1], t-1),\\ &\text{out}([v - r_w, v-1], t)\}\end{aligned} \tag{3.5}$$

The described schedule is implemented using additional registers that are placed after the leftmost $v - r_w$ local operators in order to the hold results from the previous cycle. For instance, C&P's schedule for a kernel with $w = 7$ and $v = 8$ is given in Figure 3.8c. As it can be seen, the number of pixels stored in a row remains ideal $w - 1 + v = 14$, but $v - r_w = 5$ additional registers are necessary in order to delay the results of the previous cycle. Eqs. (3.6) and (3.7) give the costs necessary per row of a coarsened sliding window for any given v and r_w.

$$C_{\text{FF}}^{\text{C\&P}}(d) = k_{\text{out}} \cdot (v - (r_w \bmod v)) \tag{3.6}$$

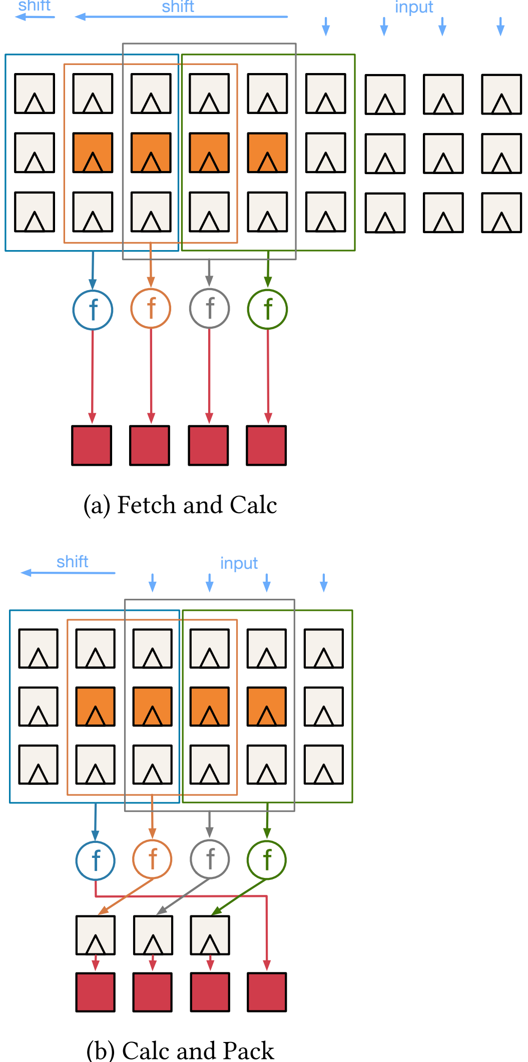
$$C_{\text{FF}}^{\text{C\&P}} = k_{\text{in}} \cdot h \cdot (2 \cdot r_w + v) + C_{\text{FF}}^{\text{C\&P}}(d) + C_{\text{FF}}^{\text{C\&P}}(b) \tag{3.7}$$

3.4 Analysis of Border Handling

A local operator may depend on pixels from outside the image at the image borders defined as:

$$\begin{aligned}0 \le x < \lfloor w/2 \rfloor \quad &\vee \quad x > W - \lceil w/2 \rceil\\ 0 \le y < \lfloor h/2 \rfloor \quad &\vee \quad y > H - \lceil h/2 \rceil\end{aligned} \tag{3.8}$$

A solution is handling the data according to well-known border patterns, as in Figure 3.10. Clamping virtually extends the input image with the pixel value on the border whereas constant pattern assigns the pixels at the virtually extended image with a constant value. Mirroring pattern replicates the pixels on border according to their distance to image border, as shown in Figure 3.10. A common solution



(a) Fetch and Calc

(b) Calc and Pack

Figure 3.9: C&P uses the minimum number of FFs to hold $(w + v - 1) * h$ pixels (only the required pixels for stencil calculation) in its sliding window at the cost of using additional registers to *pack* results at the output. The difference is shown for a $(w = 3, h = 3)$ local operator, and the coarsening factor is $v = 4$. C&P starts processing input data one iteration before F&C as shown in Figure 3.8, yet the latency of C&P is same with F&C.

is padding the input image to a larger size before local operators, but this may stall the image pipeline and decrease the throughput. Alternatively, a non-stalling implementation can be achieved by selecting appropriate data before the line buffers and sliding window according to the vertical and horizontal image coordinates, respectively [ÖRH+17b].

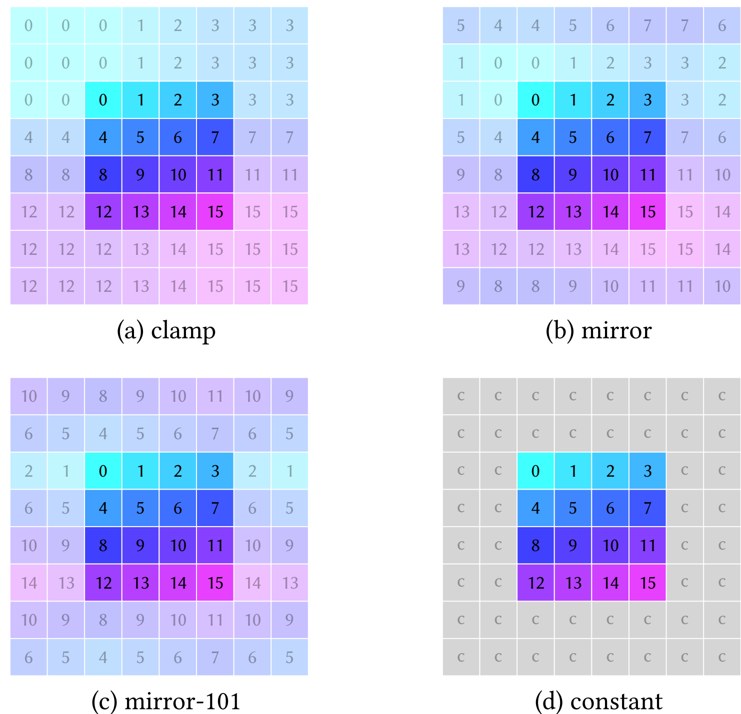


Figure 3.10: Common border handling modes. (Figure reprinted from [ÖRH⁺17b], © 2017 IEEE)

In the following, we analyze the hardware implementation properties of border handling modes shown in Figure 3.10 to provide an analytical basis for hardware architectures proposed in Section 3.4.

3.4.1 Naïve Border Handling

A straightforward approach for border handling, which is also common in software implementations, is to process reads on the sliding window. Considering W, H, w, h to be constant, and the read coordinates for the sliding window depend on the combination of input indices x, y, i, j. The indices i and j address window elements in the range $[0, w-1]$ and $[0, h-1]$, respectively. Depending on the current position of the operator within the image, we have to consider w different border cases for x, e.g., $[0, 1, \text{else}, W-2, W-1]$ for $w = 5$. Similarly, h different border cases for y need to be considered. In total, this results in $w^2 \cdot h^2$ different input index combinations for (i, x) and (j, y).

Assuming I_s is the union of all input index combinations, then the cost of the data selection hardware would be a function of $|I_s|$. Luckily, set I_s can be separated into two sets $I_w \times I_h$. Here, I_w depends only on x and i indices, resulting in $|I_w| = w^2$ index

combinations and $|I_h| = h^2$ analogously. Depending on the image size, not all operator positions are relevant to every window element, e.g., the center element is never subject to any border case, as it is always fetched from a valid image region. Consequently, for an image size of at least $\lceil w/2 \rceil \times \lceil h/2 \rceil$ pixels, the number of input index combinations can be reduced to a maximum of $|I_w| = 1 + 2 \cdot \sum_{i=2}^{\lceil w/2 \rceil} i$. According to Eq. (3.9), $|I_s|$ can be reduced to only 121 instead of 625 output indices for a 5×5 local operator.

$$\begin{aligned} |I_s| &= |I_w| \cdot |I_h| \\ |I_w| &= C(w),\ |I_h| = C(h) \end{aligned} \qquad C(n) = 1 + 2 \cdot \sum_{i=2}^{\lceil n/2 \rceil} i \tag{3.9}$$

3.4.2 Separated Border Handling

The following properties imply spatial and temporal features that can further improve the so called naive border handling architecture:

1. A MUX circuit is required for the hardware implementation of an output data selection from multiple inputs. Implementation of the naive border handling approach introduced in Section 3.4.1 requires utilizing $w^2 \cdot h^2$ data selection circuits (i.e., two input MUXs). However, MUX usage can be reduced for the 4 considered border handling modes in Figure 3.10 since number of required data selections decreases towards the center of a local operator.

2. Assume that I_h and I_w consist of all the y and x index combinations, then Eq. (3.10) always satisfies

$$|I_s| = |I_w \times I_h| = |I_w| \cdot |I_h| \tag{3.10}$$

3. Every pixel read to the first register of a sliding window row is reused in the following $w - 1$ cycles, shown in Eq. (3.11).

$$\begin{aligned} \text{in}[x, y] = \{\text{wind}[i, j], t\} &= \{\text{wind}[i-1, j], t+1\} \\ = \ldots &= \{\text{wind}[i-w+1, j], t+w-1\} \end{aligned} \tag{3.11}$$

The first feature implies that the conditional selection through the x and y axes are orthogonal to each other. This means that the row selection shown, in y-direction, can be separated from the column selection, in x, which is the same for all columns of a local operator. Furthermore, the row selection can be implemented only once before any pixel is read to the sliding window, which reduces the border handling cost from $|I_s|$ to $|I_h| + h \cdot |I_w|$. By separating the row and column selection from each other, $|I_s|$ reduces from $11 \cdot 11 = 121$ to $6 \cdot 11 = 66$. The top-level architecture that separates row selection and column selection is given in Figure 3.11.

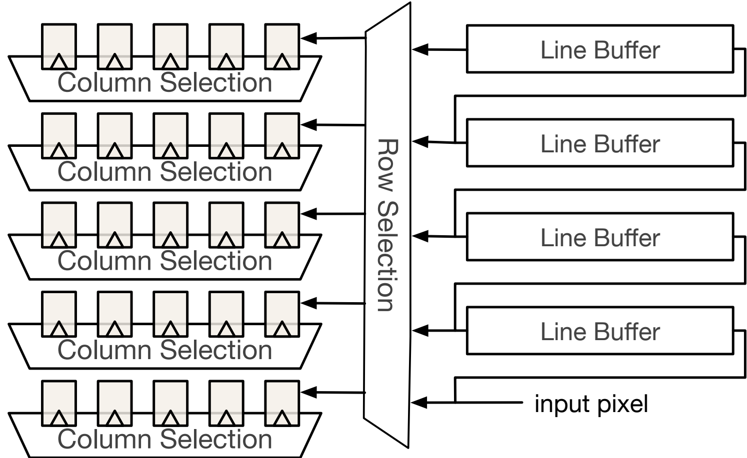


Figure 3.11: Separated border handling architecture: Row and column selection consist of MUXs for implementing border handling index combination sets I_w and I_h, respectively. One row selection is placed before the sliding window, contrary to the naïve type, deploying $w = 5$ of them after each column. (Figure reprinted from [ÖRH⁺17b], © 2017 IEEE)

Row Selection

The row selection circuit shown in Figure 3.11 reads from line buffers and writes to the sliding window without increasing the critical path of the local operator. Moreover, no locality can be exploited.

Column Selection

Whereas the column selection can use the same circuits with row selection, its output is directly connected with the local operators' data path, needing extra registers in between for pipelining the critical path. Moreover, temporal locality can be exploited. Therefore, we conduct an analysis to find minimum number of registers and MUXs under certain assumptions.

A timed data flow for column selection is given in Figure 3.12, where R_{mid} represents the center pixel of the local operator. Investigating the steps from $x = W - 4$ to process the next image row at $x = 1$ reveals all the corner cases for border handling. Investigating all 4 border handling modes shown in Figure 3.10 reveals that they can be implemented with similar design patterns. It can be observed that the window does not need to fetch a new pixel in the interval $x = [W - 3, W - 1]$. However, multiple pixels for the next image row must be read at once when $x = 0$. Assuming that the streaming is not stalled and one pixel is read in each cycle, at least r_w pixels

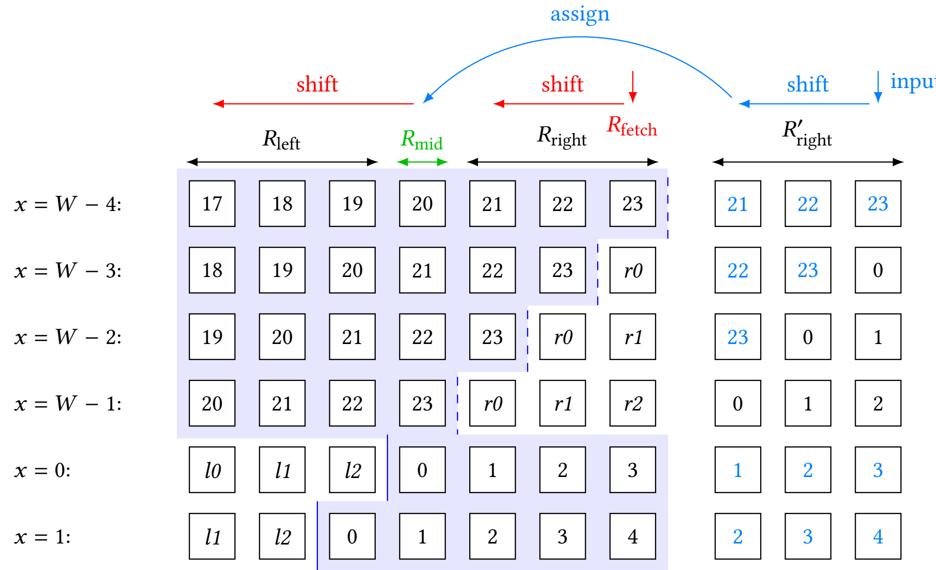


Figure 3.12: Temporal data flow for row data selection Type-1 with a local operator of size $w = 7$. The blue background denotes valid image regions, while lX and rX represent variable values depending on the corresponding border handling mode. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

per row[3] must be fetched at $x = 0$ in order to initialize all column pixels. Eq. (3.12) defines the minimum number of registers in a sliding window in terms of bits.

$$C_{FF}^{min} = h \cdot k_{in} \cdot (w + C_{FF}^{min}(b)),\; C_{FF}^{min}(b) = r_w \tag{3.12}$$

Under this assumption, a column selection architecture using the minimum number of registers has the following features:

i Except at $x = 0$, border handling can be achieved only through data selection that appropriately feeds R_{fetch} and shifts the content stored in R_{right}, R_{mid} and R_{left}.

ii All registers, except R_{fetch}, should be able to read from R'_{right} in order to initialize all column pixels at $x = 0$.

iii R'_{right} fetches one pixel in every cycle, but only at $x = 0$, reads from R'_{right} become indispensable. This fact renders the blue highlighted area in Figure 3.12 to be needless.

Under the assumption that a selection can only be implemented using a MUX, the minimum column selection for border handling uses at least the following resources:

[3]Note that additional registers utilized for border handling are represented by R'_{right}.

i There must be at least one MUX[2] before any register in R_{right} and R_{left} since they must be able to read from R'_{right} at $x = 0$.

ii R_{mid} can be implemented to read only from the leftmost register in R'_{right}. If the blue area is turned off, there must be at least one MUX[2] to read from R_{right}.

iii The size of the MUX before R_{fetch} depends on the border handling mode.

iv Only the data selection before R_{fetch} and the blue portion of R'_{right} can be optimized. Resource usage is identical for all border handling modes.

In conclusion, for further optimization, the blue highlighted portion of R'_{right} can be shut down for the interval $x = [0, W - r_w)$ by modifying the selection before R_{fetch} and R_{mid}. Eq. (3.13) shows the minimum number of MUXs for the discussed border handling modes.

$$C_{\text{mux}}^{\text{min}} = ((2 \cdot r_w - 1)\text{MUX}[2] + \text{MUX}[2]) \cdot h \cdot k_{\text{in}} + C_{\text{mux}}^{\text{new}} \tag{3.13}$$

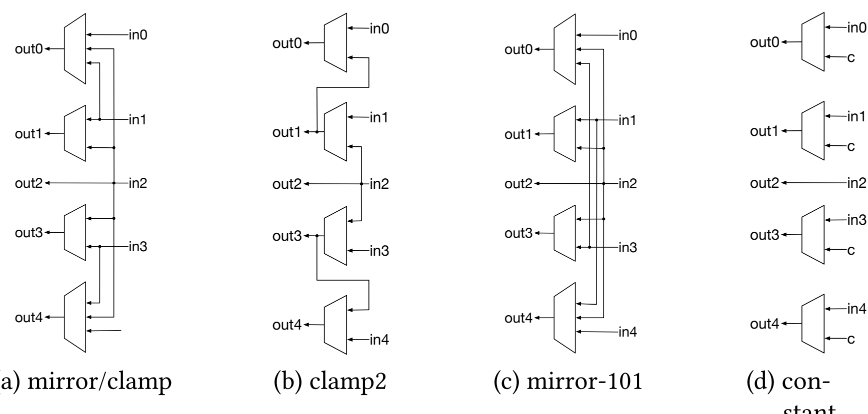
3.5 Hardware Architectures for Image Border Handling

In this section, we propose hardware implementations of the row and column selection circuits, provide their cost functions, and investigate their effects on the circuit speed. In addition, we extend our proposed implementations to loop coarsening.

3.5.1 Row Selection

Row selection circuits for the considered border handling modes are shown in Figure 3.13. *Clamp* and *mirror* may be implemented by the same architecture with different control paths. For *clamp*, I_w can be reduced even more by wiring $\text{out}(y)$ to $\text{out}(y+1)$ and $\text{out}(y-1)$ at the top and lower border, respectively. A corresponding row selection circuit, called *clamp2*, is shown in Figure 3.13b. Despite the simplification in selection size for *clamp2*, its longer critical path makes it barely appealing for fast implementations. Cost functions for all architectures are given in Eqs. (3.14) and (3.15), where $T(MUX[n])$ denotes the timing delay of the combinatorial circuit utilize for an n-input multiplexer implementation and $T_{\text{CriticalPath}}$ is the logic delay of the longest path that limits the maximum achievable clock frequency.

$$C_{\text{mux}}^{\text{RowSelect}} = 2 \cdot \begin{cases} \sum_{i=2}^{r_h+1} \text{MUX}[i], & \text{mirror-101, mirror, clamp} \\ r_h \cdot \text{MUX}[2], & \text{clamp2} \\ \text{MUX}[2], & \text{constant} \end{cases} \tag{3.14}$$



(a) mirror/clamp (b) clamp2 (c) mirror-101 (d) constant

Figure 3.13: Border handling row data selection. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

$$T_{\text{CriticalPath}}^{\text{RowSelect}} = \begin{cases} T(\text{MUX}[r_h + 1]), & \text{mirror-101, mirror, clamp} \\ T(r_h \cdot \text{MUX}[2]), & \text{clamp2} \\ T(\text{MUX}[2]), & \text{constant} \end{cases} \tag{3.15}$$

3.5.2 Column Selection

In this section, we present three types to design the column selection hardware. Type-0 is explained for comparison reasons only, since it is a commonly known approach. On the other hand, Type-1 and Type-2 follow the analysis results discussed in Section 3.4. Register requirements for both types are equal to the amount claimed to be minimum in Section 3.4.

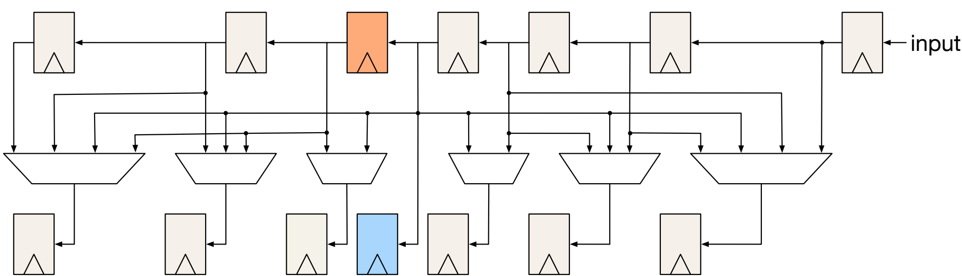
$$C_{\text{FF}}^{\text{Type-2}}(b) = C_{\text{FF}}^{\text{Type-1}}(b) = C_{\text{FF}}^{\text{min}}(b) \tag{3.16}$$

Similarly, their design approach assumes that there is one MUX[2] before every register in R_{right} and R_{left}. The optimization focuses on the selection circuit before R_{fetch} and R_{mid} by restructuring R'_{right}.

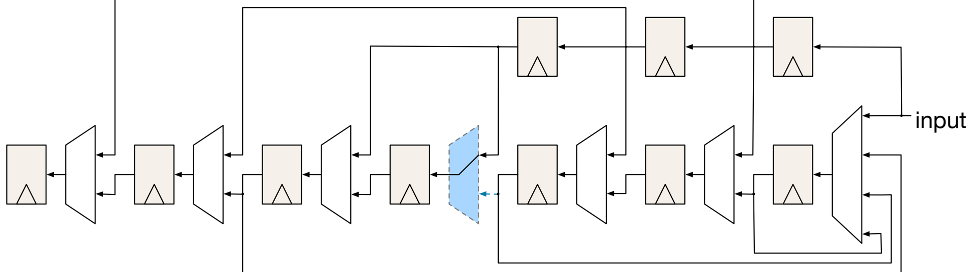
Type-0

Column selection can be implemented by transposing row selection architectures discussed in Section 3.5.1. Figure 3.14a shows the mirror border handling. Whereas the cost of this common approach, shown in Eq. (3.17), is twice the registers of Eq. (3.16).

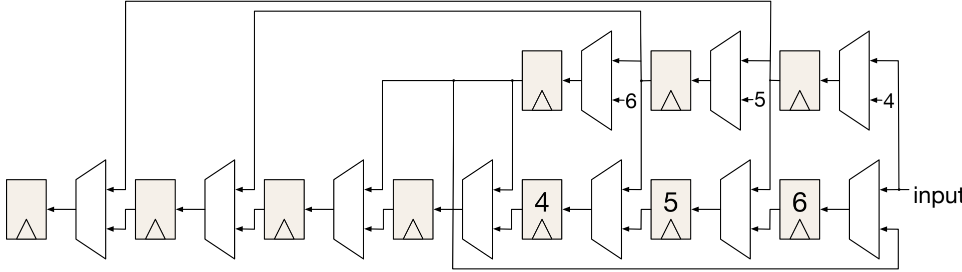
$$C_{\text{FF}}^{\text{Type-0}}(b) = h \cdot k_{\text{in}} \cdot (2 \cdot r_w) \tag{3.17}$$



(a) Type-0: Transpose of row selection followed by pipelining registers.



(b) Type-1: Minimal number of registers is used, and the selection at the upper right register region ($R_{\text{right'}}$) is optimized according to Section 3.4.



(c) Type-2: Minimal number of registers is used, and the selection before the input register (R_{fetch}) is optimized according to Section 3.4.

Figure 3.14: Column data selections for *mirror* with $w = 7$. Vivado HLS removes the blue register in Figure 3.14a since it stores the same data as the orange one. (Figure reprinted from [ÖRH⁺17b], © 2017 IEEE)

Similarly, the selection is also larger since there is more than one MUX[2] before R_{right} and R_{left}. The reason is that this type does not exploit temporal locality in x-direction. The MUX cost for the selection is the same as the cost in Eq. (3.14), where r_h is replaced by r_w. On the contrary, the critical path for clamp2, as defined by Eq. (3.18), is shorter since the selection and temporal direction is the same.

$$T_{\text{CriticalPath}}^{\text{Type-0}} = \begin{cases} T(\text{MUX}[r_w + 1]), & \text{mirror-101, mirror, clamp} \\ T(\text{MUX}[2]), & \text{clamp2, constant} \end{cases} \tag{3.18}$$

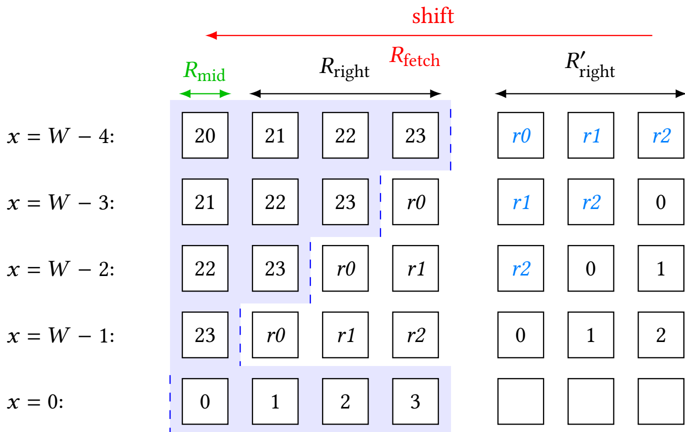


Figure 3.15: Temporal data flow that minimizes the selection before R_{fetch} (Type-2 row data selection). R_{left} was omitted for demonstration purposes only. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

Type-1

Temporal data flow for Type-1 for a $h{\times}11$ kernel is already given in Figure 3.12. Registers in the blue highlighted region in R'_{right} read from the previous one, thus, no MUX is needed for any register in R'_{right}. Moreover, the MUX before R_{mid} can be eliminated if R'_{right} is not switched off. This MUX is highlighted in blue in Eq. (3.19). By analyzing the registers for temporal data flow, it can be seen that there are types for any arbitrary size. Example architectures for all border handling modes are given in Figure 3.14b. The critical path and the selection cost in terms of MUXs are given in Eqs. (3.19) and (3.20).

$$C^{\text{Type-1}}_{\text{mux}} = (\text{MUX}[2] + \begin{cases} \text{MUX}[r_w + 1], & \text{mirror-101, mirror} \\ \text{MUX}[2], & \text{clamp, constant} \end{cases} \\ + (2 \cdot r_w - 1)\text{MUX}[2]) \cdot h \cdot k_{\text{in}} \tag{3.19}$$

$$T^{\text{Type-1}}_{\text{CriticalPath}} = \begin{cases} T(\text{MUX}[r_w + 1]), & \text{mirror-101, mirror} \\ T(\text{MUX}[2]), & \text{clamp, constant} \end{cases} \tag{3.20}$$

Type-2

The size of the MUX before R_{fetch}, depending on r_w, as Eq. (3.19) indicates, could drastically decrease the speed of the entire circuit for large windows. Section 3.4 suggests that if a more optimized architecture exists, it can be found by rescheduling the blue highlighted region of R'_{right} to minimize $C^{\text{new}}_{\text{mux}}$ in Eq. (3.13). Based on that, we propose an alternative schedule in Figure 3.15. Here, the blue highlighted region is restructured in a way that the leftmost register in R'_{right} can always output the proper

input for R_{fetch}. In this way, the selection before R_{fetch} can always be implemented with only a MUX[2] at the cost of additional MUX[2]s utilized between the registers in R'_{right}. An example architecture for this type is given in Figure 3.14c. The critical path and the selection cost in terms of MUXs are equal for all border handling modes and are given in Eqs. (3.21) and (3.22). Note that R'_{right}, thus the selection in between can be switched off in the interval $x = [0, W - r_w)$.

$$T^{\text{Type-2}}_{\text{CriticalPath}} = T(\text{MUX}[2]) \tag{3.21}$$

$$C^{\text{Type-2}}_{\text{mux}} = ((3 \cdot r_w + 1)\text{MUX}[2]) \cdot h \cdot k_{\text{in}} \tag{3.22}$$

3.5.3 Loop Coarsening

Our analysis on border handling remains valid for both loop coarsening architectures called F&C and C&P are presented in the following. Thereby, separated border handling types can be used as an efficient top-level architecture. However, F&C and C&P require different column selection patterns.

Fetch and Calc

A local operator can be considered as an F&C architecture with $v = 1$. Correspondingly, column selection architectures of Type-0, Type-1, and Type-2, can be used for F&C with slight modifications. Section 3.4 explains that the time interval between a local operator entering and leaving the border region is r_w cycles in raster order processing. As the local operator moves faster in the horizontal direction by the increase in v, this interval shrinks to r_{wv} as in Eq. (3.23) for the left and right borders.

$$r_{wv} = \lceil r_w/v \rceil \tag{3.23}$$

Consequently, column selection of a loop coarsening architecture with the parameters of w, v, consists of $\min(r_w, v)$ parallel column selection architectures, whose $r_w = r_{wv}$. Examples of two corner cases, which are $r_w > v$ and $r_w < v$, for Type-1 border handling in loop coarsening are shown in Figures 3.16 and 3.17. As a result, the increase in v reduces C_{mux} of border handling, whereas $C_{\text{FF}}(b)$ remains the same as Eqs. (3.24) and (3.25) indicate.

$$C^{\text{F\&C}}_{\text{mux}}(b) = \min(r_w, v) \cdot C^{\text{Type}}_{\text{mux}}(b, r_{wv}) \tag{3.24}$$

$$C^{\text{F\&C}}_{\text{FF}}(b) = C^{\text{Type}}_{\text{FF}}(b, r_w) \tag{3.25}$$

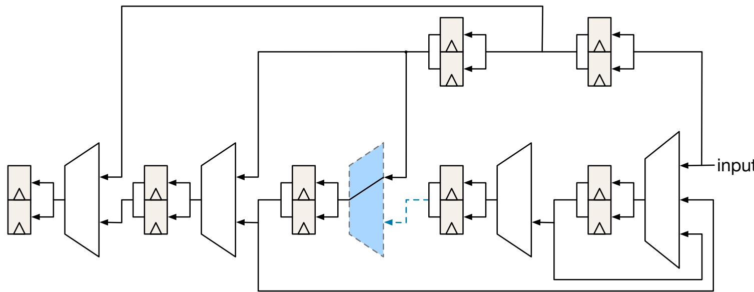
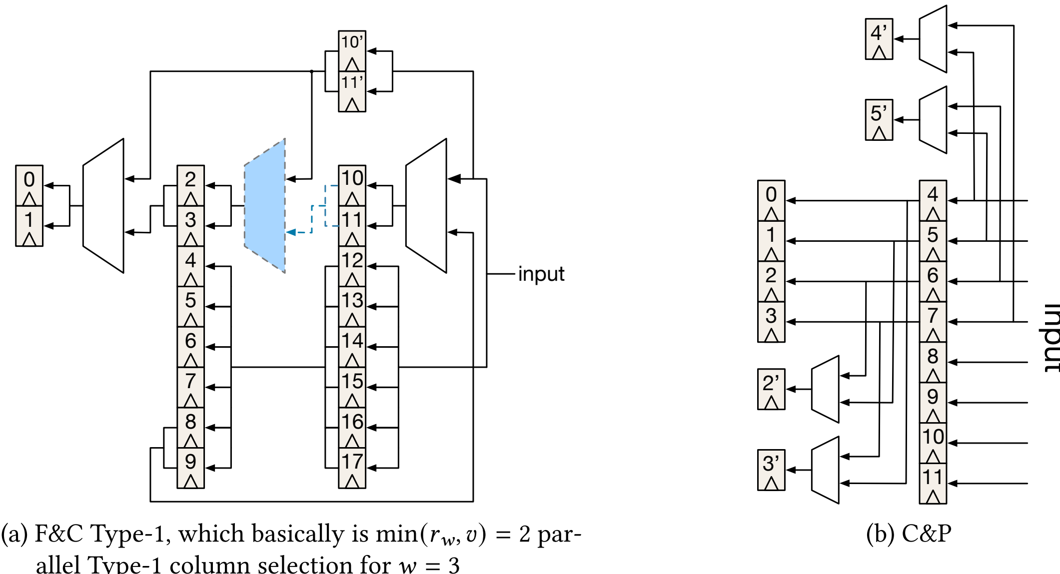


Figure 3.16: F&C Type-1 mirror border handling for $w = 9$ and $v = 2$, which basically is $\min(r_w, v) = 2$ parallel Type-1 column selection for $w = 5$. (Figure reprinted from [ÖRH⁺17b], © 2017 IEEE)



(a) F&C Type-1, which basically is $\min(r_w, v) = 2$ parallel Type-1 column selection for $w = 3$

(b) C&P

Figure 3.17: Mirror border handling for $w = 5$, $v = 8$. (Figure reprinted from [ÖRH⁺17b], © 2017 IEEE)

Calc and Pack

The regions R'_{right} and R'_{left} in Figure 3.18 illustrate the difference to F&C. As it can be seen, the first pixel of an image row is not processed in the first pixel of R_{mid}. Therefore, R_{left} cannot be initialized at $x = 0$ and additional registers R'_{left} should be used for border handling instead. An example architecture is given in Figure 3.17. However, since it consists of an additional register for each register at R_{left} and R_{right} any selection in border handling can be implemented via a MUX[2]. Hence, C_{mux}

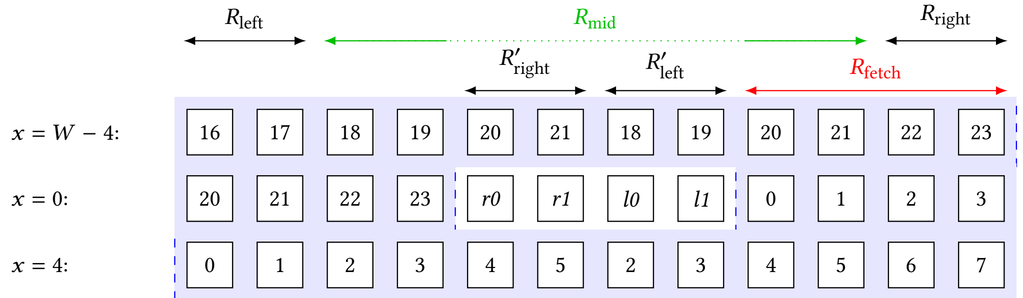


Figure 3.18: Temporal data flow for $w = 5$ with coarsening $v = 4$. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

and C_{FF} of the border handling can be defined by Eqs. (3.26) and (3.27).

$$C_{\text{FF}}^{\text{C\&P}}(b) = k_{\text{in}} \cdot h \cdot (2 \cdot r_w) \tag{3.26}$$

$$C_{\text{mux}}^{\text{C\&P}}(b) = \min(r_w, v) \cdot C_{\text{mux}}^{\text{Type-0}}(b, r_{wv}) \tag{3.27}$$

3.6 Architecture Selection

In the following, we analyze resource usage of the hardware architectures introduced in this chapter. Our goal is to provide an algorithm (shown in Algorithm 2) that selects a coarsening architecture and border handling type that requires fewer FFs, MUXs, and has lower a critical path delay (denoted by $TC_{\text{CriticalPath}}$) for the given input parameters window width (w), window height (h), border handling mode (borderMode), loop coarsening factor (v), input and output data bit width (k_{in}, and k_{out}, respectively) than any other considered architecture. The notations of these parameters are introduced in Table 3.1 for clarity. For this purpose, this section analyzes the analytical models of the considered architectures.

3.6.1 Border Handling Type Selection

Not implementing a border handling mode circuit (denoted by *none*) for the loop coarsening architectures C&P and F&C does not introduce any FF cost and MUX cost, as mathematically expressed below:

$$\begin{aligned}
\text{bmode} = \textit{none} \quad &=> \quad C_{\text{FF}}^{\text{F\&C}}(b) = C_{\text{FF}}^{\text{C\&P}}(b) = 0 \\
\text{bmode} = \textit{none} \quad &=> \quad C_{\text{mux}}^{\text{F\&C}}(b) = C_{\text{mux}}^{\text{C\&P}}(b) = 0 \\
\text{bmode} = \textit{none} \quad &=> \quad TC_{\text{CriticalPath}}^{\text{F\&C}}(b) = TC_{\text{CriticalPath}}^{\text{C\&P}}(b) = 0
\end{aligned} \tag{3.28}$$

Combining Eqs. (3.14), (3.19), (3.22), (3.25) and (3.27) reveals that the MUX costs $C_{\text{mux}}^{\text{btype}}(b)$ of all the all border handling types except Type-2 converge to the same amount where $v \geq r_w$, as Eq. (3.29) indicates.

$$\begin{aligned} r_{wv} > 1 \quad &=> \quad C_{\text{mux}}^{\text{Type-0}} \geq C_{\text{mux}}^{\text{Type-1}} \wedge C_{\text{mux}}^{\text{Type-0}} \geq C_{\text{mux}}^{\text{Type-2}} \\ r_{wv} = 1 \quad &=> \quad C_{\text{mux}}^{\text{Type-2}}(b) > C_{\text{mux}}^{\text{Type-0}}(b) = C_{\text{mux}}^{\text{Type-1}}(b) \end{aligned} \tag{3.29}$$

Merging Eq. (3.27) with Eq. (3.21) shows that the Type-2's critical path delay is always less than or equal to Type-1 and Type-0, as shown below:

$$\begin{aligned} \text{bmode} \neq \textit{none} => \quad & TC_{\text{CriticalPath}}^{\text{Type-0}}(b, r_w) \geq TC_{\text{CriticalPath}}^{\text{Type-1}}(b, r_w) \geq TC_{\text{CriticalPath}}^{\text{Type-2}}(b, r_w) \\ & TC_{\text{CriticalPath}}^{\text{Type-2}}(b, r_w) = \text{MUX}[2] \end{aligned} \tag{3.30}$$

According to Eq. (3.30), the Type-2 border handling circuit requires only one two-input MUX in its critical path, which is less than or equal to the MUX cost in border handling implementation of Type-0 and Type-1. Therefore, when synthesized Type-2 hardware circuit is expected to achieve highest clock frequency compared to other cosidered border handling implementations.

The difference in FF cost $C_{\text{FF}}(b)$ between implementing border handling types for loop coarsening architectures C&P and F&C is given in Eq. (3.31), which is derived by merging Eqs. (3.12), (3.16) and (3.26).

$$C_{\text{FF}}^{\text{C\&P}}(b) - C_{\text{FF}}^{\text{F\&C}}(b) = k_{\text{in}} \cdot h \cdot \begin{cases} 0, & \text{btype} = \text{Type-0} \\ & \vee \quad \text{bmode} = \textit{none} \\ r_w, & \text{btype} = \text{Type-1} \vee \text{Type-2} \\ & \wedge \quad \text{bmode} \neq \textit{none} \end{cases} \tag{3.31}$$

Considering Eq. (3.31) and Eq. (3.30) derives the conclusion that Type-1 border handling mode uses fewer resources (i.e., FFs and MUXs) and have smaller critical path delays than Type-0 border handling mode for both F&C and C&P loop coarsening architectures.

Combining Eqs. (3.19), (3.22) and (3.25) derives the equation below:

$$C_{\text{mux}}^{\text{Type-2}}(b) \geq C_{\text{mux}}^{\text{Type-1}}(b), \qquad \text{clamp, constant} \tag{3.32a}$$

$$C_{\text{mux}}^{\text{Type-2}}(b) - C_{\text{mux}}^{\text{Type-1}}(b) = r_w \cdot \text{MUX}[2] - \text{MUX}[r_{vw} + 1], \qquad \text{mirror, mirror-101} \tag{3.32b}$$

This analysis shows that implementing border handling modes clamp and constant by using Type-1 is more efficient than Type-2 for both F&C and C&P loop coarsening architectures in terms of resources (i.e., fewer FFs and MUXs) and critical path delay. For the border handling modes mirror and mirror-101, Type-2 uses less MUXs (and dominates Type-1) only when Eq. (3.32b)$\leq$ 0, which depends on the size of MUX gates selected by the technology mapping of the hardware synthesis tool.

Finally, combining Eqs. (3.29), (3.30) and (3.32b) shows that implementing border handling types for F&C requires fewer resources and smaller critical path than C&P, as summarized in Eq. (3.33).

$$C^{\text{C\&P}}(b) \geq C^{\text{F\&C}}(b), \qquad TC^{\text{C\&P}}_{\text{CriticalPath}}(b) \geq TC^{\text{F\&C}}_{\text{CriticalPath}}(b) \tag{3.33}$$

3.6.2 Loop Coarsening Architecture Selection

The cost of implementing the loop coarsening architectures introduced in this chapter only differs in terms of FFs, since the modifications required for extending line buffers and replicating the application datapath are same for both F&C and C&P. Unlike border handling implementations, both F&C and C&P loop coarsening architectures do not require using MUXs for data selection. Therefore, the loop coarsening architecture C&P uses less resources than F&C when its FF cost C_{FF} is smaller as denoted in the following:

$$C^{\text{C\&P}}_{\text{FF}} < C^{\text{F\&C}}_{\text{FF}} \quad => \quad C^{\text{C\&P}} < C^{\text{F\&C}} \tag{3.34}$$

Substituting Eqs. (3.3) and (3.7) into Eq. (3.34) gives the following condition for the loop coarsening architecture F&C using fewer resources than C&P:

$$\begin{aligned}
C^{\text{C\&P}}_{\text{FF}} &< C^{\text{F\&C}}_{\text{FF}} \\
0 &< C^{\text{F\&C}}_{\text{FF}} - C^{\text{C\&P}}_{\text{FF}} \\
0 &< k_{\text{in}} \cdot h \cdot (r_w + v \cdot r_{wv} + v) + C^{\text{F\&C}}_{\text{FF}}(b) - \\
&\quad (k_{\text{in}} \cdot h \cdot (2 \cdot r_w + v) + k_{\text{out}} \cdot (v - (r_w \bmod v)) + C^{\text{C\&P}}(b)) \\
0 &< k_{\text{in}} \cdot h \cdot (v \cdot r_{wv} - r_w) - k_{\text{out}} \cdot (v - (r_w \bmod v)) + C^{\text{F\&C}}(b) - C^{\text{C\&P}}(b)
\end{aligned} \tag{3.35}$$

As can be seen, the loop coarsening architecture C&P using fewer resources than F&C (where Eq. (3.35) is true) depends on many variables, including the input/output bit widths $k_{\text{in}}/k_{\text{out}}$, the coarsening factor v, local operator window height h and width, as well as to the cost of implementing image border handling.

The following expands the Eq. (3.35) for a given border handling mode (i.e., none, mirror, mirror-101, clamp, and constant).

Border Handling Mode is Undefined (None)

Substituting Eq. (3.31) into Eq. (3.35) for the *none* border handling mode (border handling is not needed) gives the equation below:

$$C^{\text{C\&P}}_{\text{FF}} \leq C^{\text{F\&C}}_{\text{FF}} \quad \text{where} \quad k_{\text{out}} \leq k_{\text{in}} \cdot h \wedge \text{btype} = \text{Type-0} \tag{3.36}$$

Eq. (3.36) indicates that the loop coarsening architecture C&P uses fewer FF than F&C (thus requires fewer are resources according to Eq. (3.34)) when the output bit width is smaller than the multiplication of local window height h and input bit width k_{in}.

Mirror-101, Mirror, Clamp, Constant

In the following, we analyze Eq. (3.35) to find the region that C&P requires fewer FFs than F&C for the border handling types Type-1 and Type-2 (for one of the border handling modes clamp, constant, mirror, and mirror-101). For this purpose, Eq. (3.35) is simplified in the following for the cases of the coarsening factor v being larger than the horizontal radius of the local operator r_w or not:

$v > r_w$: Combining Eqs. (3.29), (3.31) and (3.35) for $v > r_w$ gives the following equation:

$$\begin{aligned} v > r_w \quad \wedge \quad \text{btype} \in \{\text{Type-1}, \text{Type-2}\} \quad &=> \quad C_{\text{FF}}^{\text{C\&P}} < C_{\text{FF}}^{\text{F\&C}} \quad \text{where} \\ 0 \quad &< k_{\text{in}} \cdot h \cdot (v - r_w) - k_{\text{out}} \cdot (v - r_w) - k_{\text{in}} \cdot h \cdot r_w \\ r_w \cdot (2 \cdot k_{\text{in}} \cdot h - k_{\text{out}}) \quad &< \quad v \cdot (k_{\text{in}} \cdot h - k_{\text{out}}) \end{aligned} \tag{3.37}$$

The loop coarsening architecture C&P uses fewer FFs than F&C (thus fewer resources according to Eq. (3.34)) when Eq. (3.37) satisfies.

$v \leq r_w$ Merging Eqs. (3.31), (3.33) and (3.35) for $v \leq r_w$ gives the following equations:

$$\begin{aligned} v \leq r_w \quad \wedge \quad \text{btype} \in \{\text{Type-1}, \text{Type-2}\} \quad &=> \quad C_{\text{FF}}^{\text{F\&C}} < C_{\text{FF}}^{\text{C\&P}} \quad \text{where} \\ 0 \quad < \quad k_{\text{in}} \cdot h \cdot (v \cdot r_{wv} - r_w) &- k_{\text{out}} \cdot (v - (r_w \bmod v)) - k_{\text{in}} \cdot h \cdot r_w \\ 0 \quad &< \quad k_{\text{in}} \cdot h \cdot (v \cdot r_{wv} - 2 \cdot r_w) - k_{\text{out}} \cdot (v - (r_w \bmod v)) \\ k_{\text{out}} \cdot (v - (r_w \bmod v)) \quad &< \quad k_{\text{in}} \cdot h \cdot (v \cdot r_{wv} - 2 \cdot r_w) \end{aligned} \tag{3.38}$$

The left-hand side of Eq. (3.38) is always positive while the right-hand side is always negative for $v \leq r_w$. Therefore, F&C requires fewer resources than C&P when $v \leq r_w$.

3.6.3 Architecture Selection Algorithm

Algorithm 2 calculates the architecture that requires fewer resources and shorter critical path according to analysis given in Section 3.6 as well as the column selection architecture discussed in Section 3.4.

3.7 Evaluation and Results

In this section, we evaluate the FPGA implementation results of the considered loop coarsening and border handling architectures. We individually evaluate the

Algorithm 2: Architecture selection algorithm

input : w, h, borderMode, v, k_{out}, k_{in}, designGoal
output: BorderHandlingPattern ∈ {Type-1, Type-2}
CoarseningArch ∈ {F&C, C&P}

```
1  func selectArchitecture(BorderHandlingPattern, CoarseningArch
2                          w, h, borderMode, v, k_out, k_in, designGoal)
3      r_w = ⌊w/2⌋
4      if borderMode = UNDEFINED then
5          if k_out < k_in · h then
6              CoarseningArch ← Calc and Pack (C&P)
7          else
8              CoarseningArch ← Fetch and Calc (F&C)
9          end
10         BorderHandlingPattern ← none
11     else
12         if r_w · (2 · k_in · h − k_out) < v · (k_in · h − k_out) then
13             CoarseningArch ← Calc and Pack (C&P)
14         else
15             CoarseningArch ← Fetch and Calc (F&C)
16         end
17         if borderMode = (CLAMP ∨ CONSTANT) then
18             BorderHandlingPattern ← Type-1
19         else
               // borderMode = (MIRROR ∨ MIRROR-101)
20             if (designGoal = speed) ∨ (Eq. (3.32b) = true) then
21                 BorderHandlingPattern ← Type-2
22             else
23                 BorderHandlingPattern ← Type-1
24             end
25         end
26     end
27 end
```

two loop coarsening and border handling architectures in Sections 3.7.1 and 3.7.2, respectively. The additional pipelining overhead that a higher target logic speed might introduce to these architectures is evaluated in Section 3.7.3. We use Xilinx Vivado HLS to evaluate the synthesis results of our proposed implementations when an HLS tool is used. In order to examine the analytical models of the proposed architectures, we investigate the synthesis of a single kernel algorithm, the mean filter, and subtract the cost of the data path from the total cost using estimation

results. Confirming analytical models in this way facilitates the confidence that we avoid reporting any overhead caused by the HLS tool's heuristics. Finally, we compare the FPGA implementation results with the estimation results to examine their validity in Section 3.7.4.

3.7.1 Coarsening Types

In this section, we compare the coarsening architectures proposed in Section 3.3. As already discussed, F&C is an extension of Schmid's loop coarsening technique that avoids redundant registers. Therefore, we compared F&C with C&P in Figures 3.19 and 3.20. Moreover, some of these results are given in Table 3.2. Algorithm 2 suggests using C&P in case of *none* border handling mode for any v and r_x. Moreover, it can be derived from Eq. (3.36) that C&P should use less registers than F&C in this case according to $((v - (r_w \bmod v)) \bmod v) \cdot (k_{\text{in}} \cdot h - k_{\text{out}})$. F&C uses 960 more registers (48 %) than C&P for $w = 5$ and $v = 32$, as given in Table 3.2. The improvement reaches up to 2160 registers (50 %) for $w = 11$ and $v = 32$, as shown in Figure 3.19. This validates our coarsening equations, and reveals that thousands of registers can be saved with an appropriate architecture even for an application consisting of a simple single kernel.

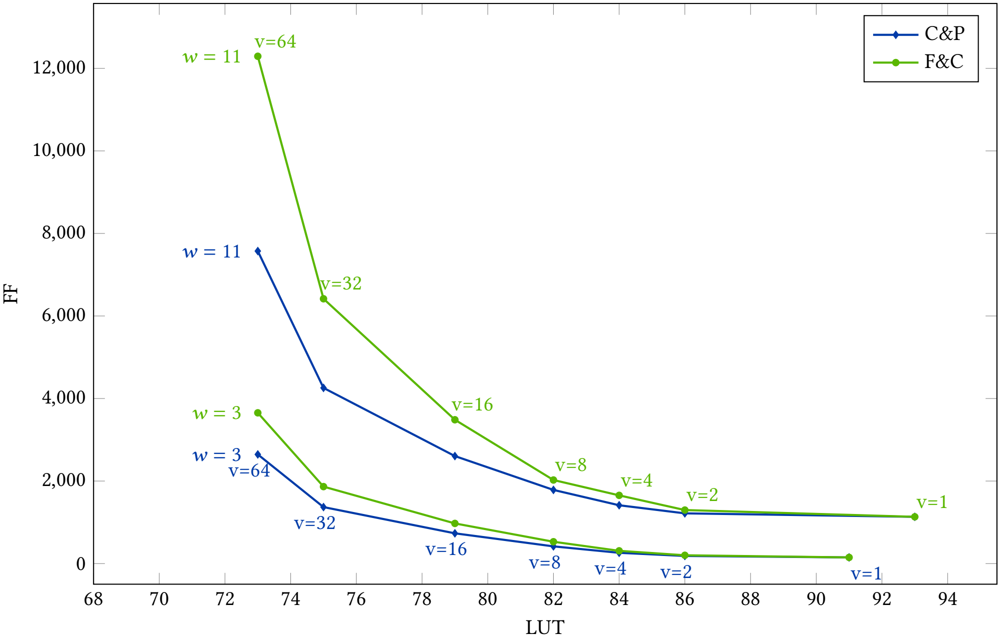


Figure 3.19: HLS estimation results of the proposed coarsening architectures for different kernel sizes and 5.0 ns target clock period (no border handling). (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

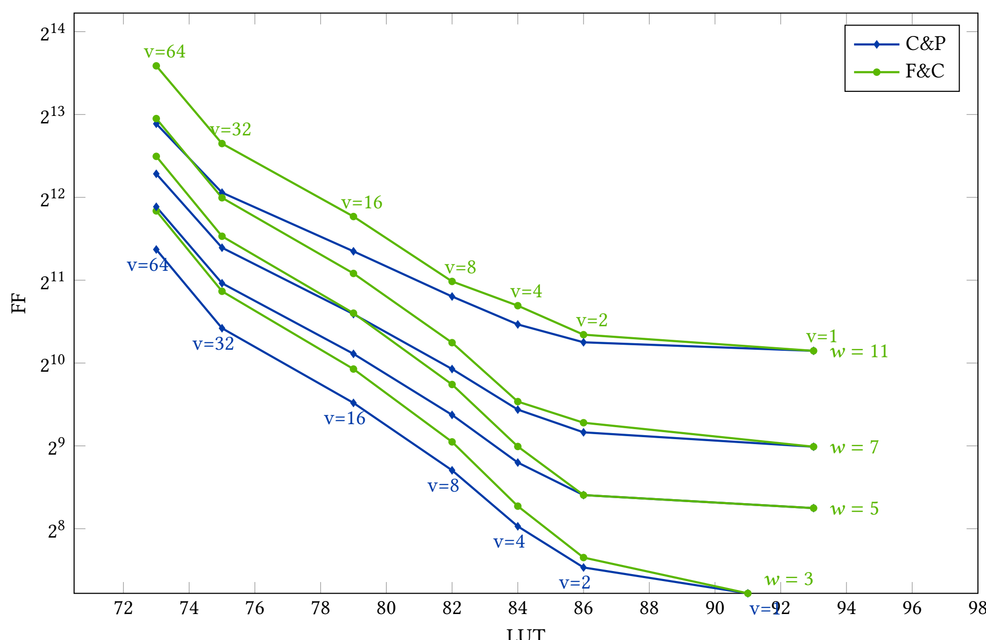


Figure 3.20: HLS estimation results of the proposed coarsening architectures for different kernel sizes and 5.0 ns target clock period (no border handling). (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

3.7.2 Border Handling Architectures

Figures 3.21 and 3.22 show that separated border handling significantly improves the naive approach. This indicates that synthesis tools are not able to separate column selection and row selection automatically (see Section 3.4 for the analysis of redundancy). As Eq. (3.29) indicates, all the discussed separated border handling architectures except Type-2 converge to the same design point for $v \geq r_w$ as shown in Figure 3.22. On the other hand, Figure 3.22 clearly shows that Type-1 uses considerably less logic than Type-0 for $v < r_w$. Investigating the generated HDL codes reveals that Vivado HLS optimizes the Type-0 column selection by eliminating the blue-colored registers in Figure 3.14a. This can be achieved with a simple analysis that detects the registers set via the same wire. Considering Eqs. (3.16) and (3.17) with this optimization Type-1 uses $h \cdot k_{in}\ (r_w - 1)$ fewer registers than Type-0 for the sliding window as expected. Speed optimization introduced via Type-2 highly depends on technological mapping, since MUX[2] is mapped via LUTs in FPGA. Table 3.2 shows that Eq. (3.32b) does not satisfy in our evaluation environment, therefore Type-1 uses fewer LUTs than Type-2. Yet Type-2 uses fewer registers, which makes it a different design point.

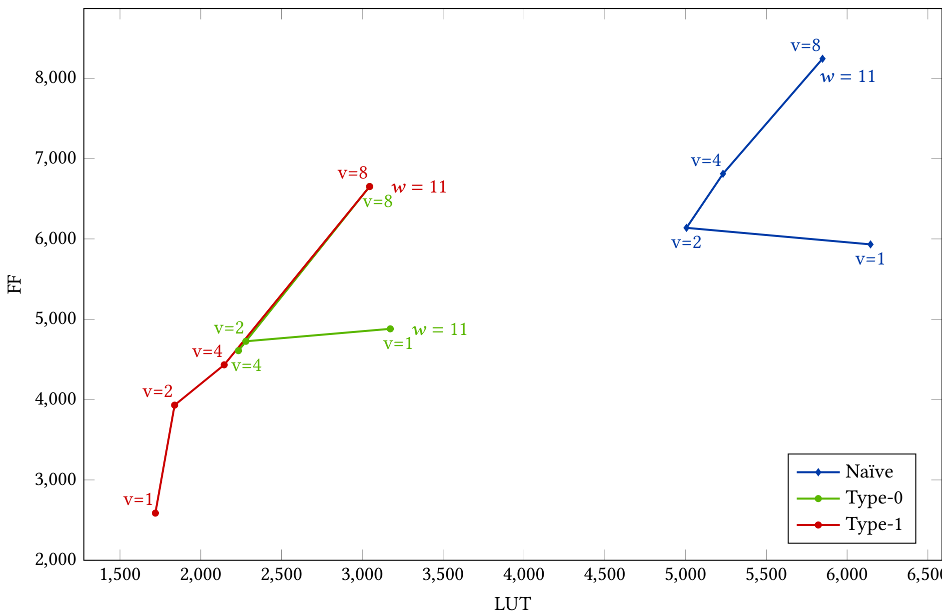


Figure 3.21: HLS estimation results of the proposed mirror border handling architectures for a 11×11 kernel with F&C coarsening. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

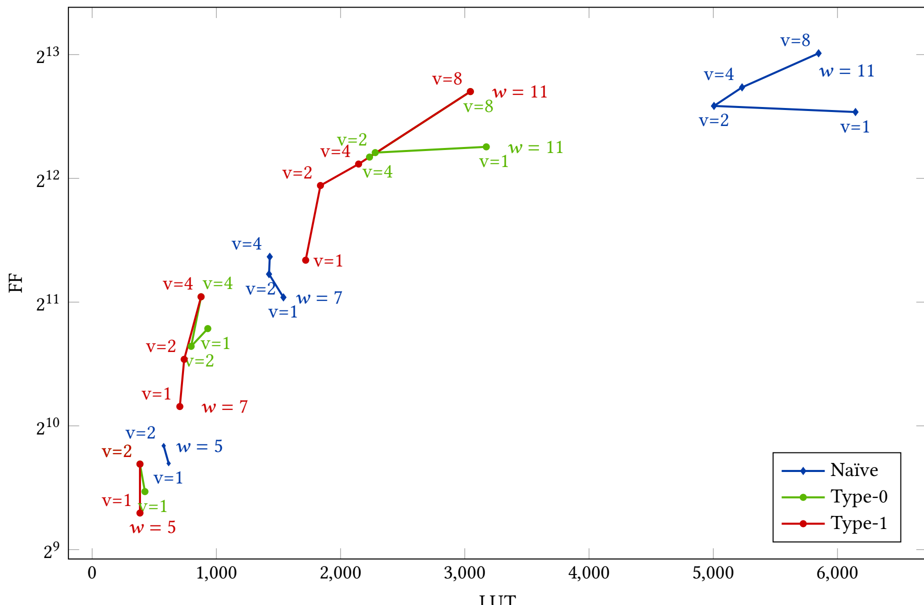


Figure 3.22: HLS estimation results of the proposed mirror border handling architectures for different kernel sizes F&C coarsening. (Figure reprinted from [ÖRH+17b], © 2017 IEEE)

Table 3.2: HLS estimation results for a local operator and implementation results for a Mean Filter. T_{clk_tar} (input clock period constraint given to HLS tool) denotes the target critical path delay. Defining a lower T_{clk_tar} requires the HLS tool to increase the level of structural pipelining to achieve a higher clock frequency.

Parameters			Estimation (T_{clk_tar} = 20 ns)				Estimation (T_{clk_tar} = 3.1 ns)				Implementation (T_{clk_tar} = 3.1 ns)						
v	w/h	Coars.	BRAM	FF	LUT	T_{clk_es}	BRAM	FF	LUT	T_{clk_es}	SLICE	BRAM	FF	LUT	DSP	T_{clk_syn}	T_{clk_imp}
1	5	C&P	4	304	93	13.50	4	378	93	3.10	152	4	600	270	29	2.48	2.55
1	5	F&C	4	304	93	13.50	4	378	93	3.10	139	4	600	269	29	2.48	2.61
2	5	C&P	4	339	86	14.43	4	446	88	3.09	215	4	873	448	14	2.52	2.63
2	5	F&C	4	339	86	14.43	4	446	88	3.09	222	4	873	449	14	2.52	2.46
8	5	C&P	8	663	82	14.43	8	954	84	3.06	675	8	2589	1565	6	2.39	2.56
8	5	F&C	8	855	82	14.43	8	1146	84	3.06	603	8	2781	1566	6	2.39	2.74
32	5	C&P	32	1995	75	14.43	32	3045	77	3.03	1951	32	9256	5367	6	2.38	2.83
32	5	F&C	32	2955	75	14.43	32	4005	77	3.03	2023	32	10216	5367	6	2.38	3.09

(a) Coarsening Architectures

Parameters			Estimation (T_{clk_tar} = 20 ns)				Estimation (T_{clk_tar} = 3.1 ns)				Implementation (T_{clk_tar} = 3.1 ns)					
v	w/h	BH Patt.	BRAM	FF	LUT	T_{clk_es}	BRAM	FF	LUT	T_{clk_es}	SLICE	BRAM	FF	LUT	T_{clk_syn}	T_{clk_imp}
1	5	Naïve	4	471	575	13.5	4	643	576	3.10	213	4	879	533	2.63	2.67
1	5	Type-0	4	438	381	13.5	4	522	385	3.10	192	4	772	422	2.53	2.65
1	5	Type-1	4	398	341	13.5	4	482	345	3.10	176	4	732	419	2.52	2.58
1	5	Type-1	4	403	459	13.5	4	487	468	3.10	179	4	742	475	2.52	2.53
2	5	Naïve	4	495	538	14.4	4	664	542	3.09	288	4	1182	741	2.46	2.65
2	5	Type-0	4	432	344	14.4	4	565	349	3.09	268	4	1053	625	2.52	2.65
2	5	Type-1	4	432	344	14.4	4	565	349	3.09	244	4	1053	626	2.52	2.60
2	5	Type-1	4	435	501	14.4	4	569	511	3.09	262	4	1061	700	2.53	2.80
1	7	Naïve	6	855	1443	15.0	6	1434	1446	3.10	482	6	1976	1174	2.55	2.74
1	7	Type-0	6	806	867	15.0	6	1332	871	3.10	415	6	1843	878	2.56	2.85
1	7	Type-1	6	693	642	15.0	6	1107	646	3.10	381	6	1620	741	2.56	2.83
1	7	Type-1	6	698	808	15.0	6	885	817	3.10	316	6	1409	851	2.52	2.83
2	7	Naïve	6	957	1308	15.9	6	1589	1339	3.09	653	6	2715	1595	2.53	2.70
2	7	Type-0	6	862	730	15.9	6	1496	735	3.09	595	6	2570	1219	2.52	2.95
2	7	Type-1	6	806	674	15.9	6	1384	679	3.09	554	6	2459	1189	2.52	2.76
2	11	Type-1	6	923	1287	17.2	6	1107	1298	16.1	514	6	2194	1405	2.53	2.94
1	11	Naïve	10	2033	5390	16.6	10	4324	5393	3.10	1352	10	5601	3852	2.55	2.87
1	11	Type-0	10	1952	2997	16.6	10	3549	3018	3.10	1028	10	4781	2627	2.55	2.95
1	11	Type-1	10	1597	1583	16.6	10	1892	1594	3.45	711	10	3104	1883	2.55	2.67
1	11	Type-1	10	1601	1845	16.6	10	1891	1862	3.10	685	10	3114	2062	2.52	2.90
2	11	Naïve	10	2184	4562	16.7	10	4597	4566	3.09	1607	10	6839	4366	2.56	2.91
2	11	Type-0	10	2025	2160	16.7	10	3372	2228	3.09	1286	10	5663	3183	2.53	2.82
2	11	Type-1	10	1760	1719	16.7	10	2843	1787	3.09	1204	10	5136	2956	2.56	2.95
2	11	Type-1	10	1989	3564	25.3	10	2230	3639	25.3	1221	10	4537	3724	2.53	2.85

(b) Border Handling Architectures

3.7.3 Effects of Target Speed Constraint

Vivado HLS generates different HDL codes for different clock speed targets. We evaluated the pipelining overheads of the considered architectures using the target speeds of 50 MHz and 300 MHz for a ZYNQ-zc706 FPGA. Estimation results revealed that the number of registers and MUXs match our equations for the low-speed target. This can be seen by Comparing resource utilization of different coarsening factors in Table 3.2 Likewise, comparing the estimation results of architectures that have

the same loop coarsening parameters but different target speeds shows that the improvement gets more significant for higher speed constraints.

3.7.4 Implementation Results

Reporting implementation results is a good practice to eliminate estimation errors. On the other hand, using estimation reports make it possible to subtract the calculation overhead of a local operator and distinguish the selection cost of border handling among other MUXs. We observed that the estimation results, from Vivado HLS v2016.3, are reliable enough for our implementations other than Type-2. As a measure of this reliability, Table 3.2 consists of implementation results for the provided estimation results. It can be seen that the estimation differences between coarsening and border handling architectures reflect the implementation results with only very minor deviations. On the other hand, we investigated the results of Type-2 only through its implementation results in Section 3.7.2.

3.8 Conclusion

Being inspired by Schmid's [SRH+15] architecture for loop coarsening, we proposed two new loop coarsening architectures. While the first one always uses fewer resources than [SRH+15], the architecture using fewer resources depends on the local operator's parameters (i.e., stencil width w, stencil height h, coarsening factor v, border handling mode (e.g., mirror, clamp), input and bit-widths k_{in} and k_{out}). Moreover, as a novel contribution, we integrated image border handling problem into loop coarsening hardware architecture design.

We conducted a systematic analysis for the minimal resource utilization on image border handling and proposed novel architectures based on that analysis. Previous works on image border handling [Bai11b; RN16] uses more resources than our Type-1, but in none of them switching off additional registers is considered, although utilizing an additional MUX for it. Moreover, we proposed faster architectures (shorter critical path) for border handling modes mirror and mirror-101, and discussed how other Pareto-optimal architectures can be designed based on the analysis. Aside from that, their architectures do not consider loop coarsening at all.

Finally, we analyzed the resource usage (i.e., FF, MUX) of every architecture discussed in this chapter and provided an algorithm selecting the best loop coarsening and border handling architectures for given parameters of a local operator. As a side contribution, we investigated Vivado HLS implementations of our architectures as well as the naïve approach. Thus, we show that describing the structure (underlying architecture) of a hardware implementation can significantly improve the quality of hardware (in terms of resource usage, latency, and speed) synthesized by HLS.

Part II

Lifting High-Level Synthesis for Image Processing

4 FPGA-Based Accelerator Design from a Domain-Specific Language

This chapter shows that raising the abstraction level in HLS by clearly separating the concerns of algorithm description (*what?*) from its implementation (*how?*) solves productivity, performance, and portability issues of current HLS. In particular, we present a source-to-source compiler for an image processing DSL [ÖRH+16; RÖM+17b] and an image processing library [ÖRH+17a] that relies on C/C++ metaprogramming techniques to generate highly optimized, target-specific FPGA implementations from high-level, functional abstractions used for the description of an algorithm.

Our approach leverages the benefits of general-purpose HLS tools. That is, we generate code as input to existing commercial HLS tools to utilize their highly-studied and optimized scheduling, allocation, and binding techniques. These tools and techniques produce satisfactory hardware synthesis results only when programmers specify hardware-favorable implementations (*how the circuit works*) using hardware-centric pragmas. Contrarily, our approach lets users describe their algorithm by language constructs or library functions without considering its implementation for the specific domain of image processing algorithms and applications. As a result, it provides performance portability across different HLS tools and computing platforms such as GPUs and CPUs.

4.1 Introduction

Many image processing applications have stringent performance, energy efficiency, and power requirements. FPGAs have a great potential for improving throughput per watt in many applications. However, unlike traditional software programming, one needs to design hardware for FPGA-based acceleration. This is a time-consuming task and requires hardware expertise. HLS has received much attention over 30 years and has become much more sophisticated in the last decade. Modern HLS tools allow users to use C-based languages, such as the Open Computing Language (OpenCL) and C++, but they expect them to describe the behavior of application-specific hardware.

Furthermore, hardware designers still face challenges when handling data exchange and device communication when using the FPGA as a co-processor or dedicated accelerator in a heterogeneous system.

OpenCL provides a standard API to support communication between a host and a device. Hence, it allows controlling the hardware accelerator generated by HLS in a heterogeneous system (e.g., consisting of a CPU and a GPU), similar to many-core programmable devices. The HLS tool automatically synthesizes control hardware for system interfaces (such as PCIe) and deploys the necessary drivers. This way, a complete system is automatically generated from a higher-level hardware description. However, OpenCL is considered to be a platform-specific language since its programming paradigm indicates a specific memory hierarchy (i.e., consisting of local and global memory) and the execution of a multi or many-core processor. It follows a data-parallel programming paradigm, meaning that source codes, so-called *kernels*, usually only describe computations processed by a single thread. All computations are data-independent, and the creation and scheduling of threads are initiated and managed by the OpenCL runtime. The number of threads spawned depends on a range (1D, 2D, or 3D) specified by the developer. Kernels meant to perform across a specified range are called *NDRange kernels*. In OpenCL, NDRange kernels are designed to exploit parallel compute resources of programmable processor. They are executed exactly one time for each point in the NDRange index space. This unit of work for each point in the NDRange is called a work-item. Unlike for loops in C, where loop iterations are executed sequentially and in-order, an OpenCL runtime and device is free to execute work-items in parallel and in any order. This NDRange paradigm differs from hardware design techniques that leverage the capabilities of FPGAs, where the memory architecture and data path can be tailored to the application by reconfiguration. Therefore, users of the HLS tools need to specify a single-work item kernel and describe a hardware implementation similar to writing a sequential code in C++ to achieve high-quality synthesis results.

Despite pursuing new programming methodologies for many-core, multi-threading, or vector architectures, the FPGA community mostly tries to advance the design techniques from existing programming languages that are sequential or developed for other computing platforms. To handle the complexity of future systems and to increase development productivity, even without hardware design expertise, we believe that the next step for HLS requires an increased level of abstraction on the language side combined with modern metaprogramming approaches. One solution to solve this challenge is domain-specific languages and libraries. DSLs allow raising the level of abstraction and separating the algorithmic description from hardware-specific transformations such as parallelization, vectorization, or memory-related optimizations. This facilitates fast prototyping while achieving high performance on different hardware platforms from the same high-level description. Furthermore, it relieves the programmer from hardware-specific optimizations, which requires

architecture expertise and is often error-prone, non-portable, and time-consuming.

As a solution, in this chapter, we leverage the algorithm description to an even higher language level, such as a Domain-Specific Language (DSL) or an image processing library, by using metaprogramming techniques. The main contributions of this chapter are summarized as follows[1]:

- We present a novel OpenCL (source-to-source compiler) backend [ÖRH+16] that generates input code for the Intel HLS tool (OpenCL SDK for FPGAs) from an image processing DSL (namely Hipacc [MRH+16]), initially developed for GPUs. Our backend applies various transformations using domain knowledge: It leverages compiler transformations introduced in previous work [RSH+14] (Hipacc's HLS backend for Xilinx Vivado HLS), such as creating a streaming pipeline, but generates OpenCL code optimized for Intel OpenCL SDK for FPGAs [ÖRH+16; RÖM+17b]. Furthermore, it uses our novel loop coarsening and border handling techniques [ÖRH+17b] introduced in Chapter 3 to generate circuits that deliver high performance. We demonstrate that implementations produced by our compiler backend are on par with the handwritten applications provided by Intel (when our implementation is not parallelized by loop coarsening) and significantly better when compared with the parallelization intrinsics offered by the Intel HLS compiler.

- We alleviate the tasks of Hipacc's HLS backends [RSH+14; ÖRH+16] by integrating metaprogramming libraries as part of its code generation flow. More specifically, we utilize metaprogramming concepts of C++ to build a modular and highly parameterizable function library [ÖRH+17a] for describing image processing applications. Our proposed library increases the productivity of HLS users by providing high-level abstractions (e.g., point, local, and global operators) and key hardware design elements (e.g., line buffers, sliding window, streaming elements) to describe their hardware in a modular way [ÖRH+17a]. It is highly optimized with hardware design techniques such as bit-level optimizations, deep pipelining, and our novel parallelization techniques explained in Chapter 3.

- Since Hipacc is able to generate high-performance code for CPUs and GPUs as well, we show that our approach allows using the same application description to target drastically different computing platforms.

[1] The contents of this chapter are based on and partly published in [ÖRH+16], which has appeared in the Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL), [ÖRH+17a], which has appeared in the Proceedings of the Fourth International Workshop on FPGAs for Software Programmers (FSP), and [RÖM+17b] which has appeared in the Proceedings of the International Conference on Computer Aided Design (ICCAD).

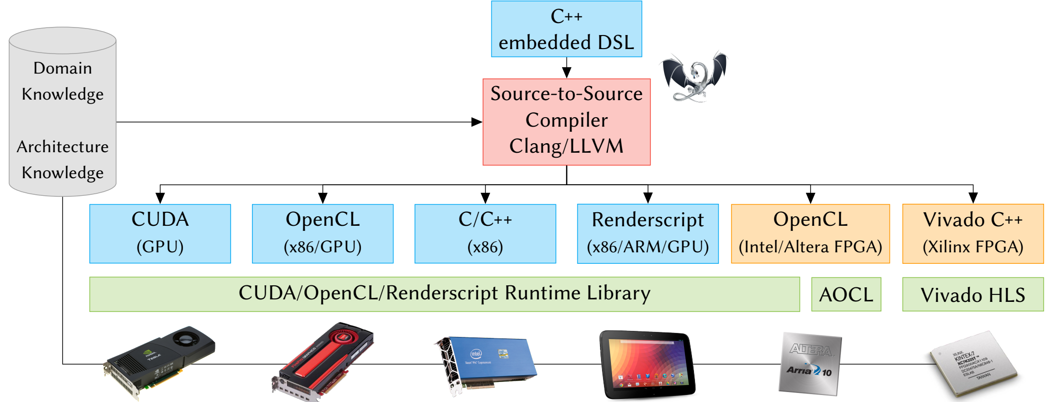


Figure 4.1: Overview of the Hipacc framework and its target architectures. The hardware targets that this thesis contributes (in [ÖRH⁺16; ÖRH⁺17a], and partially in [RÖM⁺17b; RÖH⁺18]) are highlighted in orange. (Figure reprinted from [RÖM⁺17b], © 2017 IEEE)

The remainder of this chapter is structured as follows: Section 4.2 gives a brief overview of the Hipacc DSL and Section 4.3 presents our code generation backend. Then, Section 4.4 discusses how the tasks of Hipacc's HLS compilers are alleviated by using such a library and Section 4.5 show the image processing library developed by using C++ metaprogramming. Finally, Section 4.6 evaluates the performance of our proposed compiler backend and image processing library.

4.2 Background: The Hipacc Framework

In this work, we use the Heterogeneous Image Processing Acceleration (Hipacc) framework to decouple the description of image processing algorithms from their low-level implementation details. Hipacc [MRH⁺16; RÖM⁺17b] comprises an open-source image processing DSL embedded in C++ and a source-to-source compiler. It was initially designed to target GPUs from Nvidia and AMD solely [MHT⁺12; MDH⁺19] and has undergone numerous extensions over the years. Hipacc's image processing DSL is shallowly embedded into C++, where the language components of the DSL are built using C++ classes. Any standard C++ compiler can be used to compile a Hipacc application, e.g., for quick prototyping purposes. Hipacc's source-to-source compiler, on the other hand, will produce highly optimized target code for the chosen accelerator platform (e.g., a CPU or a GPU) if it is employed.

Figure 4.1 shows an overview of the framework and its target architectures. Hipacc can generate code for the Compute Unified Device Architecture (CUDA), OpenCL for GPUs, Renderscript, and a specific kind of C++ suitable for Xilinx's Vivado HLS. In this thesis, we add a new backend emitting code for Intel's FPGA SDK for

OpenCL [ÖRH+16], and redesign Hipacc's Xilinx Vivado HLS backend to significantly improve performance [ÖRH+17a].

4.2.1 Overview of Hipacc DSL

Figure 4.2 illustrates a description of a Gaussian filter in Hipacc (its code is shown in Listing 4.3 after the explanation of the DSL). Language components such as `Image` and `Kernel` describe an application in an abstract way without exposing data layout and low-level platform-specific declarations to users. That is, a Hipacc application describes image processing applications as a directed acyclic graph (DAG) of point, local, and global operators without writing C++ loops and arrays. This allows for capturing the inherent parallelism at a higher level, thus generating highly-optimized target-specific code.

In the following, we briefly summarize the primary language components of Hipacc, proposed by Membarth et al. [Mem13; MRH+16]. We refer to [Mem13; MRH+16] for a more detailed explanation.

4.2.2 Data Storage

Hipacc uses `Image` and `Mask` structures to store pixel and filter coefficients, respectively.

Definition 4.1 (Image)

In Hipacc, the data storage for a digital image's pixels is referred to as an `Image`. It represents a two-dimensional data structure, where any standard data type (such as `float` or `unsigned char`) can be used as a pixel type. Constructors of an `Image` are listed below:

```
Image<pixel_t>(const int width, const int height)

Image<pixel_t>(const int width, const int height, pixel_t* pixels)
```

The pixel data type is represented by `pixel_t`. An `Image` can be created using a contiguous memory block (e.g., a C++ array) or as empty. A memory allocation and zero initialization will implicitly be done for an empty image when needed.

Definition 4.2 (Mask)

A `Mask` holds constants on a two-dimensional rectangular area. It can be defined by using one of the constructors shown below:

```
Mask<mask_t>(const int width, const int height)

Mask<mask_t>(mask_t coefficients[height][width])
```

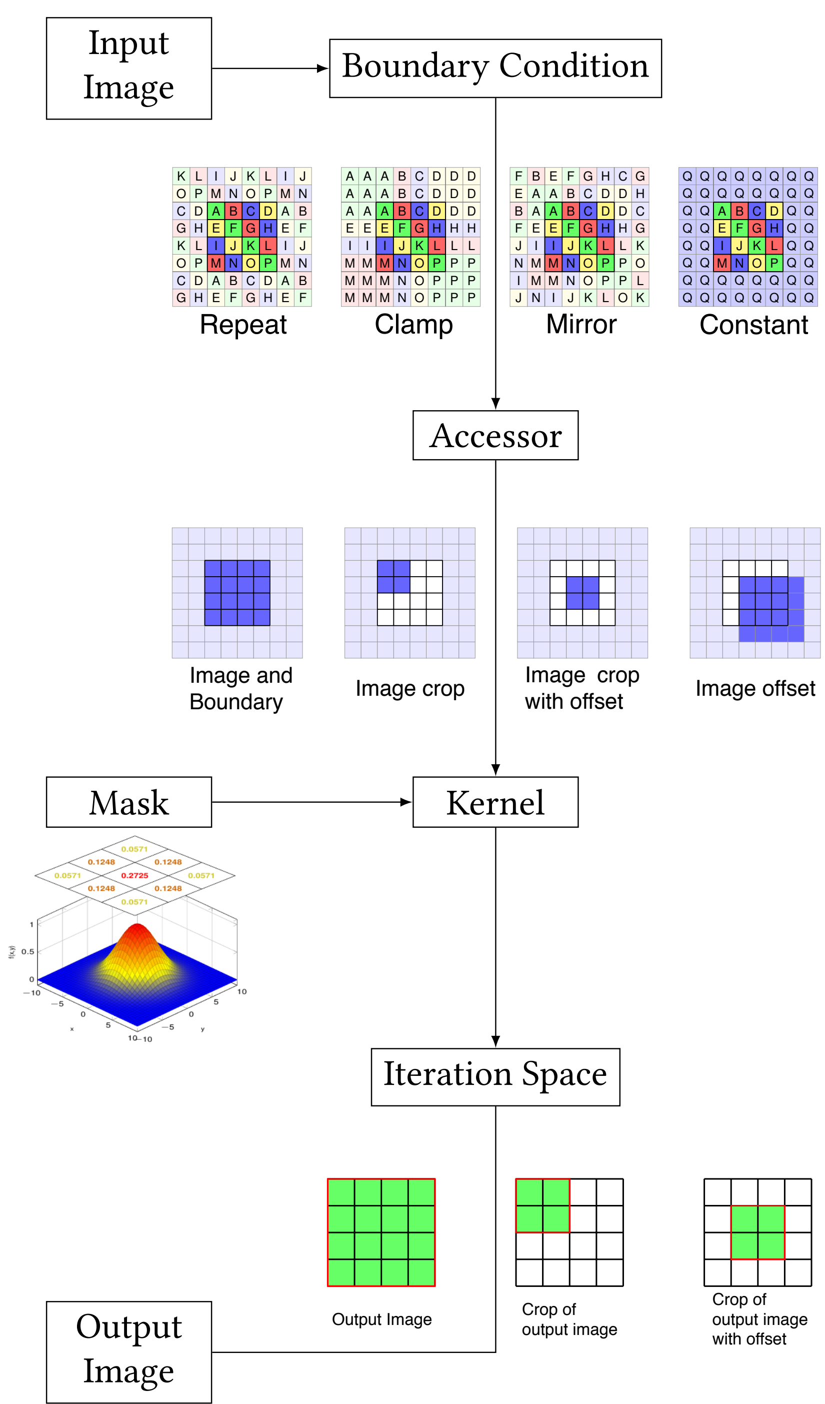


Figure 4.2: Illustration of an example Hipacc application (its code is explained in Section 4.2.6), where an application is described as a dependency graph of domain-specific abstractions such as *Image* and *Mask*. A local operator is defined as a *Kernel* object. A *Kernel* can read and write images only by using an *Accessor* and an *Iteration Space*, respectively. Users can select a border handling mode through an *Accessor*. A *Mask* is used for storing filter coefficients and defining a stencil pattern.

Table 4.1: Supported border handling modes in Hipacc (Boundary type).

Enumeration Type	Border handling mode
UNDEFINED	No border handling (default)
CONSTANT	Constant value
CLAMP	Clamping
MIRROR	Mirroring at image border
MIRROR-101	Mirroring at last valid pixel
REPEAT	Repeating

The dimensions of a mask must be constant, which could be deduced from a two-dimensional array when given as an input parameter. An example application would be a filter using a Mask for the coefficients.

4.2.3 Read/Write Operations on Images

Hipacc captures memory access patterns by allowing users to access images and masks only using special data structures, namely Accessor and Iteration Space. Similarly, the DSL offers BoundaryCondition as a language component to prevent users from introducing conditional descriptions for handling out-of-bound access.

Definition 4.3 (Boundary Condition)

As explained in Chapter 2, image processing algorithms virtually extend an input image to avoid out-of-bound accesses, often using a well-known pattern (so-called *boundary mode*). Hipacc offers a language component called BoundaryCondition. It can be created by one of the following constructors:

```
BoundaryCondition<pixel_t>(Image<pixel_t> img, const int size,
                           const Boundary bmode = UNDEFINED,
                           const pixel_t val = 0)

BoundaryCondition<pixel_t>(Image<pixel_t> img,
                           const int width, const int height,
                           const Boundary bmode = UNDEFINED,
                           const pixel_t val = 0)

BoundaryCondition<pixel_t>(Image<pixel_t> img, MaskBase stencil,
                           const Boundary bmode = UNDEFINED,
                           const pixel_t val = 0)
```

The supported boundary modes are listed in Table 4.1. A constant value of a pixel type (denoted as pixel_t above) could be given as an additional parameter when CONSTANT mode is selected (zero value is set otherwise). The default mode

is UNDEFINED, allowing out-of-bound access. This might be desirable when an application is run on a smaller portion of an algorithm (i.e., on a region of interest (ROI)).

An image and dimensions of a local window (one-dimensional or two-dimensional) must be given as input to create a BoundaryCondition. Alternatively, the last constructor takes an image and a MaskBase object (i.e., a Mask or a Domain), which allows deducing the dimensions of the local window.

Definition 4.4 (Accessor)

Hipacc's image processing operators can only read an image using an Accessor. Its constructors are listed below:

```
Accessor<pixel_t>(Image<pixel_t> img)

Accessor<pixel_t>(Image<pixel_t> img,
                  const int width, const int height,
                  const int offset_x, const int offset_y)

Accessor<pixel_t>(BoundaryCondition<pixel_t> bmode)

Accessor<pixel_t>(BoundaryCondition<pixel_t> bmode,
                  const int width, const int height,
                  const int offset_x, const int offset_y)
```

An Accessor allows describing an ROI over the input image, thus running an application only on a smaller rectangular portion of the input image. It can be created solely by using an Image or a BoundaryCondition when the whole image is used as input.

Definition 4.5 (Iteration Space)

An IterationSpace allows writing to an image. Its constructors are listed below:

```
IterationSpace<pixel_t>(Image<pixel_t> img)

IterationSpace<pixel_t>(Image<pixel_t> img,
                        const int width, const int height,
                        const int offset_x, const int offset_y)
```

Similar to an Accessor, an IterationSpace allows for selecting an ROI on the output image.

4.2.4 Stencil Patterns

As introduced in Chapter 2, local operators produce an output pixel using neighboring pixels on a local window. A programmer can describe the shape of the neighborhood

(i.e., stencil pattern) by using a `Domain` or a `Mask`.

Definition 4.6 (Domain)

A `Domain` describes a stencil pattern on a two-dimensional rectangular area. Its constructors are listed below:

```
Domain(unsigned char stencil[height][width])

Domain(Mask<mask_t> mask)

Domain(const int width, const int height)
```

A domain can be created using a two-dimensional array or a `Mask`. The stencil pattern is defined by the *nonzero* elements. Alternatively, a full selection of a rectangular stencil can be created using the latest constructor above (i.e., every coordinate has a nonzero value). Later, the configuration can be changed by using the index operator shown below:

```
Dom(1, 2) = 0;
```

Similar to a `Domain`, nonzero values of a `Mask` define the stencil pattern when used in a Hipacc operator.

4.2.5 Describing Computational Patterns

Hipacc allows for describing a point, local, or global operator's computations using user-defined C++ classes derived from a `Kernel`.

Definition 4.7 (Kernel (Class))

In Hipacc, programmers describe computations on images using high-level abstractions. For this purpose, a `Kernel` class must be inherited as shown below:

```
class UserDefinedComputation : public Kernel <uchar > {
  // ...
  public:
    UserDefinedComputation(
        Accessor<uchar> &input,    // input image
        IterationSpace<uchar> &out, // output image
        // ...                      // Domain, Mask
        ) : {/* ... */}

    // override the virtual kernel function (of the Kernel class)
    void kernel_method() {
        // describe computation using Hipacc's abstractions
    }

    // optional: additional functions for global operators
};
```

Table 4.2: Hipacc's `Kernel` methods used for describing computations on images.

Kernel Method	Operator Type using the `Kernel` method
`kernel()`	point, local, global
`reduce()`	global reduction

Inputs and outputs of computations are defined by setting the constructor of the `Kernel` class. Users describe a class (e.g., `UserDefinedComputation` above) by overriding the implementation of a special member function of `Kernel` (denoted by `kernel_method`). These functions are listed in Table 4.2.

4.2.6 Point and Local Operators

Point and Local operators are described solely using the `kernel` method.

Definition 4.8 (kernel)

Hipacc carries out the computations described in `kernel()` method for every pixel in the output image (which is written by an `IterationSpace`). Its declaration is shown below:

```
void Kernel<pixel_t>:: kernel()
```

The result of a computation described in `kernel` is then written to an `IterationSpace` by using `output()` function as shown in Listing 4.1.

Listing 4.1 shows the description of a point operator `Kernel` in Hipacc. The computation is described solely for one input pixel in an abstract way.

Definition 4.9 (Description of a local operator: convolve, reduce, iterate)

The computation of a local operator can be described by using the language constructs `convolve`, `reduce`, or `iterate` within the `kernel` method. These functions take as input a `Mask` or a `Domain` and return an output pixel as shown below:

```
T convolve(Mask<mask_t>, Reduce, const std::function<pixel_t()>)

T reduce(Domain, Reduce, const std::function<pixel_t()>)

void iterate (Domain, const std::function<void()>)
```

The `convolve` and `reduce` methods take an aggregation mode of a Reduce type as an input parameter. Supported modes are listed in Table 4.3. The last parameter (`std::function`) allows users to pass their stencil computation, e.g., using a

Listing 4.1: Hipacc Description of a color conversion as a point operator.

```
class MyPointOperator: public Kernel <uchar4> {
  private :
  Accessor <uchar4> &img;     // a colored input image with uchar4 type

  public:
  // ...

  // Hipacc applies this operation for every output pixel
  void kernel() {
    uchar4 pixel = img();
    output() = 0.3f * pixel.x + 0.59f * pixel.y + 0.11f * pixel.z;
  }
};
```

Table 4.3: Supported aggregation modes for the Reduce type.

Enumeration (of Reduce Type)	Aggregation Mode
SUM	Sum of all values (default)
MIN	Minimum of all values
MAX	Maximum of all values
PROD	Product of all values
MEDIAN	Median of sorted values

lambda function. See Example 4.1 for an example application using `convolve` function. As `Domain` does not hold coefficients on a stencil pattern, `reduce` can be used where a `Mask` is not needed. For instance, a box filter that averages neighboring pixels on a given stencil pattern could easily be described by using `reduce` (see Hipacc repository [Hip22] for more sample codes).

The `iterate` allows for describing more complex operations on a local window, where the aggregation needs to be described explicitly. For an example, see the `Kernel` description of a Bilateral filter shown in Example 4.2.

Example 4.1: Convolution

Listing 4.2 shows an example Hipacc `Kernel` using the `convolve` function to describe a local operator. The stencil pattern of `convolve`'s computation is defined by `mask`. The computation for every pixel on the stencil pattern is described in a lambda function (in Line 11). Finally, the aggregation mode `Reduce::SUM` indicates the final operation on all the computed values. The results are summed here to produce one output pixel from the local input window.

Listing 4.2: Description of a local operator in Hipacc. This example defines a linear filter by using the convolve abstraction.

```
class LinearFilter: public Kernel <uchar > {
  // ...
  public:
    LinearFilter(Accessor<uchar> &input, // input image
                 IterationSpace<uchar> &out, // output image
                 Mask<float> &mask)          // mask
                 : {/* ... */}

  void kernel() { // convolve -> local operator
    output() = convolve(mask, Reduce::SUM, [&] () -> uchar {
          return mask() * input(mask);
    });
  }
};
```

Listing 4.3: Application graph of a Gaussian filter described in Hipacc. It instantiates a local operator called `LinearFilter`. See Figure 4.2 for the illustration of the application graph described by this code.

```
// input and output images
size_t width, height;
uchar *image = read_image(&width, &height, "input.pgm");
Image<uchar> in(width, height, image);
Image<uchar> out(width, height);

// filter mask for Gaussian blur filter
const float filter_mask[3][3] = {
  { 0.057118f, 0.124758f, 0.057118f },
  { 0.124758f, 0.272496f, 0.124758f },
  { 0.057118f, 0.124758f, 0.057118f }
};
Mask<float> mask(filter_mask);

// reading from in with clamping as boundary condition
BoundaryCondition<uchar> cond(in, mask, Boundary::CLAMP);
Accessor<uchar> acc(cond);

// output image (region of interest is the whole image)
IterationSpace<uchar> iter(out);

// instantiate and launch the Gaussian blur filter
LinearFilter Gaussian(iter, acc, mask);
Gaussian.execute();
```

Listing 4.3 presents a Hipacc application that uses the `LinearFilter` `Kernel` (described in Listing 4.2). This code describes the Hipacc application graph illustrated

in Figure 4.2. Hipacc DSL can be used in conjunction with standard C++, such as using OpenCV for I/O and Hipacc for computation. For instance, the user function `read_image` in Line 3 reads the content of a Portable Graymap File Format (PGM) image to a `uchar` memory area before creating a Hipacc `Image` in Line 4. An output `Image`, `Accessor`, `IterationSpace`, and `BoundaryCondition` are defined between Line 5 and Line 20, before creating a `LinearFilter` object (which is defined in Listing 4.2) in Line 23. Finally, the `Gaussian` filter is computed using the `execute` function in Line 24. It can be seen that Hipacc's execution is not eager, meaning that the computational function is executed after its definition. A video processing function can easily be implemented using the `execute` function (Line 24) in a loop. The Hipacc compiler generates the corresponding code when this loop is compiled for a target computing platform.

Example 4.2: Bilateral Filter Kernel (use of `iterate`)

`Iterate` allows describing complex local operators without using an aggregation mode. For instance, in the description of the Bilateral filter below, pixels on the input stencil pattern (defined by `dom`) are used to calculate two variables named `p` and `d`. Later, these parameters are used to produce an output pixel.

```
void kernel() {
  float d = 0.0f, p = 0.0f;

  iterate(dom, [&] () -> void {
    float diff = in(dom) - in();
    float s = expf(-c_r * diff * diff);
    d += s * mask(dom);
    p += s * in(dom);
  });

  output() = (uchar)(p / d + 0.5f);
}
```

Note that `iterate` describes a sequential execution on a local window, but Hipacc's source-to-source compiler can generate a parallel code from this description. Furthermore, early exits from the sequential execution of `iterate` are supported by Hipacc with a language construct called `break_iterate()`.

4.2.7 Global Reduction

Hipacc supports the description of global reductions by the `Kernel` method named `reduce` as explained below.

Definition 4.10 (Description of global reductions: reduce)

Hipacc supports the following reduce method for the reduction operations described on an image:

```
pixel_t Kernel<pixel_t>::reduce(pixel_t left, pixel_t right)
```

The reduction pattern is described in an abstract way solely by using two pixels representing the operation's left and right sides. For instance, an application calculating the maximum of an input image can be described as follows:

```
uchar reduce(uchar left, uchar right) const {
    return max(left, right);
}
```

Then, Hipacc will apply this operation to produce one result from the whole input image (or generate target code when its compiler is used).

4.3 Generating Hardware Accelerators From Hipacc

Previous works include the following: First, Membarth et al. [Mem13; MRH+16] introduced the Hipacc DSL and initial compiler workflow to generate C++, CUDA, and OpenCL code for CPUs and GPUs. A simple overview of this compiler workflow is given in Section 4.3.1. Then, Reiche et al. [RSH+14] extended the initial compiler flow with a new backend that generates C++ code for Vivado HLS. These extensions include several hardware-centric transformations and streaming pipeline generation (see Section 4.3.2). As part of this thesis, we present a new Hipacc backend that generates highly-optimized code as input to Intel FPGA SDK for OpenCL (see also [ÖRH+16]).

The OpenCL code generated for FPGA targets must be significantly different from those generated for CPU and GPU targets to achieve good synthesis results. Unlike the previous Hipacc backend for Vivado HLS [RSH+14], our backend [ÖRH+16] generates an OpenCL host code optimized for Intel FPGA SDK (which requires improving streaming pipeline generation as explained in Section 4.3.2) and introduces further hardware-centric optimizations for accelerating Hipacc `Kernel`s (see Section 4.3.4). These include the novel loop coarsening and image border handling techniques presented in Chapter 3, which are also introduced to the Vivado HLS backend by integrating our metaprogramming library [ÖRH+17a] presented in Section 4.4.

To apply our improvements, we refined the Hipacc's existing compiler workflow as shown in Figure 4.3, where compiler steps highlighted by green are either introduced or modified as part of this thesis' contributions.

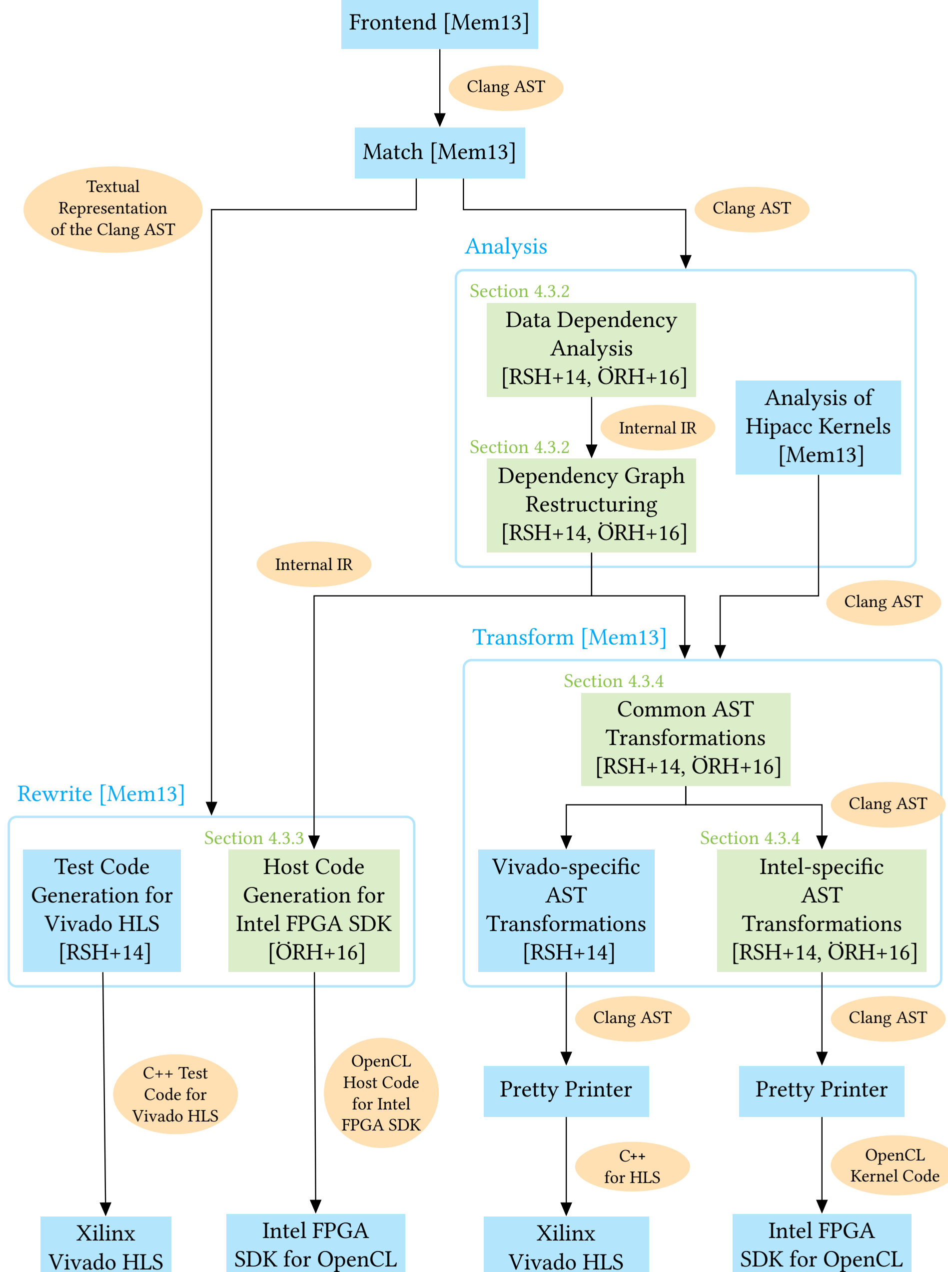


Figure 4.3: Hipacc's source-to-source compilation workflow for generating input code to HLS. Hipacc generates target code using its own backends, where several analyses and transformations are applied on the Clang AST created in the *frontend*. The initial compiler workflow [Mem13] (developed for CPUs and GPUs) is extended in [RSH+14] with a Vivado HLS target, where the *Data Dependency Analysis* and *Dependency Graph Restructuring* steps are introduced for generating a streaming pipeline. We refined this compiler flow in [ÖRH+16] (i.e., compiler steps highlighted by green) to generate OpenCL code for Intel FPGA SDK and to integrate our optimization techniques introduced in Chapter 3 for Hipacc's Vivado HLS backend.

4.3.1 Background: Overview of the Compiler Work Flow

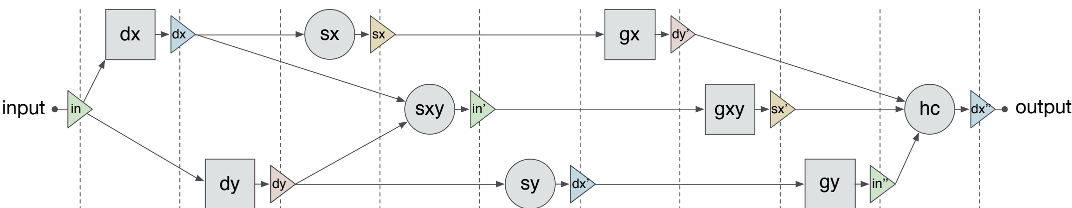
Hipacc's compiler leverages Clang/LLVM compiler infrastructure for typical steps of a compiler front-end (i.e., lexing and parsing, semantic analysis, and intermediate representation (IR) creation). It performs source-to-source transformation through the AST created by Clang, as shown in Figure 4.3. First, in the *Match* step, all AST nodes are traversed to detect declarations, definitions, statements, and expressions using the language components of Hipacc. The *Rewrite* modifies the textual representation of the AST derived from the input application code to generate a so-called *host* code, i.e., the part of the application except the code generated for Hipacc `Kernel` definitions. These modifications include removing Hipacc's `Kernel` definitions and replacing the statements that define the DSL objects and expressions using these objects. For GPU targets, the code generated by *Rewrite* would be the C++ host program controlling CUDA kernels or OpenCL Kernels.

The *Analysis* step analyzes and categorizes the Hipacc `Kernels` and image processing operators to capture the information required for optimizations and code generation. This information includes operator types, memory access patterns, filter types, data dependence, and the number of instructions. Then, *Transform* applies optimizations and modifications for the target device on an internal clone of the AST. Finally, *Pretty Printer* generates code for the target platform from the AST produced by *Transform*. The generated code contains the implementation of Hipacc `Kernels` for the target device, which can be in C++, OpenCL, CUDA, or Renderscript.

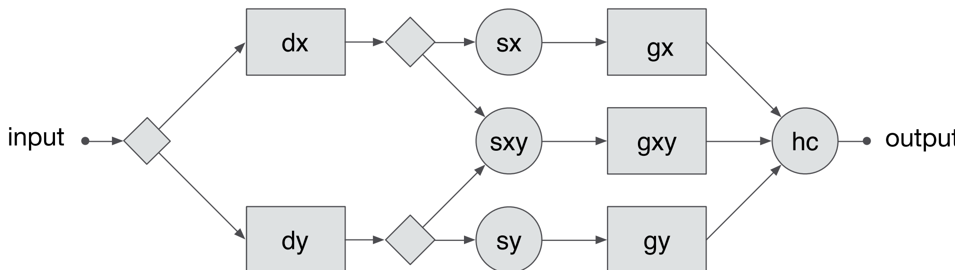
4.3.2 Generating a Streaming Pipeline

Hipacc's CPU and GPU code generation backends (proposed in [Mem13]), use a buffer-wise execution mechanism to generate code for GPUs and CPUs. The buffers used by kernels to exchange data have the size of an entire image. Each subsequent kernel begins only after all earlier kernels have completed their execution. This is a viable strategy for GPUs since buffers act as synchronization points, eliminating the need for slow alternatives. In contrast, FPGAs allow stream processing where only a small amount of data has to be stored for exchange between dependent kernels.

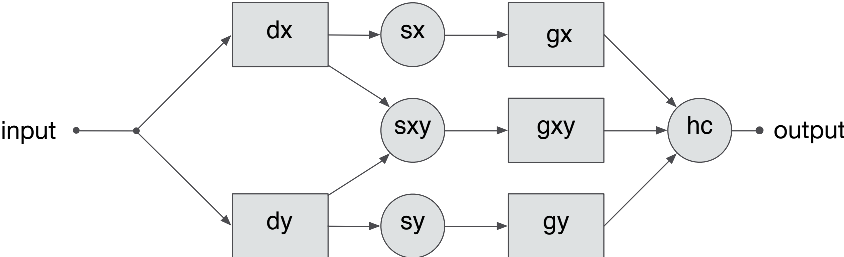
Previous Work: The initial buffer-wise execution model of Hipacc is modified in [RSH+14] to support stream processing when Xilinx Vivado HLS is selected as a target. Figure 4.4 shows the difference between the implemented execution models for a Harris corner detection algorithm. Thereby, streaming objects replace the buffers to construct a streaming pipeline where FIFO semantics are implemented for data exchange between kernels. Additional glue logic (so-called split stream objects)



(a) Hipacc's sequential execution for the Harris corner detector [Mem13] The code generated for CPUs and GPUs utilizes image size buffers for synchronization and data exchange (triangle: buffers). (Figure reprinted from [RSH+14], © 2014 IEEE)



(b) The execution model of the streaming pipeline generated for Vivado HLS [RSH+14]. It consists of streaming objects to split data of a single streaming object when needed for multiple kernels (diamond shape: split stream objects). (Figure reprinted from [RSH+14], © 2014 IEEE)



(c) The execution model of the streaming pipeline generated for Intel FPGA SDK for OpenCL. In our work [ÖRH+16], we eliminate the cost of creating split stream kernels by generating kernels that can write to multiple streams.

Figure 4.4: The execution model of a Harris Corner Detection algorithm for different targets (circle: point operators, rectangular: local operators).

is added to multiply the input stream when multiple kernels read from the same buffer (the case of one producer and multiple consumers), as shown in Figure 4.4b.

For this purpose, Reiche et al. [RSH+14] proposed the *Data Dependency Analysis* and the *Dependency Graph Restructuring* steps shown in Figure 4.3, where the compiler backend for Vivado HLS creates a streaming pipeline as follows:

(i) First, in *Data Dependency Analysis*, kernel executions, buffer allocations, and memory transfers are traced to identify data dependencies between `Kernels`. The dependency information is fed into an internal representation (depicted as *Internal IR*), which is a condensed AST-like structure based on a bipartite graph with two vertex types: *space* denoting buffers and *process* denoting kernel executions. Writes to buffers are transferred to the *internal IR* in a single static assignment (SSA) manner by iterating over the kernel executions in the sequential sequence in which they are provided. With this, the *spaces* between *process* nodes are marked as streams.

(ii) Second, in *Dependency Graph Restructuring*, additional AST nodes are created to define streaming objects. The buffers requiring split stream logic (i.e., multiple kernels reading from the same stream) are identified and split stream objects are added. Then, the unused nodes (i.e., the ones that do not contribute to output) are pruned by traversing the rebuilt graph backward in depth-first order, starting from the output spaces.

(iii) Finally, as part of *Common AST Transformations*, the Clang AST *frontend* is transformed to insert copy *process* nodes, according to the *internal IR*. In this way, the AST description of the buffer-wise execution is modified to a streaming pipeline execution.

Our Contributions: The previous work [RSH+14], explained above, adds glue logic (so-called split stream objects) to multiply the input stream when multiple kernels read from the same buffer (the case of one producer and multiple consumers), as shown in Figure 4.4b. That is, $n + 1$ streams are instantiated for the buffers read by n kernels. This is a practical approach since the overhead on Vivado HLS is not significant. However, using an additional stream severely affects the resource usage when Intel FPGA SDK for OpenCL is used as the target HLS tool[2]. What is more, utilizing glue logic between OpenCL kernels for multiplying an input stream (to implement the split as mentioned above stream mechanism) requires implementing another OpenCL kernel, introducing a severe overhead (mostly for interface synthesis) as presented in [ÖRH+16]. Instead, we modify the kernels when one of its output buffers is read from subsequent multiple kernels as shown in Figure 4.4c. As a result,

[2] Stream objects are implemented by using so-called *channel* objects in Intel FPGA SDK for OpenCL

reads with n kernels produce n many streams without inserting split stream kernels, thus generating an optimized streaming pipeline for Intel FPGA SDK for OpenCL.

Correspondingly, the following modifications are applied in our compiler workflow:

(I) The Hipacc DSL does not support writing to multiple `Image` objects from the same `Kernel` (i.e., the Hipacc compiler allows using only one `IterationSapce` in one `Kernel` description). To facilitate writing the same data to multiple streams in our Intel FPGA SDK for OpenCL backend, we altered the (initially proposed [RSH+14]) *internal IR* of the streaming pipeline to support merging split stream objects with *process* nodes [ÖRH+16].

(II) In our implementation [ÖRH+16], the *Data Dependency Analysis* locates the nodes requiring split stream logic, and the *Dependency Graph Restructuring* merges them with the previous kernel nodes. Subsequently, in our *internal IR*, the *process* nodes before these buffers are modified to have additional stream parameters to output multiple streams using the same data.

(III) Finally, the *Common AST Transformations* step is redesigned to use our *internal IR* to translate *process* nodes to OpenCL kernels that can produce multiple output streams (i.e., for image and data parameters).

4.3.3 Generating Host Code

OpenCL standard provides an open, royalty-free API to control heterogeneous systems using a host-device paradigm. Intel FPGA SDK synthesizes FPGA circuits as well as system-level interfaces from OpenCL kernels to manage and execute these accelerators from a CPU. Thereby, OpenCL host code (a C++ code using OpenCL API) deals with runtime tasks such as programming FPGAs with the bitstream generated by Intel FPGA SDK (called offline compilation in OpenCL standard), initialization of the device (FPGA), memory management, synchronization, and context management.

Hipacc supports generating OpenCL code for GPUs and CPUs [Mem13]. Still a new host code generation module had to be developed to target Intel FPGA SDK for OpenCL, as summarized in the following:

- First, to generate a host code according to the streaming pipeline execution model, we redesigned the compiler flow in [RSH+14] to use *internal IR* created by *Dependency Graph Restructuring* (shown in Figure 4.3) as part of host code generation in the *rewrite* step. With this, the kernel parameters marked as a *stream* type are generated as *channels* for the *Intel* tool. Corresponding copy operations (i.e., data transfers between the host CPU and FPGA) and parameter settings are described in the OpenCL host code. Finally, OpenCL kernels

are *enqueue*d to corresponding *command queues* according to the streaming execution model, as recommended in the Intel FPGA SDK user guide [Int17].

- Second, we modified Hipacc's OpenCL runtime library since Intel FPGA SDK for OpenCL

  (i) extends the OpenCL API with tool-specific functions and structs, such as *channels* (stream objects)

  (ii) does not support compilation of the kernel code from the source code

  (iii) forces users to add an extra OpenCL *command queue* to an OpenCL *context* for every kernel that is supposed to run on the FPGA logic (which is unusual for the OpenCL execution model where a single *queue* is used to add tasks of a programming *context*).

Note that Hipacc's Vivado HLS backend [RSH+14] generates a test code where the whole FPGA implementation is described as one C++ function. Hence, its test code generation in the rewrite module removes all the Hipacc-related descriptions except the input/output parameter copies, thus not dealing with the host code generation challenges of our OpenCL backend mentioned above.

4.3.4 Generating Kernel Codes (Hardware Accelerators)

As shown in Figure 4.3, our backend generates an OpenCL kernel for each Hipacc `Kernel` by using *Pretty Printer* after transforming the Clang AST produced by the *frontend* with several hardware-centric optimizations. First, the portions of the AST describing Hipacc `Kernels` are extracted in the *Common AST Transformations*, and the following optimizations are applied:

(I) The extracted AST is extended with new nodes to describe an OpenCL kernel, e.g., nodes for setting (OpenCL thread) index calculations for buffer access and ROI index shifts. Memory access operations are replaced by numeric literals to apply constant propagation, which reduces area usage.

(II) Capturing the memory access patterns and computational information from the high-level abstractions from Hipacc's language components allows generating application-specific code. An application-specific memory architecture is described depending on the `Kernel` type. For instance, a description of line buffers and sliding windows are added for local operators. These are described by using *local* memory and registers, which are synthesized to on-chip memory blocks and registers by Intel FPGA SDK for OpenCL. Correspondingly, loops,

including arithmetic operations, are unrolled when needed (e.g., arithmetic operations reading from a sliding window are unrolled to achieve an II of 1).

(III) Depending on the compiler arguments, the throughput of a `Kernel` is increased by using our loop coarsening and image border handling techniques (see Chapter 3). This requires reading multiple pixels from input buffers, replicating arithmetic operations to produce multiple outputs in every clock cycle, and modifying on-chip memory architecture to hold more data efficiently.

(IV) Finally, in *Intel-specific AST Transformations*, primitive data types (such as *integer*, *short*) are changed to arbitrary bit width representations according to *Intel* syntax. That is, a masking operation must be added when data is read and written. FPGA resources can be saved if smaller bit width is sufficient. For instance, if one knows that a 2-bit variable is sufficient, an 8-bit variable (char) y has to be declared, which can be decreased to 2 bits as follows:

```
y & 0x3
```

Similarly, arithmetic operations are described as follows:

```
y = (RHS) & 0x3
```

These bit width optimizations reduce the readability of the generated code but provide correct results and can significantly reduce hardware cost. Correspondingly, we extended the Hipacc DSL in [ÖRH+16] to support definition of arbitrary bit width variables as shown in Listing 4.4.

Listing 4.4: An example for using our bit width annotation extension in Hipacc DSL

```
1  void kernel () {
2      #pragma hipacc bw(sum, 12)
3      uint sum = 0;
4
5      #pragma hipacc bw(x, 3)
6      int x = 0;
7
8      /* hipacc kernel code */
9      output() = sum << 4;
10 }
```

4.4 Metaprogramming in C-based HLS: Alleviating the Tasks of a Source-to-Source Compiler

Metaprogramming is known as writing programs that treat other programs as their data, thus having the ability to read, analyze, transform, and generate other programs

(or itself). Examples include writing compilers, interpreters, assemblers, and program analyzers. C++ and OpenCL provide the kind of metaprogramming concerned with creating code that can be used as a portion of a program. They can be viewed as simple compilers. In this section, we advocate using these metaprogramming features to alleviate the tasks of the HLS compiler backends in Hipacc.

As explained in Section 4.3, generating code for target HLS tools from a Hipacc application requires several transformations at AST level. Modifying AST is a complex task and hinders extensibility, where adding further optimizations requires modifying the compiler backend. This complexity can be significantly eased by using a template library that contains generic implementations of Hipacc's computational abstractions for the target HLS tools. Using such a template library as part of Hipacc's HLS backends as shown in Figure 4.5 has the following benefits: Our approach

(i) reduces the required AST transformations to a smaller set of optimizations

(ii) allows extending the implementation techniques of the library without modifying the Hipacc compiler (or only with minor changes)

(iii) provides a standalone meta library for HLS users to describe their application in a generic way even without using Hipacc (which could especially be useful to describe applications that Hipacc does not support).

4.4.1 Metaprogramming Techniques in OpenCL and C++

We developed two distinct template libraries as shown in Figure 4.5, where the libraries consist of the same hardware abstractions but are built with different metaprogramming techniques. OpenCL (based on C99) only supports C preprocessor macro system, whereas C++ offers more sophisticated tools as part of its template metaprogramming language. Therefore, we only present our C++ template library in Section 4.5 after summarizing both metaprogramming techniques in this section.

Code Generation with C Preprocessor Macros

C macros can only perform text replacement and macro expansions, translating an input sequence to an output sequence according to a user-defined rule. The preprocessor commands start with a # symbol, as in `#define`, `#if`, and `#include`. For instance, the following defines a macro that swaps two objects like a function:

```
#define SWAP(x, y) {  \
    auto tmp = y;     \
    y = x;            \
    x = tmp; }
```

However, C macros are are handled by a lexical preprocessor before being parsed by the actual compiler. They do not act like functions. In particular, they do not create

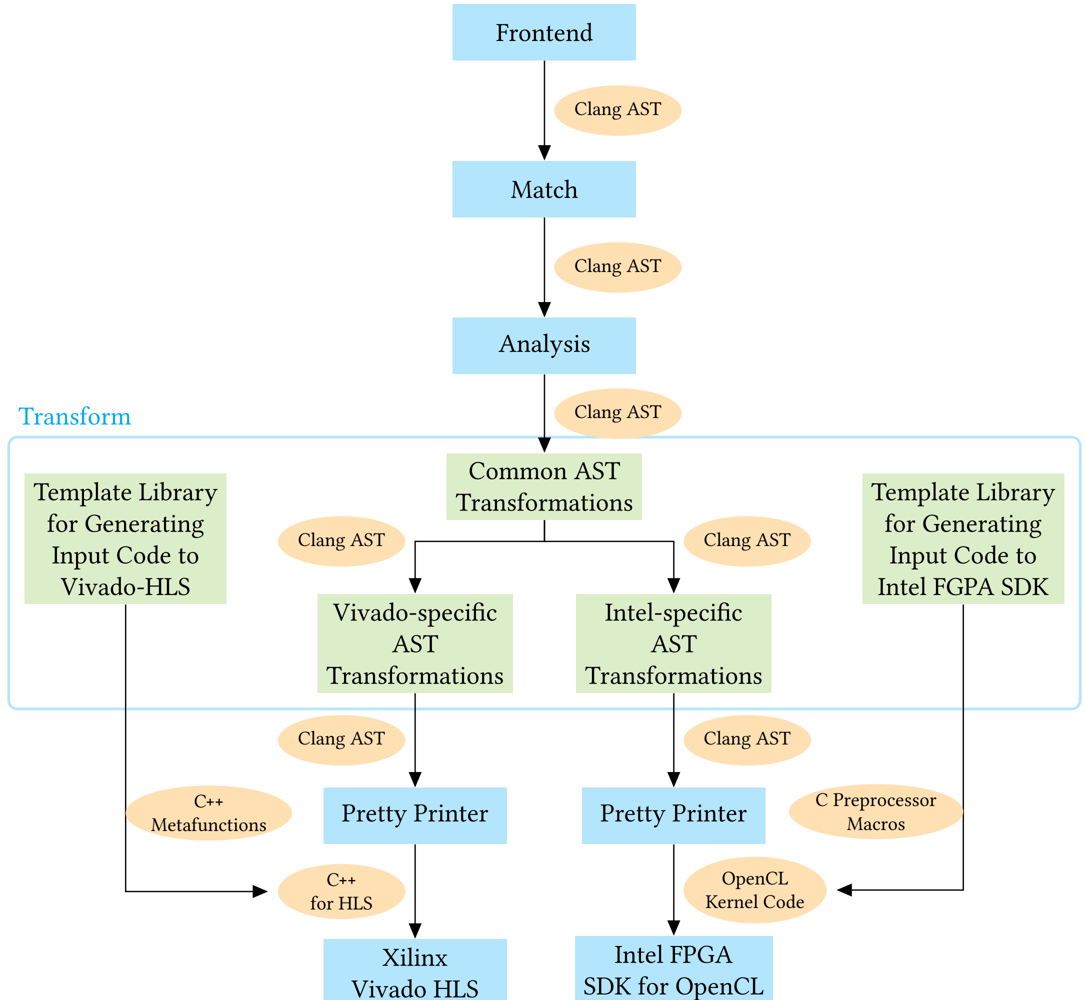


Figure 4.5: A simplified representation of the Hipacc compiler flow that uses template libraries for code generation (compiler steps and template libraries highlighted by green are introduced as part of this thesis and [ÖRH+17a]). Using a template library significantly alleviates the tasks of our source-to-source compiler backend while increasing its extensibility. That is, C++ and OpenCL code generated by Hipacc as input to HLS tools include metafunctions (high-level abstractions) from the template libraries. This reduces the AST transformations required by *Transform* step. Furthermore, these template libraries contain hardware abstractions for HLS users to describe their custom hardware implementations in a generic way, even when Hipacc is not used.

a separate scope or consider arguments as independent entities. Hence, when not carefully written, the macros can produce unexpected results.

Code Generation with C++ Template Metaprogramming

C++ metalanguage is different from its core language. A compile-time subset of the C++ core language is available for writing metaprograms. Nevertheless, templates are Turing-complete [ST06] together with other C++ features–that is, in principle, capable of computing anything computable. C++ template metaprogramming allows for generalizing concrete implementations by abstracting data types, compile-time constants, and complete functions without additional runtime overheads. Thereby, programmers can design high-level abstractions that take as input a functional description and generate transformed residual code without introducing any run-time overhead. This allows the decoupling of the algorithm description from its implementation.

A C++ compiler treats an identifier following the `template` keyword as a parameterized entity. An entity can be a function or a class (type). Template parameters can be types, non-types (also called value parameters), or template parameters. We give a brief overview of template metaprogramming in the following and refer to [Ale01] for a more comprehensive overview of the metaprogramming techniques used in this chapter (e.g., compile-time computations, type traits, policy-based design).

Definition 4.11 (Function templates)

The definition of a function template looks like a normal function, except that it is expressed with a generic type. An example C++ function template is shown below:

```
template<typename T>
T max(T x, T y)
{
  return (x > y) ? x : y;
}
```

The template function `max` can be called with different types such as **`int`**, **`float`**, or a user-defined type. A distinct definition of a function is generated for each type that this function is called before the compilation of the application.

Definition 4.12 (Class templates)

Similar to function templates, classes can also be defined by parametric types. This allows the definition of abstract types as shown below:

```
template<typename T, size_t W, size_t H>
class Image {
public:
    T data[W][H];
}
```

The template class `Image` is a container parametrized on the element type `T` and dimensions `W`, `H`. Like function templates, the compiler generates code for

defining a unique class definition for every type the user calls the Image class.

Definition 4.13 (Template Instantiation)

The process of generating types and functions from template definitions is referred to as template instantiation. The compiler generates a new function or a type for the first call to a template entity. The resulting entity is called a specialization. The same specialization is called every time a template entity is called with the same type. For instance, the compiler generates three instantiations of the max template function for the code below:

```
max<int>(x, y);
max<int>(3, 4);
max<float>(a, b);
%max<Image<uchar, 1024, 1024>>(a, b);
```

Note that only one copy of the instantiation will end up in an executable file, regardless of multiple identical instantiations happening in different modules. Similarly, no code (specialization) will be generated in case of a no-call to a template entity.

Template instantiation is performed at compile time but before compiling the whole program. Specialization of a template entity is in a type-safe state only after its successful instantiation. Current C++ compilers type-check the code of a template entity in two phases [CE14]: First, expressions not depending on template parameters are type-checked when the template definition is parsed. Then, the template parameters are replaced with concrete arguments during the instantiation, and full type-checking is performed. This two-phase checking has the following disadvantages:

i) well-typedness of a generated program cannot be guaranteed by running the metaprogram,

ii) errors are hard to understand since most of them are captured during instantiation, not where they occur,

iii) type correctness for modules involving templates cannot be guaranteed from a library.

4.4.2 C++ Metaprogramming for Building High-Level Abstractions

In computer science and mathematics, a higher-order function is a function that i) takes one or more functions as arguments (i.e., procedural parameters), and ii) returns a function as its result. All other functions are categorized as first-order functions.

Functions and Callables

A function is a named group of statements that can be invoked from other parts of the program or from the function itself in the case of recursive functions. As explained in the following, C++ provides several ways to define a function.

Definition 4.14 (C-like functions)

The usual C++ functions are defined and called as follows:

```
int max(int x, int y)
{
  return (x > y) ? x : y;
}

auto z = max(4, 7);
```

Definition 4.15 (Function Objects:)

C++ offers a technique to construct new types that act like functions. Unlike other operators, a call operator `()` can take an arbitrary number of arguments of any type, allowing programmers to create a function object with any signature they want. This allows for a better way of passing callables to functions than passing C-like functions, where types are implicitly transformed to function pointers when passed as function parameters:

Defining a `max` function using a function object is shown below:

```
class Max {
public:
    int operator()(int x, int y) const
    {
        return (x > y) ? x : y;
    }
}

auto z = Max(4, 7);
```

An instance of such a class is a *callable*.

Definition 4.16 (Lambda Functions)

Lambda functions in C++ allow to define an anonymous function object (i.e., called a closure) right at the location where it's invoked or passed as an argument to a function. A lambda expression is a syntactic sugar for function object description and is often used to encapsulate a few lines of code that are passed to algorithms or asynchronous functions.

The semantics of lambda expressions were first introduced in C++ 11 (the first version of the standard called Modern C++). The syntax of a lambda expression in C++ consists of several parts: a capture clause (i.e., `[]`), which specifies which

variables are captured and whether the capture is by value or by reference; an optional parameter list (i.e., ()); an optional mutable specification; an optional exception-specification; an optional trailing-return-type; and a lambda body (i.e., {}).

Using lambda functions increase readability of the code by decreasing required boiler code for function object description. This makes them perfect candidates to be used in generic programming. An example definition of a C++ lambda function is shown below.

```
auto max = [](int x, int y) { (x > y) ? x : y; };
```

The compiler will treat the type `max` as a function object whose call operator `()` has two integer parameters for `x` and `y`. Writing a lambda function requires less boilerplate code compared to function objects.

Creating Higher Order Functions

Higher-order functions allow for raising the expressiveness of the language and make the programs shorter. In this work, we use them to hide low-level implementation details by building high-level abstractions.

In C++, a function can be given as input to another function using

i) a function pointer,

ii) `std::function` wrapper, or

iii) template type deduction.

Using a function pointer leads to polymorphic code, and having a `std::function` type as a function parameter can cause hidden copies. Unlike these techniques, the template type deduction mechanism allows high-level abstractions to be created without additional runtime overhead. An example is shown in Example 4.3.

Example 4.3: Passing function objects using template deduction

An example of creating a higher-order function is shown below:

```
template <typename FN, typename T1, typename T2>
auto high_level_abs(const FN& op, T1 x, T2 y) -> decltype(op(x, y))
{
    return op(x, y);
}
```

The meta function `high_level_abs` accepts any callable (op) as a parameter. The return type is deduced according to the return value of the op callable. Compilers create one instance of the passed object and fully inline the corresponding expressions. In this way, C++ metaprogramming provides a functional way of

describing a program and generating code from another program taken as input.

4.5 A Comprehensive Metaprogramming Library for Highly Efficient C++-based High-Level Synthesis of Image Processing Applications

In this section, we present a metaprogramming library [ÖRH+17a] that consists of high-level abstractions for image processing operators as well as hardware design elements crucial for image processing algorithm implementations, i.e., line buffers, and sliding windows. In this way, a high level of productivity and modularity is provided for application developers and circuit designers. Application developers can describe an algorithm by using image processing operators. During hardware synthesis, these operators are refined by specific hardware implementations. Furthermore, we use this library to alleviate the tasks of Hipacc's Vivado HLS backend as discussed in Section 4.4.

Our library allows programmers to express image processing applications as DFGs. During hardware synthesis, these operators are refined by specific hardware implementations. This allows the compiler to perform temporal and spatial parallelization transformations for the implementations of a described algorithm according to user-defined parameters. For example, the user can set the initiation interval for pipelining and the number of processed input pixels in each iteration to exploit data-level parallelism.

Our library contains multiple template architectures for different specifications of the same algorithmic instances. More specifically, it consists of multiple descriptions of image border handling implementations and varying coarse-level parallelization strategies considered for different input parameters of a local operator specification (see Chapter 3 for more details). Selecting the most suitable implementation is hidden from users by implementing a compile-time selection procedure based on the trade-offs analysis introduced in Section 3.6.

4.5.1 Motivational Example

This section presents a collection of higher-order functions in the form of a C++ template library that describe hardware implementations of image processing applications without exposing low-level implementation details to the users. This allows library users to define *what* the program should do on a higher level. To put it in another way, assume that the generic functions $f^{algorithm}$ and $f^{implementation}$ describe the behavior of an algorithm (what?) and its hardware implementation (how?),

respectively. Input parameters of these functions can be static (known at compile time, e.g., coefficients of a constant filter) or dynamic (known at runtime, e.g., input data). Then, $f^{implementation}$, which takes $f^{algorithm}$ as input, should be specialized for the static inputs as well as for $f^{algorithm}$.

$$f^{implementation}_{static\ input\ parameters}(f^{algorithm},\ dynamic\ parameters) \tag{4.1}$$

Users are only required to define $f^{algorithm}$ and input parameters since our library provides $f^{implementation}$. In this way, our library allows users with no hardware design experience to achieve high-quality synthesis results through a subsequent high-level synthesis applied on the source code generated by using our library.

Example 4.4: Non-generic description of a local operator in C++

A mathematical description of convolution for a w-by-h mask is shown as follows:

$$output\ image(x, y) = \sum_{j=-\lfloor h/2 \rfloor}^{j=\lfloor h/2 \rfloor} \sum_{i=-\lfloor w/2 \rfloor}^{i=\lfloor w/2 \rfloor} mask(i, j) \cdot img(x + i, y + j) \tag{4.2}$$

Non-generic sequential example: A non-generic sequential C++ description of a convolution function is given as follows, where image border handling is ignored for simplicity:

```
void convolution(const uchar& img[2048][1024],
                 const uchar& output_image[2048][1024]) {
  const char mask[3][3] = {
    {-1, 0 1}, {-2, 0 2}, {-1, 0 1}
  };

  for(size_t y = 1; y < 1023; y++)
    for(size_t x = 1; x < 2047; x++){
      short sum = 0;

      for(int j = 0; j < 3; j++)
        for(int i = 0; i < 3; i++)
          sum += img[y - 1  + j][x - 1 + i] * mask[j][i];

      uchar output_pixel = static_cast<uchar>(sum);
      output_image[y - 1  + j][x - 1 + i] = output_pixel;
    }
}
```

The above code describes a 3-by-3 convolution to be performed on an unsigned 8-bit 2048-by-1024 image. Writing image or mask size as function parameters creates a more generic function but at the cost of increased resource usage and execution time. The slow-down is caused by the complex control structure utilized for handling the runtime parameters image and mask size.

Generic optimized example: The convolution in Eq. (4.2) can be expressed as the iteration of a local operator computing an output pixel for every input pixel in a local window of neighboring pixels by using a computation function *Convolve* (as shown in Figure 4.6):

$$output\ image = localOp(img, Convolve) \tag{4.3}$$

where the computation function *Convolve* can mathematically be expressed as follows in the case of a constant mask with the dimensions of width w and height h.

$$output\ pixel = Convolve_{win,w,h}(img, x, y) \tag{4.4}$$

Correspondingly, by using C++ template metaprogramming, we can express the function *Convolve* in Eq. (4.4) as a lambda function (or as a function object);

```
short Convolve = [](unsigned char win[3][3]) {
    const char filter[3][3] = {
        {-1, 0 1}, {-2, 0 2}, {-1, 0 1}
    };

    short sum = 0;
    for(int j = 0; j < 3; j++)
        for(int i = 0; i < 3; i++)
            sum += win[j][i] * filter[j][i];

    output_pixel = static_cast<uchar>(sum);
    return output_pixel;
};
```

and the higher order function of a *localOp* class as follows:

```
Image<unsigned char, 1024, 2048> img{ data };
Image<char, 1024, 2048> output_image;
localOp<ImageWidth, ImageHeight, KernelWidth, KernelHeight> local;

local(output_image, img, Convolve);
```

Here, the C++ template object `localOp` takes as input function object `Convolve` and generates highly optimized code as input for a high-level synthesis tool, i.e., Vivado HLS. The generated C++ code will lead to the instantiation of optimized line buffers and a sliding window hardware unit to deliver a pipelined implementation. The dimensions of the required on-chip memory elements to be allocated thereby depend on the input pixel bit width, input image width, and window size of the *convolve* function. The output image size depends on the output data type of the f function.

The static parameters, such as image width, and height, are defined as template parameters. Therefore, the `local` function can be called with any image and mask size. The users of the template library are not required to understand the implementation of `localOp`. They are only required to describe the callable

function Convolve. Similarly, the Vivado HLS backend of Hipacc is required to generate solely the Convolve function, Image, and localOp descriptions shown above. This way, the tasks of several AST transformations are alleviated by using C++ metaprogramming.

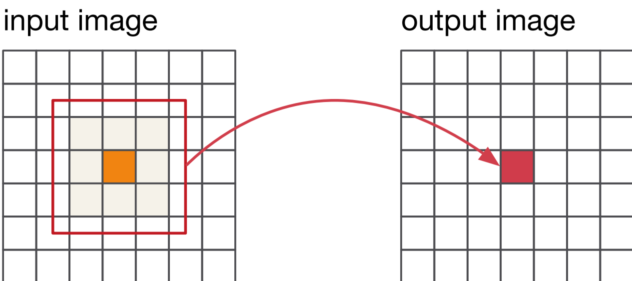


Figure 4.6: Output of a local operator depends on neighboring pixels within a local window at the input image.

4.5.2 Overview of the Library

This section presents the specifications in our proposed library for describing image processing applications.

Data Types

First, we present common functions, types, and traits used to describe hardware in C++ that support the efficient synthesis of point, local, and global operators using Vivado HLS.

Arbitrary Bit Widths Vivado HLS [Xil17c] provides special data types to describe customized bit width operations. It allows representing integer types by `ap_int` and `ap_uint`; and the fixed-point data types by `ap_fixed`. The library [ÖRH+17a], proposed in this thesis, can operate on both built-in types (such as `char`, `integer`) as well as the data types mentioned above provided for bit-accurate operations. We often need to get the bit width of a given variable at compile time, regardless of its type. Therefore, we offer a `get_bitwidthw` metafunction in our library:

Definition 4.17 (Getting bit width of a type)

The bit width of a variable can be calculated using the following type trait at compile time:

```
// definition
template<typename T>
struct Bitwidth;

// helper type
template<typename pixel_t, unsigned p_factor>
using get_bidwidth = typename Bitwidth<T>::bitwidth
```

In C++ template metaprogramming, creating type traits allows for inspecting and transforming the properties of a type at compile time. The `Bitwidth` is a meta-type that holds the bit width of the template type `T`. Using the template specialization technique, implementation of the `Bitwidth` type changes according to `T` being a C++ built-in type or one of the types provided by Vivado HLS for representing arbitrary bit width variables. The bit width of a given variable can be obtained by using our helper type `get_bitwidth` as shown below:

```
// usage
constexpr int bw1 = get_bitwidth(int);
constexpr int bw2 = get_bitwidth(ap_int<8>);

ap_uint<16> c;
ap_uint<get_bitwidth(decltype(c)) * 2> new_c;
```

The return value of `get_bitwidth` can be used as a parameter since it is calculated at compile time.

Our library operates on arbitrary bit width values. Hence, the implementation of image processing operators holds all data in `ap_uint` type but casts them back to their original type just before arithmetic operations. In this way, we use bit-precise data types for the intermediate calculations and memory operations.

Parallelization Support Loop coarsening increases the data bandwidth of an implementation's input, output, and interconnecting streams, as explained in Chapter 3. In our library, we support parallelizing the implementation of a described application for a given compile-time parameter denoted as *parallelization factor* (`p_factor`) in the rest of this section. For this purpose, users and library functions must be able to pack multiple data units into a vectorized type, a so-called *data beat*. For instance, reading four pixels of `unsigned char`, which is a 1-byte data, requires packing them into a data beat of size 4 bytes. This packing is similar to creating a vectorized data type by using `ap_uint`.

Definition 4.18 (Creating Customized Bit Width Vectorized Data Types)

We introduced a metafunction to create a data beat type for a given paralleliza-

tion factor (p_factor) and a given data type, as follows:

```
// definition
template<typename pixel_t, unsigned p_factor>
struct vect_type;

// helper type
template<typename pixel_t, unsigned p_factor>
using vectorize_t = typename vect_type<pixel_t, p_factor>::type
```

Then the return type of vectorize_t can be used for defining types as shown below:

```
// usage
vectorize_t<int, 4> my_vectorized_int = 0; // type: ap_uint<128>
vectorize_t<uchar, 16> my_databeat = 0;    // type: ap_uint<128>
```

In our library, inputs and outputs of the functions are defined with parallelizable data types, whose parallelization factor depends on a global variable that can be set by Hipacc (or an HLS user). This enables an automatic parallelization of the implementation.

Packing and Unpacking Data A data element or consecutive elements can be read or written partially from a data beat as shown below:

Definition 4.19 (extract)

Programmers can use extract to return a data token from a data beat (vectorized data) as follows:

```
// definitions
template<typename Data_t, typename Databeat_t>
Data_t extract(const unsigned index, const Databeat_t& databeat);

template<typename Data_t, typename Databeat_t>
void extract(Data_t& data, const unsigned index, const Databeat_t&
     databeat);
```

Then the return type of vectorize_t can be used for defining types as shown below:

```
// usage
vectorize_t<uchar, 4> databeat = /*..*/;
uchar pixel;

// way1
pixel = extract<uchar>(2, databeat);

// way2
Data_t extract(pixel, 2, databeat);
```

Definition 4.20 (assign)

The `assign` function allows writing a data token to a databeat for a given index as follows:

```
// definition
template<typename Data_t, typename Databeat_t>
void assign(const Data_t& data, const unsigned index, Databeat_t&
    databeat);
```

For instance, if a `Databeat_t` contains four `uchar` values, writing to index two as in the following modifies the second least significant byte:

```
// usage
vectorize_t<uchar, 4> databeat = /*..*/;
uchar pixel = 128;

assign<uchar>(2, pixel, databeat);
```

Stream-Based Image Processing

Stream processing is a beneficial paradigm to express an algorithm as a DFG without explicitly managing allocation, synchronization, or communication. Xilinx Vivado HLS provides the data type `hls::stream<pixel_t>` to support the description of this paradigm. Thereby, the modules can be connected via streams according to a producer-consumer relation in which stream data is passed through introduced FIFO buffers. However, care should be taken when a stream's output is read from multiple nodes. To solve this issue, we provide the following `split_stream` function.

Definition 4.21 (Interconnecting streams)

In the library, an input stream could be replicated to multiple output streams by using one of the overloads of `split_stream` function as follows:

```
hls::stream<pixel_t> repl1, repl2, in;

split_stream(in_s, out0_s, out1_s);
split_stream(in_s, out0_s, out1_s, out2_s);
split_stream(in_s, out0_s, out1_s, out2_s, out2_s);
```

In the implementation, the `split_stream` function reads from the input stream and writes to output streams. Inserting a `split_stream` function costs only the registers holding the additional streams and a small control logic for synchronization.

Image processing abstractions in our library use `hls::streams` as input and output parameters to leverage the benefits of stream processing and overlap the executions between functions.

Specification of Image Processing Operators

By using our library, programmers define an algorithm for point, local, and global operators by writing a lambda function or a function object to describe the behavior of an operator and then call the operator function with the user-defined callable, as briefly shown below:

Definition 4.22 (Point operators)

A point operator calculates an output from each input pixel. The output data type can be different from the input type. This behavior can be described as follows:

```
// way1: point operator behavior of the datapath
class PointKernel {
public :
    T  operator ()(const pixel_t& pix) const {
        return static_cast<T>(pix * pix);
    }
}
```

Using a lambda function as part of our metaprogramming approach makes this specification more compact:

```
// way2: point operator behavior of the datapath
auto point_kernel = [](const pixel_t& pix) -> T {
    return static_cast<T>(pix * pix);
}
```

Then, this description can be passed to the `PointOp` class template as follows:

```
PointOp(hls::stream<out_t>, hls::stream<in_t>, typename KernelType);

PointOp<p_factor>(hls::stream<out_t>, hls::stream<in_t>,
                  typename KernelType);
```

A `PointOp` reads input data from and writes output data to a stream. It allows parallelizing the implementation for a given template parameter (denoted by `p_factor` above. When `p_factor` is not explicitly specified, a value of `p_factor = 1` is assigned as the default. An example use is shown below:

```
hls::stream<vectorize_t<pixel_t, p_factor>> stream_in, stream_out;
PointOp<p_factor>(stream_out, stream_in,
    [](const pixel_t& pix) -> pixel_t {
        return pix * pix;
    });
```

Definition 4.23 (Local operators)

A local operator calculates a result from a two-dimensional region, the so-called window, as follows:

```
auto local_kernel = [](const pixel_t (&win)[width][height]) -> T {
    unsigned sum = 0;

    for(uint j = 0; j < height; ++j){
      #pragma HLS unroll
        for(uint i = 0; i < width; ++i){
        #pragma HLS unroll
            sum += win[j][i];

    return (T)(sum / (width * height));
}
```

A local operator can be described in two steps. First, a C++ object should be created by using one of the following functions:

```
// instantiation
// simple
auto local_op = make_local<ImageWidth, ImageHeight, KernelWidth,
    KernelHeight, pixel_t>();

// with p_factor and BoundaryCondition
auto local_op = make_local<ImageWidth, ImageHeight, KernelWidth,
    KernelHeight, pixel_t, p_factor, BoundaryCondition>();

// with a local operator kernel
auto local_op = make_local<ImageWidth, ImageHeight>(local_kernel);

auto local_op = make_local<ImageWidth, ImageHeight,
    p_factor>(local_kernel);
```

As can be seen, giving the kernel description (e.g., `local_kernel` above) as an input parameter decreases the required number of template parameters. BoundaryCondition can be set to one of the following types (same as the boundary modes supported in Hipacc, see Table 4.1): `UNDEFINED`, `CONSTANT`, `CLAMP`, `MIRROR`, `MIRROR-101`, `REPEAT`. The created local operator object, then, can be called with an input and output stream as follows:

```
local_op(stream_out<out_t>, stream_in<in_t>, local_kernel);
```

It can be seen that the description of a local operator kernel above uses Vivado HLS pragmas, e.g., `unroll`. In this case, our library eliminates the required hardware design knowledge significantly. Hipacc applies the required optimizations, including constant propagation and loop unrolling, for the description of a local operator kernel (e.g., `local_kernel` above) and creates the corresponding local operator object by calling (e.g., `make_local`) function. This eases the number of AST transformations required from the HLS backends of Hipacc.

Definition 4.24 (Global Reduction)

A reduction kernel is described by two parameters that represent the left and right side, as shown below:

```
auto max_kernel = [](pixel_t left, pixel_t right) -> pixel_t {
    return (left < right) ? right : left;
}
```

Then programmers can describe the iteration over the image as follows:

```
reduce<p_factor, width, height>(hls:stream stream_out<out_t>,
    hls::stream stream_out<in_t>, typename ReduceKernel)

reduce<p_factor, width, height>(hls:stream stream_out<out_t>,
    hls::stream stream_in<pixel_t>, typename ReduceKernel,
    in_t pixel_t)
```

User-defined Operators Other operators can be described using global or static variables in a combination of the abstractions provided by the library. Thereby, the parallelization of a user-defined implementation is supported as illustrated below:

```
for(size_t i = 0; i < image_size / p_factor; ++i)
{
    // ...
    dataBeatIn << inStream;

    // code below this loop works in parallel
    for(v = 0; v < p_factor; v++)
    {
        #pragma HLS unroll
        extract<pixel_t>(pixel, data_beat_in, v);

        // ... do something with pixel
        out_t result = /* .. */;

        assign(databeat, result, v);
    }

    outStream << dataBeatOut;
    // ...
}
```

Memory Instances for User-Defined Operators

In order to support the development of user-defined operators, abstractions are provided for describing common memory instances used in image processing, two of which are line buffers and a sliding window.

Definition 4.25 (Line Buffers)

A line buffer of size kernel-height rows, image-width columns, and data type `pixel_t` can be specified as follows:

```
LineBuffer<image_width, kernel_height, pixel_t> line_buffer;
```

The shift and read operations are defined on a line buffer. The line buffer's shift operation stores a pixel (or a data beat) to its FIFO buffer and returns a column of pixels as follows:

```
// update line buffer with a new pixel and return a column of
    pixels
linebuf.shift(col_of_pixels, new_pixel);          // way1

auto col_of_pixels = linebuf.shift(new_pixel);    // way2
```

The pixels stored at a line buffer can be read without shifting its content as follows:

```
// reads the column of pixels at the position col
line_buf.read(col);

// reads the pixel stored at the position row and col
line_buf.read(row, col);
```

Definition 4.26 (Sliding Window)

A sliding window can be specified as follows:

```
SlidingWindow<width, height, pixel_t> swin;
SlidingWindow<width, height, pixel_t, p_factor> swin_vect;
```

It takes as template parameters window width, height, and optionally a parallelization factor (`p_factor`). The pixels in a sliding window are horizontally shifted by `p_factor` steps every time the `shift` function is called. The `shift` function takes as input an array of databeats (which is `pixel_t` in the case of `p_factor = 1`) to update the content of the sliding window.

```
// update the sliding window with a column of pixels
swin.shift(col_of_databeats);

// return the pixel/databeat at the position row and col
auto pixel    = swin.read(row, col);
auto databeat = swin.read_vect(row, col);

// return the column of pixels/databeats at the position col
auto col_of_pixels = swin.get_col(col);
auto col_of_pixels = swin.get_col_vect(col);

// return the row of pixels/databeats at the position row
auto row_of_pixels = swin.get_row(row);
auto row_of_pixels = swin.get_row_vect(row);
```

```
// get an array pointer to a window of pixels
auto ptr_to_window = swin.get_window(offset_x, offset_y,
                                      width, height);
```

Either individual pixels or databeats (pack of pixels) stored in a (2D) sliding window can be read (row-wise or column-wise) for a given position row and col. Additionally, we provide `get_window` function to return a pointer to a window of pixels for given specific offset x and offset y. This function is beneficial to extract multiple windows from a larger sliding window, as used in loop coarsening.

4.5.3 An Example Application: Harris Corner Detection

As an example, in Listing 4.5, we show the description of a Harris corner detector application by using our library. Its dataflow is shown in Figure 4.7. The algorithm works as follows:

1. Sobel local operators calculate horizontal and vertical derivatives of an input image.
2. Multiplication and squares of the derivatives are calculated through point operators and then smoothed by a Gaussian kernel.
3. Finally, the ranking of every pixel, the determinant of the derivatives, is calculated. Then the image's corners are selected through binary thresholding within the last point operator.

Listing 4.5 describes the behaviour of this application as follows: The operators and streams instantiated in Line 19-Line 39 resembles very much the structure of a system-level hardware implementation of a pipelined streaming application. Hipacc generates the above code (and the kernel functions) for the described Harris Corner Detection algorithm automatically (see VHLS-code.cpp in Figure 4.5). Hipacc's

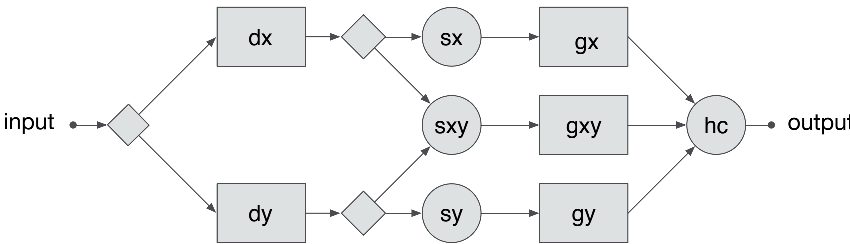


Figure 4.7: DFG for stream-based implementation of Harris corner detection (*line*: stream, *diamond*: split stream, *square*: local, *circle*: point). (Figure reprinted from [ÖRH+17a], © 2017 IEEE)

Listing 4.5: Harris corner code that generated by Hipacc. It uses uses our proposed library. The kernel descriptions are not shown for simplicity.

```
1  #define W 1024      // Image Width
2  #define H 1024      // Image Height
3  #define p_factor 1  // Parallelization factor
4
5  // Parallelized data type
6  using DataT = uchar;
7  using VecDataT = vectorize_t(DataT, pFactor);
8
9  // Local operator definitions
10 localOp<W, H, 3, 3, pFactor, DataT, MIRROR> sobelX, sobelY;
11 localOp<W, H, 5, 5, pFactor, DataT, MIRROR> gaussX, gaussY, gaussXY;
12
13 // Hardware top function
14 void harris_corner(hls::stream<VecDataT> &out_s,
15                    hls::stream<VecDataT> &in_s) {
16     #pragma HLS dataflow
17
18     // Stream definitions
19     hls::stream<VecDataT> in_sx, in_sy, Dx_s, Dx_s1, Dx_s2, Dy_s,
20         Dy_s1, Dy_s2, Dxy_s, Mx_s, My_s, Mxy_s, Gx_s, Gy_s, Gxy_s;
21
22     // calls for point and local operators
23     split_stream(in_sx, in_sy, in_s);
24
25     sobelX(Dx_s, in_sx);
26     sobelY(Dy_s, in_sy);
27
28     split_stream(Dx_s2, Dx_s1, Dx_s);
29     split_stream(Dy_s2, Dy_s1, Dy_s);
30
31     PointOp<pFactor>(Mx_s, Dx_s1, square_kernel);
32     PointOp<pFactor>(My_s, Dy_s1, square_kernel);
33     PointOp<pFactor>(Mxy_s, Dy_s2, Dx_s2, mult_kernel);
34
35     gaussX(Gx_s, Mx_s, gauss_kernel);
36     gaussY(Gy_s, My_s, gauss_kernel);
37     gaussXY(Gxy_s, Mxy_s, gauss_kernel);
38
39     PointOp<pFactor>(out_s, Gxy_s, Gy_s, Gx_s, threshold_kernel);
40 }
```

description does not require any tool-specific pragmas or metaprogramming techniques. It applies additional optimizations such as constant propagation and loop unrolling for generating the kernel code. In addition, a developer can extend the library without any need to change the Hipacc compiler. For instance, modifying the implementation of a local operator requires changing the implementation of the

`localOp` class, which is significantly easier than modifying Clang AST from Hipacc's HLS backend.

4.5.4 A Deeper Look into the Library

Achieving the *best* results for different parameter specifications of an algorithm (e.g., a convolution for different filter coefficients and window sizes) is not an easy task since the *best implementation* in hardware depends on the design objectives and the resource constraints. Hardware design, thus HLS, is susceptible to acquiring multiple Pareto-optimal design points for the same algorithm, which minimizes different resource types but facilitates the same performance. For instance, a Pareto-optimal architecture might require less BRAM, while another uses fewer LUTs. In this case, an efficient algorithm implementation requires the less budget-critical type of hardware resources and thus maximizes the desired design objective, i.e., throughput or cost. Moreover, a Pareto-optimal implementation of an algorithmic instance typically depends on the values of the input parameters of the algorithm. For example, an optimal implementation of an image border handling specialized for a small kernel might provide a poor throughput or cost for a large kernel.

We address the latter challenge by including more than one template design for each image processing operator in our proposed library. This means that a user of our library or a design space exploration tool can select trade-off solutions from a set of design points as part of the decision making. Unlike previous approaches, which instantiate the *same* implementation for a whole range of input parameter values, our approach sustains high-quality synthesis results for different specifications. Furthermore, even selecting the most suitable implementation from a set of design points is hidden from users when the design trade-offs are analyzed for the possible input parameters. This mechanism is possible for the coarse-level parallelization strategies as shown in Section 3.6.

In our library, different optimization strategies of a local operator are implicitly utilized from a high-level generic description, as discussed in Section 4.5.4. Furthermore, we apply several bit-level optimizations summarized in Section 4.5.4.

Policy-Based Design

Users of modern HLS tools can leverage the benefits of object-oriented programming by using C++ classes. Polymorphism allows providing a single interface from different types (which could be a C++ class). However, dynamic polymorphism is not supported by most of the HLS tools (e.g., not supported in Vivado HLS), where the type of a variable is defined only at runtime.

Our proposed library has a policy-based structure to offer multiple implementations of a class (high-level abstraction such as a local operator) when there is a trade-off. The term *policy-based design* is a C++ idiom for customizing the behavior of a template class at compile time for various optimization strategies, so-called *policies* [Ale01]. As an example, the unified modeling language (UML) diagram of the implementation of a local operator in the proposed library is shown in Figure 4.8. A local operator is instantiated through three composite object classes. First, the registers and the shifting mechanism of the sliding window are set according to the selected loop coarsening policy. Then selections, and thus, final data assignments are determined through a border handling class, which contains a loop coarsening policy class through composition. Finally, the control path policy sets the corresponding local operator schedule for the selected border handling policy.

One can explicitly create a local operator object according to a supported loop coarsening and border handling policy, for instance, using the functions below:

```
// W: image width, H: image height, PF: parallelization factor

auto local_op = make_local<W, H, PF, CoarseningPolicy>(local_kernel);

auto local_op = make_local<W, H, PF, BoundaryMode, BorderPolicy>(
    local_kernel);

auto local_op = make_local<W, H, PF, BoundaryMode, CoarseningPolicy>(
    local_kernel);
```

The supported policies for `CoarseningPolicy` and `BorderPolicy` are listed in Tables 4.4 and 4.5. Their implementation techniques are explained in Chapter 3. `FetchAndCalc` requires implementing a larger sliding window than the one `CalcAndPack` requires, as shown in Figure 4.9. Both `FetchAndCalc` and `CalcAndPack` provide the same latency and throughput. In `CalcAndPack` schedule, every output data beat is calculated in two consecutive cycles, but the execution starts earlier than `FetchAndCalc`. Correspondingly, the implementation of border handling circuits must be tailored to the selected loop coarsening policy. Therefore, despite doing the same task and providing the same interface, the implementation of a local operator is changed drastically according to the selected policies. According to the policies defined by the user, C++ metaprogramming allows generating the corresponding code at compile time . By integrating our library into Hipacc's source-to-source compiler, we alleviated the task of generating different policies to C++ compiler.

Automatic architecture selection

Explicitly selecting an implementation policy requires the users of our proposed library to understand implementation techniques and offered trade-offs, e.g., understanding loop coarsening and border handling techniques. We solved this problem by

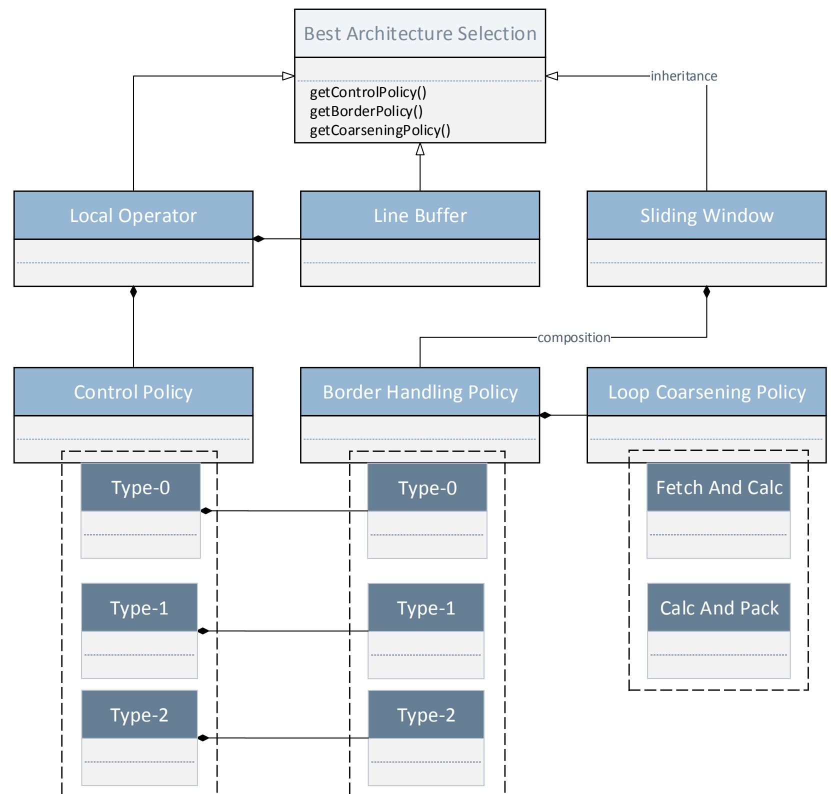


Figure 4.8: Policy-based structure of the proposed library. The object relationship diagram of a local operator has three border handling policies and two loop coarsening policies. The policies are defined as template classes for parametrization, and dynamic linking is avoided. Furthermore, it consists of an architecture selection algorithm that implements the analytical model shown in Section 3.6 to pick the standard best coarsening policy at compile time according to input template parameters. (Figure reprinted from [ÖRH+17a], © 2017 IEEE)

Table 4.4: Supported loop coarsening policies (`CoarseningPolicy`).

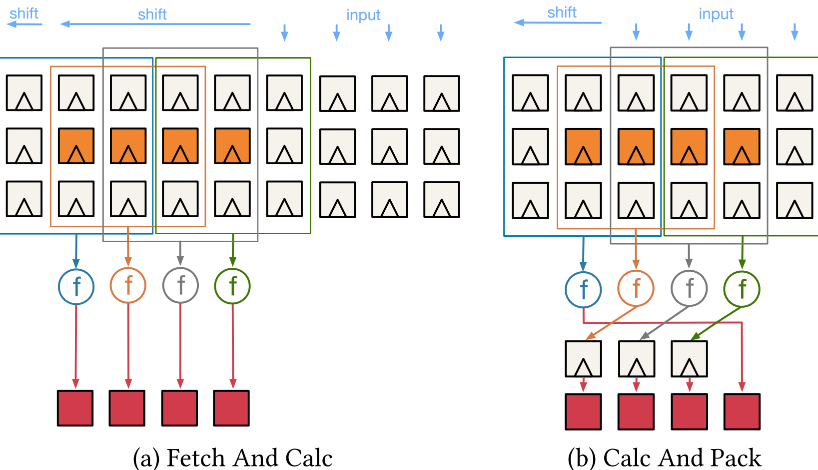
`CoarseningPolicy`	Loop Coarsening Type
`FetchAndCalc`	Fetch and Calculate
`CalcAndPack`	Calculate and Pack

implementing a policy selection algorithm and/or providing design objective settings for the user.

In Chapter 3, we analyzed the resource usage of loop coarsening and border han-

Table 4.5: Supported image border handling policies (`CoarseningPolicy`).

`BorderPolicy`	Border Handling Type
`Type-0`	base method
`Type-1`	minimizes the area (*default*)
`Type-2`	optimizes achievable clock speed



(a) Fetch And Calc (b) Calc And Pack

Figure 4.9: Considered loop coarsening architectures for a 3-by-3 kernel where the parallelization factor is 4.

dling techniques as supported in our proposed library. As discussed in Section 3.6.3, either one of the loop coarsening policy uses less resources depending oon instantiation parameters such as the local operator window dimensions, bit widths of input and output data types, and parallelization factor. We implemented this as a meta function that is evaluated at compile time. Thereby, by checking the corresponding template parameters of the specification of a local operator, a meta function selects one of the coarsening policies. Then, the local operator class is created by using the selected policy. All this happens implicitly when the user creates a local operator as simple as shown below:

```
// W: image width, H: image height, PF: parallelization factor
auto local_op = make_local<W, H, PF>(local_kernel);
```

However, the border handling policies offer different trade-offs. `Type-0` uses more resources than `Type-1` and `Type-2` but does not change the structure of the sliding window. Therefore, `Type-0` is needed if our local operator implementation is extended to take multiple image border handling algorithms, such as mirror, and clamp, as a runtime parameter. `Type-2` minimizes the critical path, thus increasing the clock

Table 4.6: Supported design objectives (`DesignObjective`).

`DesignObjective`	objective of architecture selection
`LessLUT`	minimizes LUT usage (*default*)
`FasterClock`	optimizes clock speed

speed, whereas `Type-1` requires less LUT and mostly fewer area resources. The users of our proposed library do not need to remember all these low-level implementation details. Instead, they can select a design goal listed in Table 4.6, as shown below:

```
auto local_op = make_local<W, H, PF, DesignObjective>(local_kernel);
```

Bit-Level Optimizations

Modern HLS tools benefit from bit-level optimizations [ÖRH+16; ÖRH+17a; Xil17c; Int17]. We have tuned specification of the library implementations with various low-level optimizations known for designing hardware at RTL. Some of these bit-level optimizations are explained below:

(i) *Bit-level precision for the variables:* In the library, we specified all the variables with bit-level precision using the data structures provided by Xilinx Vivado HLS.

(ii) *Compile-time flags to exploit bit-specific properties of input specification:* Often, the hardware implementation of arithmetic operators, comparisons, and logical blocks can be optimized for specific characteristics of the input parameters. We exploit this in our library by creating compile-time flags whose values depend on the template parameters. For instance, resource requirements to count the coordinates of an image can be reduced when the image width is a power of two, as shown in Listing 4.6[3]. The Boolean value of `is_image_width_power_of_two (image_with)` in Line 11 depends on `image_width`. The if block (Line 13 to Line 16) is optimized for the case of image width being a power of two and only requires wire assignments. The else block (Line 20 to Line 27), however, requires adders and multiplexers to implement counters for tracking the image coordinates of the input pixel. The final program includes only the if block or the else block depending on the template parameter `image_width` since C++

[3] Note that writing a top-level loop for the whole iteration (instead of nested loops) is recommended by the optimization guides [Xil17c]. The `clock_tick` tracks the iteration of the whole image, whereas `col_im` and `row_im` track the horizontal and vertical indices of the pixel read from the input image. C++ compilers generate code specialized to template parameters before compiling the program. Hence, depending on the template parameters, either the if block (Line 13 to Line 16) or the else block (Line 20 to Line 27) will be compiled in the final program.

compilers generate code specialized to template parameters before compiling the program.

Listing 4.6: Example for reducing the resource usage (by is_image_width_power_of_two (image_with), where the image_width is a template parameter).

```
1  template<image_width, image_height, /* .. */>
2  function(/* .. */)
3  {
4      // ..
5
6      for(BW(Latency) clock_tick = 0; clock_tick < Latency; ++i)
7      {
8          // ...
9
10         // compile-time selection based on template parameters
11         if(is_image_width_power_of_two(image_with) == true)
12         {
13             // this block uses less resource than the else block
14             col_im = clock_tick[BW(col) - 1 : 0];
15             row_im = clock_tick[BW(row) + BW(col) - 1 : BW(col)];
16             new_col= (col_im == image_width - 1);
17         }
18         else
19         {
20             // this block uses more resource than the if block
21             new_col = false;
22             col_im++;
23             if(col_im == image_width)
24             {
25               col_im = 0;
26               row_im++;
27               is_col_read = true;

28             }
29         }
30     }
31     // ..
32 }
```

(iii) *Exploiting similarities in expressions:* We optimize resource usage by exploiting sub-common expressions and concisely using their outputs to express other parts of the hardware. For instance, in Listing 4.7, we assign the results of comparison circuits to 1-bit temporary variables, which are later used for the program's parts that require the same comparisons.

(iv) *Exploiting temporal locality:* We exploit the locality of a given algorithm by designing a particular memory architecture (e.g., line buffers) and using hardware design techniques such as deep pipelining. In this context, we even pipeline the control signals when possible. For instance, border handling of

Listing 4.7: Optimization of the sub-common expressions by assigning the intermediate results to temporary variables and using them later as part of other expressions.

```
// ..

if(is_initial_latency_passed ) {/* .. */}

// ..

if (is_initial_latency_passed && other_logical_expression) {/* .. */}

// ..

is_initial_latency_passed = (clock_tick > initial_latency);
```

a local operator of a w width requires $w - 1$ number of comparisons (e.g., a 5-by-1 window iterating over an image whose width is 1024 will be out of the boundary when the horizontal coordinate is 0, 1, 1022, and 1023). However, our implementation requires only one comparison in the horizontal access for checking if a local operator enters the border handling area. Then it uses pipelined registers to handle the control signals accordingly. For instance, for an image width 1024, instead of checking at the horizontal coordinates 0, 1, 1022, and 1023, our implementation checks if the column counter is at 1022 for a raster order scan, then uses shift registers to produce corresponding flags for other column coordinates). A representative code is shown in Listing 4.8.

The hardware-centric optimizations discussed in this section significantly increase the performance but complicate the code. However, since we hide the implementation of the library through high-level abstractions, the tedious description of the hardware is not exposed to library users.

Listing 4.8: Exploiting the temporal locality of the control flow by using shift registers

```

// shift registers to exploit the temporal locality of the control flow
for(int i = width - 1; i > 0; i--){
  border_flags[i] =  border_flags[i - 1];
}

// does the window enter the out-of-image border
border_flags[0] = is_col_read;
```

4.6 Evaluation and Results

This section presents evaluations of i) our Hipacc backend targeting Intel FPGA SDK for OpenCL and ii) the proposed library for Xilinx Vivado HLS.

4.6.1 Algorithms

In the following, the algorithms that were chosen for evaluation throughout this section are briefly introduced.

GB The Gaussian blur first applies a 3×3 convolution with a specific mask of unsigned integers that total 16, followed by a normalization (division) by 16.

LP The Laplacian filter detects vertical, horizontal, and diagonal edges using signed integer arithmetic with a 5×5 local operator.

SB The Sobel filter first computes vertical and horizontal derivatives with 3×3 masks then calculate the Euclidean distance and clamps with a given threshold in the third kernel to detect edges.

LSB An edge detection that first transforms an RGB input to LUMA, then applies horizontal and vertical Sobel filters and finally clamps the sum of the absolutes of derivatives according to a threshold.

BL The bilateral filter [TM98] is a 3×3 local operator used for reducing noise while preserving edges. It consists of an exponential function and employs floating-point arithmetic.

HC The Harris corner consists of 9 kernels. First, the horizontal and vertical derivate of the input image are computed, then, the derivatives are squared and multiplied with each other within 3 point operators. The resulting images are blurred by 3 Gaussian kernels. Finally, after calculating the determinant, a point operator detects corners by clamping with a threshold.

OF The optical flow issues a Gaussian blur and computes a bit vector signature for every pixel of two input images using the census transform [Ste04]. Those signatures are then compared within a 15×15 window to find corresponding points and therefore detect the optical flow. Overall, five kernel executions are involved.

LK Lucas Kanade [Bou01] is a dense optical flow that utilizes nine kernels with floating point arithmetic. The first difference between two input images and their derivatives are computed. Then their multiplications with each other are accumulated in a $7{\times}7$ window to find motion vectors. Finally, corresponding *Munsell* color indexes, including square and arctangent, are calculated.

Table 4.7: Synthesis results for different loop coarsening factors v of a 3×3 bilateral filter with clamping applied on an image of size 1024×1024 for an Intel Stratix V DE-5.

v	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
1	1	54490	82780	22.77	373	23	305.53
2	1	60283	89573	24.45	371	37	304.50
4	1	71772	103836	27.91	371	65	304.50
8	1	92927	118080	34.28	375	121	256.73
16	1	140368	189010	48.93	381	233	255.75

4.6.2 Hipacc Compiler Backend Targeting Intel FPGAs

This section evaluates the source-to-source compiler backend generating OpenCL code for Intel FPGA SDK for OpenCL from Hipacc DSL as introduced in Section 4.3. We evaluate implementation results for varying image processing applications and compare handwritten implementations given in [Int23]. Furthermore, we compare the performance of our FPGA implementations with a server-grade discrete GPU and a multiprocessor system-on-a-chip (MPSoC) hosting an embedded GPU. We used the same Hipacc application to generate target-specific code for these GPUs and our target FPGA. This shows one of the benefits of our DSL-based approach, where portability of performance is achieved from a high-level declarative description.

Loop Coarsening Optimizations

Figure 4.10 and Table 4.7 show the resource usage and clock speed of a bilateral filter for different parallelization factors. It can be observed that a speedup of roughly 13.4 is achieved compared to non-parallelized implementation (where $v = 1$) at the cost of an increase of only 2.1× in logic utilization and 10.1× in DSP blocks thanks to our loop coarsening optimization applied from our Hipacc backend. In our implementation, parallelization by a factor of $v = 16$ increases the size of the sliding window from $n \times m$ to $(n + v - 1) \cdot m$, and increases the size of line buffers to read and write v number of pixels, and replicates the arithmetic units, thus increases resource usage. Despite processing 16 pixels in every clock cycle, the maximum achievable clock frequency degrades for the larger design to 255.57 MHz; therefore, the parallelization optimizations provide the 13.4× speedup. Furthermore, we observed that HLS compilers eliminate redundant computations within multiple merged local windows and optimize the logic better when only the arithmetic operations in the innermost kernel are replicated. As a result, we increase the throughput by approximately a factor of v at the cost of a sublinear increase in logic.

An alternative parallelization approach is to use the parallelization intrinsics

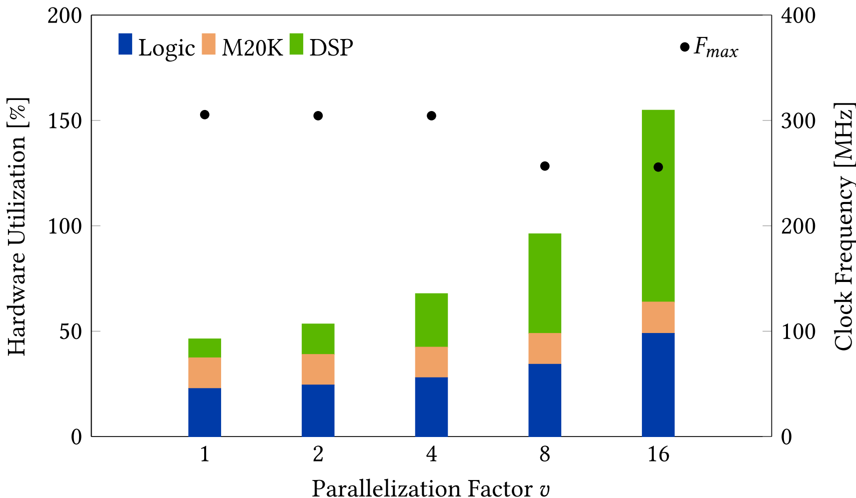


Figure 4.10: Hardware utilization for the loop coarsening of a 3×3 bilateral filter with clamping on an image of size 1024×1024 for an Intel Stratix V DE-5. (Figure reprinted from [ÖRH+16], © 2016 IEEE)

offered by Intel, namely *num_compute_units(n)* and *num_simd_work_items(n)*. The *num_compute_units(n)* intrinsic replicates the whole acceleration unit as shown in Figure 4.11. These parallelization strategies follow OpenCL's many-core computing paradigm. However, this means caching mechanisms such as line buffering need to be modified to share data across the computing units, which is not supported by Intel FPGA SDK for OpenCL. The *num_simd_work_items(n)* intrinsic tries the replicate only the arithmetic units similar to designing vectorization units in a processing unit, as shown in Figure 4.12. Ideally, this optimization would generate a circuit similar to our loop coarsening technique. However, modern HLS tools fail to achieve this efficiency for a given OpenCL code written for its many-core paradigm. This is not surprising since generating vectorization instructions for a fixed programmable device (such as a CPU) from a program (e.g., written in C++ without using vectorization instructions) is a challenging task where researchers have not yet provided a satisfying solution despite decades of work [CCF+10; RKH+17].

Intel FPGA SDK for OpenCL supports using the parallelization intrinsics to NDRange kernel written according to OpenCL paradigm (not to a single item kernel describing an application-specific memory architecture such as line-buffering). Figure 4.13 shows the implementation results for a Gaussian filter. It can be seen that the line-buffered solution generated by Hipacc uses significantly fewer resources while providing up to 5× higher throughput than the implementations that can be achieved by using Intel's parallelization intrinsics. Hipacc's implementation can be accelerated even more by using loop coarsening. As a result, we conclude that

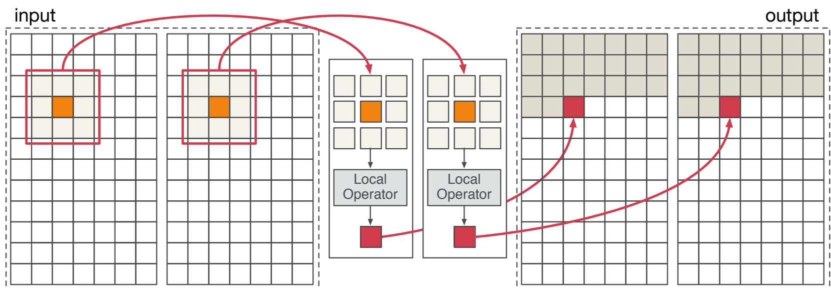


Figure 4.11: Automatic replication of the entire accelerator by specifying `num_compute_units(n)`. (Figure reprinted from [ÖRH+16], © 2016 IEEE)

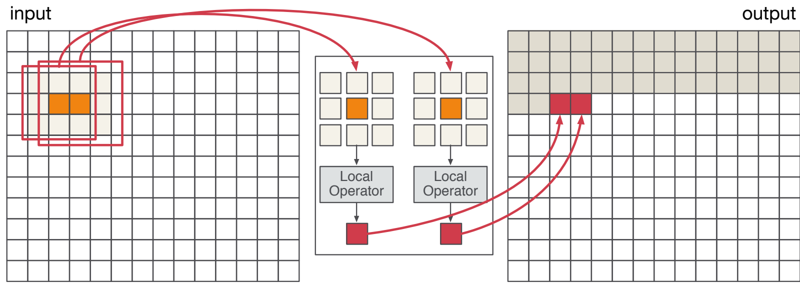


Figure 4.12: Automatic replication of the innermost kernel computation by specifying `num_simd_work_items(n)`. (Figure reprinted from [ÖRH+16], © 2016 IEEE)

our DSL-based approach using source-to-source compilation performs significantly better than compiler-based acceleration methods applied to a regular OpenCL code (e.g., written for a GPU).

Comparison with Handwritten Applications provided by Intel

Figure 4.14 and Table 4.8 present the implementation results on an Intel Stratix V FPGA for a number of image processing design examples provided by Intel [Int23] and the equivalent Hipacc implementations. The design examples from Intel consist of single work-item line buffered kernels similar to the OpenCL codes generated by Hipacc for the Intel FPGA SDK. All arithmetic operations in Hipacc's operators are written the same way as specified in design examples for comparison. Hipacc's implementations are assembled from two kernels (LSB) and four kernels (LK), coupled by channels, whereas Altera's are from a single kernel only.

Hipacc's LSB uses significantly fewer DSP blocks and slightly fewer logic in

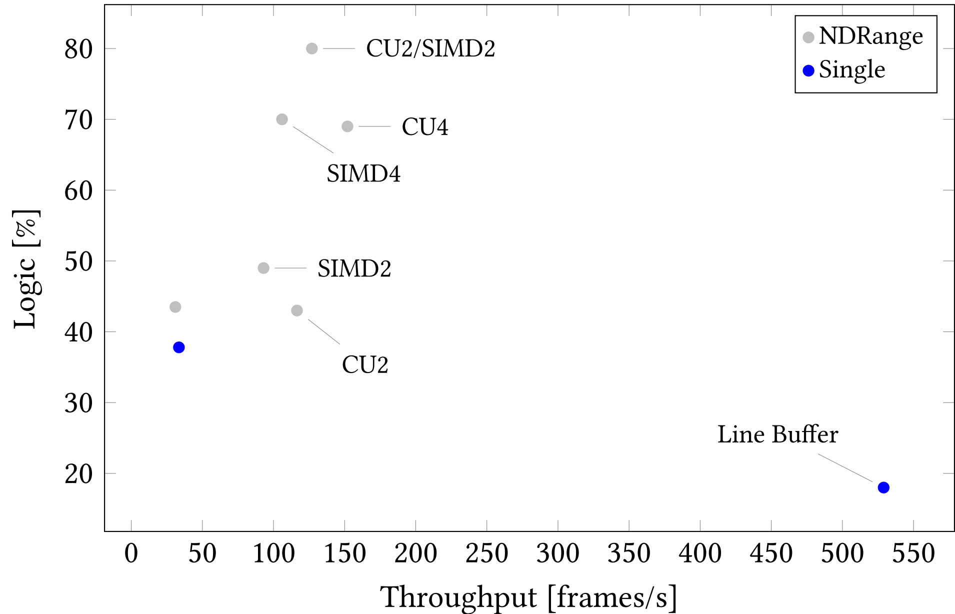


Figure 4.13: Design points of NDRange and single work-item kernels for a Gaussian blur filter. NDRange kernels have been varied in automatic replication of the entire accelerator (CU) and innermost kernel computation (SIMD). (Figure reprinted from [ÖRH+16], © 2016 IEEE)

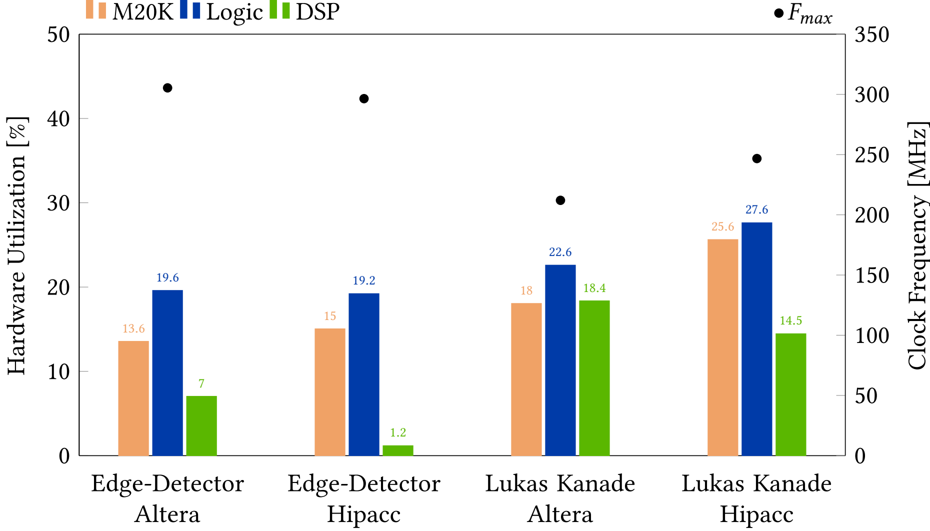


Figure 4.14: Solutions generated using our Hipacc DSL vs. handwritten examples [Int23] for a 1024×1024 image size and an Intel Stratix V FPGA target. (Figure reprinted from [ÖRH+16], © 2016 IEEE)

Table 4.8: Comparison of handwritten examples [Int23] and Hipacc (image size is 1024×1024) [ÖRH+16].

Filter	Source	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
LSB	Intel	1	43844	67418	19.59	347	18	305.42
	Hipacc	1	45069	68624	19.95	346	3	304.50
LK	Intel	1	53768	84746	22.59	462	47	212.03
	Hipacc	1	63595	92810	24.87	562	37	289.85

exchange for more on-chip memory, M10K. The difference in clock frequency is negligible. Hipacc's LK consumes 1.4× M20K blocks of Inte's design uses. However, the achieved clock frequency using Hipacc is significantly higher. We argue that the difference in logic utilization is acceptable when the number of used DSPs is considered. The reason for obtaining considerably different results from LK in comparison to those from LSB, are algorithm-specific manual optimizations applied in the handwritten design examples by Intel, where the average filters are implemented using shift registers. This is an optimization we can add to our Hipacc backend in the future.

In summary, our DSL-based Hipacc source-to-source compilation approach achieves the quality of results close to hand-optimized implementations provided by Intel.

Furthermore, users of Hipacc can easily modify their implementation for different boundary conditions, which is often a requirement for LK, or efficiently increase the throughput (via loop coarsening) without changing the application code. In contrast, these require drastic changes in the hand-written implementations [Int23].

Comparison with GPUs

We can generate GPU implementations from exactly the same DSL source code since all OpenCL codes used for evaluation are generated by Hipacc's domain-specific compiler. We generate CUDA implementations due to the lack of OpenCL support in recent versions of NVIDIA's programming environment. In the generated CUDA codes, all GPU implementations are carefully tuned by considering the use of shared memory and texture memory as well as by applying optimizations, such as thread-coarsening, to ensure a fair comparison.

Figure 4.15 shows the throughput comparison of Tegra K1 and Tesla K20. The exact numbers for the presented Stratix V FPGA implementations are given in Table 4.9. Algorithms are listed in order of complexity, mainly in terms of required bandwidth. In particular, the performance of GPU architectures suffers from increasing memory access. Therefore, the GPU throughput for the Gaussian blur, with a small 3×3 local window is rather high compared to the optical flow, for which a 15×15 local window needs to be processed.

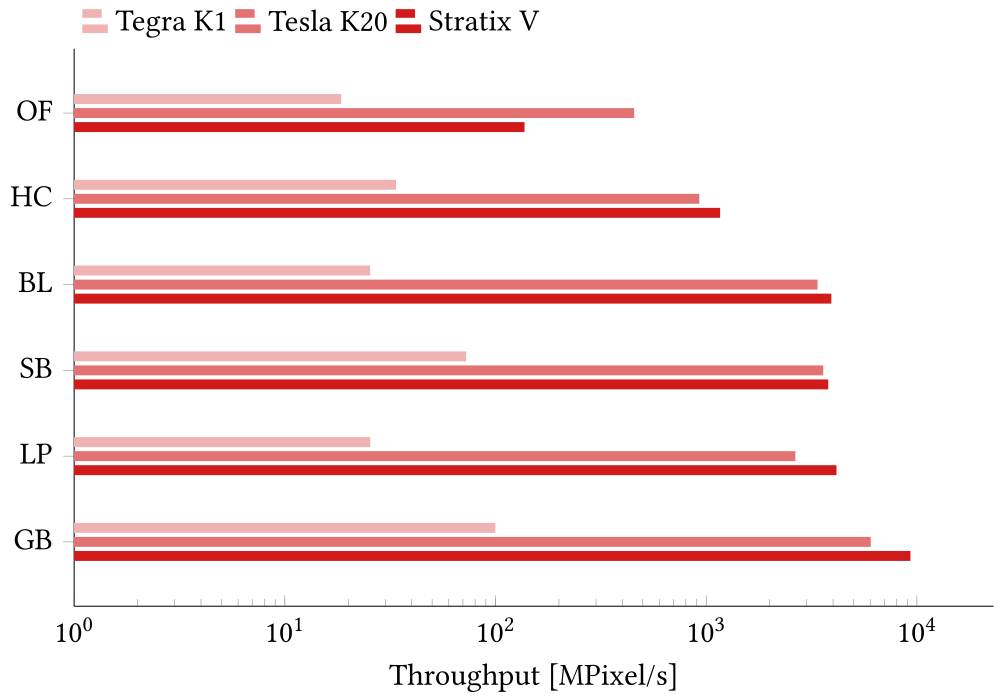


Figure 4.15: Comparison of throughput for the NVIDIA Tegra K1 GPU, NVIDIA Tesla K20 GPU, and Altera Stratix V FPGA. (Figure reprinted from [ÖRH+16], © 2016 IEEE)

Table 4.9: Synthesis results of multiple image processing algorithms with different loop coarsening factors v [ÖRH+16].

Filter	v	#Kernels	II	ALUTs	Registers	Logic (%)	M20K	DSP	Freq [MHz]
GB	32	1	1	47045	73584	20.64	363	0	303.58
LP	16	1	1	107310	142069	39.69	419	64	270.62
SB	16	3	1	58308	96673	24.59	497	96	247.34
BL	16	1	1	140368	189010	48.93	381	233	255.75
HC	4	9	1	135808	192397	45.86	493	36	303.39
OF	1	5	2	81551	128816	32.61	646	18	286.36

The throughput of image processing implementations on Stratix V FPGA does not diminish as much for larger local window sizes as the GPU implementations. Increasing the line buffer sizes lead to almost double the M20K usage when comparing the Gaussian blur with the optical flow. However, FPGA implementations suffer from resource limitation for the implementations where the loop coarsening factor v couldn't be increased to more than 4 without increasing II of 1. Unfortunately, even without loop coarsening, synthesis could not maintain an II of one 1 for the

optical flow. Nevertheless, for the Gaussian blur, bilateral filter, and Harris corner, the Stratix V could even outperform the server-grade Tesla K20 GPU.

Our approach eliminates the need for platform expertise and allows targeting different computing platforms (in this case, GPUs and an FPGA) without changing the application code. Using Hipacc's source-to-source compiler, we can generate highly optimized target-specific code without tuning the code at low-level using different programming languages. This allows analyzing a program for different computing platforms and further opportunities for advanced design space exploration techniques for heterogeneous systems.

4.6.3 The Metaprogramming Library Targeting Xilinx Vivado HLS

This section evaluates our proposed library by investigating implementation results for varying image processing algorithms. We targeted a Zynq xc7z100ffg900-2 FPGA, in which a rate of 1,024 bits per cycle can be reached for streaming the data from the main memory to the reconfigurable logic. We provide implementation results for the Vivado HLS IP blocks obtained through Vivado HLS 2018.2 using the *export* feature.

One of the main advantages of using Vivado HLS is the ability of setting different speed targets without changing the code. The logic is highly pipelined in the case of ambitious speed targets, causing the utilization of additional registers between LUT. Therefore, different target speed constraints should be considered when the implementation results of an HLS code are evaluated. Moreover, we observed that the drastically high resource utilization for the same speed or relatively less logic speed despite the same area could be acquired when too ambitious or too relaxed target speed constraints are set. In this section, we evaluate our algorithms with two different target speed constraints, which are 50 MHz and 200 MHz. As expected, the higher speed constraint slightly increases the number of registers and, thus, the initial latency for all the architectures. Moreover, architectural differences' effects on implementation results are amplified with faster logic frequencies.

Quality of Synthesis Results

In Chapter 3, we thoroughly investigated the implementation of the proposed loop coarsening and border handling techniques by using the Vivado HLS tool. In the following, we show that the HLS results of our C++ code use the expected number of resources, where we mathematically counted the required number of LUT, FF, and BRAM resources. All the presented results in Chapter 3 have been acquired using our metaprogramming library. Thus, we already showed that the proposed library provides good results for stencil-based applications. In this section, we provide

synthesis results for a Gaussian filter for different parallelization factors, border handling modes, and clock speed targets.

Table 4.10 compares different border modes regarding resources used to implement a 5-by-5 integer Gaussian filter. The border handling mode *UNDEFINED* does not introduce additional cost (i.e., reads a garbage value from line buffers). The constant border handling mode is the cheapest in terms of resources. A higher parallelization factor simplifies the data selection in border handling, as also discussed in Chapter 3. Thus, the implementation overhead of all the considered border handling modes

Table 4.10: 5-by-5 Gaussian filter with different border handling modes for a 1024×1024 grayscale image [ÖRH+17a]. The results are obtained for 50 MHz and 200 MHz speed constraints. v denotes the coarsening factor (see Chapter 3).

v	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEFINED	94	248	249	4	0	4.0	1050630
1	CONST	119	301	360	4	0	4.4	1050630
1	CLAMP	121	335	361	4	0	3.6	1050630
1	MIRROR	128	356	361	4	0	3.8	1050630
1	MIRROR101	131	374	361	4	0	3.8	1050630
32	UNDEFINED	2238	6099	2683	32	0	6.2	32838
32	CONST	2431	6246	3035	32	0	7.3	32838
32	CLAMP	2406	6935	3042	32	0	8.3	32838
32	MIRROR	2240	6927	3042	32	0	7.1	32838
32	MIRROR101	2291	6928	3042	32	0	7.4	32838

(a) Target speed is 50 MHz.

v	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEFINED	157	316	625	4	9	2.3	1050637
1	CONST	198	374	881	4	10	2.1	1050639
1	CLAMP	222	453	922	4	9	2.3	1050639
1	MIRROR	226	511	922	4	10	2.3	1050639
1	MIRROR101	215	508	882	4	9	2.3	1050639
32	UNDEFINED	3211	8032	13380	32	202	2.8	32845
32	CONST	3465	8094	15607	32	198	3.0	32846
32	CLAMP	3591	8890	16188	32	200	2.8	32846
32	MIRROR	3576	8939	16272	32	200	3.4	32846
32	MIRROR101	3648	8928	16280	32	195	3.3	32846

(b) Target speed is 200 MHz.

converges for large parallelization factors.

Table 4.11 shows the implementation results of the same Gaussian filter for different parallelization factors. It is shown that the increase in throughput is linear to the parallelization factor, while the increase in resource usage is sublinear.

Table 4.11: 5-by-5 Gaussian filter with different coarsening factors (v) for 1024×1024 grayscale images [ÖRH⁺17a].

v	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEF	105	247	249	4	0	3.4	1050631
2	UNDEF	156	438	295	4	0	3.3	525319
4	UNDEF	265	818	388	4	0	8.6	262663
8	UNDEF	477	1576	577	8	0	9.5	131335
16	UNDEF	981	3106	958	16	0	9.2	65671
32	UNDEF	1919	6160	1723	32	0	9.3	32839

(a) Target speed is 50 MHz.

v	Brdr Mode	SLICE	LUT	FF	BRAM	SRL	CPimp	Latency
1	UNDEF	161	316	625	4	10	2.2	1050637
2	UNDEF	257	562	1029	4	16	2.4	525325
4	UNDEF	439	1058	1823	4	28	2.4	262669
8	UNDEF	818	2055	3381	8	52	2.5	131341
16	UNDEF	1563	4125	6506	16	101	2.8	65677
32	UNDEF	3092	8034	12416	32	204	3.1	32845

(b) Target speed is 200 MHz.

Comparison with Previous Work

In Table 4.12, we compare our library-based approach with previous work [RSH⁺14; RÖH⁺18] for varying image processing algorithms. The previous work [RSH⁺14] already generates correct and efficient circuits, e.g., eliminating unnecessary memory transfers via the support of line buffering, and achieves higher quality of results compared to Xilinx's OpenCV library [Xil23] in terms of throughput and resource usage. Both the C++ codes generated from our backend and the previous work instantiate the same streaming pipeline and data path functions. Table 4.12 shows that our library significantly improves the results for all the considered applications. This is partially due to the superiority of our loop coarsening techniques. More interestingly, the improvement in resource usage becomes more significant for the more complicated data path functions. For instance, the bilateral filter can only be

Table 4.12: Comparison of the library-based approach compared with Hipacc's previous Vivado HLS backend [ÖRH+17a].

Application	Framework	v	SLICE	LUT	FF	DSP	BRAM	SRL	CPimp	Latency
Mean Filter	proposed	1	106	206	409	0	4	0	2.96	1050633
		32	1698	4722	6073	0	32	1	4.16	32841
	previous work [RSH+14]	1	151	253	581	0	4	1	2.77	1052684
		32	2078	5008	8487	0	32	121	2.70	33866
Laplace	proposed	1	469	1126	1762	0	8	17	3.90	1050634
		32	12235	40157	33440	0	116	2	4.85	32842
	previous work [RSH+14]	1	581	11307	2057	0	8	0	3.88	1052684
		32	12430	41349	36514	0	116	1404	4.85	33868
Sobel Edge	proposed	1	1113	2809	4942	8	4	85	3.94	1049687
		32	26716	76667	137267	256	14	2560	4.73	33878
	previous work [RSH+14]	1	1138	2899	5028	8	4	85	3.82	1050632
		32	27770	83470	145072	256	32	2565	4.87	33878
Harris Corner	proposed	1	763	1731	2528	14	10	38	3.88	1049633
		32	8293	20017	31399	363	39	998	4.34	33825
	previous work [RSH+14]	1	936	2125	3086	15	10	72	4.15	1050637
		32	14739	37424	56691	480	80	1081	4.89	33837
Bilateral	proposed	1	6049	15691	18535	190	2	811	4.26	1049763
		8	38776	119123	135711	1520	4	5604	4.87	131364
	previous work [RSH+14]	1	15875	43859	50453	558	4	2638	4.48	1052967
		2	29669	85228	96159	1116	4	4307	4.84	526630

parallelized by the factor of 2 using Hipacc, whereas the factor of 8 was possible with our proposed library. In addition, the latency results of our proposed library fit the latency equations we showed in Chapter 3 and are smaller than the previous work on Hipacc.

4.7 Related Work

Exploiting parallelism or memory layout of an algorithm for a specific device from a general-purpose program like C++, Java, or Python is too difficult [HP19; CCF+10; LTE+20; RBA+13; SMB+16; BRR+19]. The EDA community has been looking at new approaches to alleviate the burden of compilers such as providing programming languages, language extensions, language constructs to provide higher level abstractions that can be mapped to different computing platforms [CCF+10]. As a result, drastically different programming paradigms and languages have emerged. For instance, CUDA or OpenCL for programming GPU accelerators, TensorFlow for programming deep neural networks (DNNs), C++ combined with OpenMP for parallel programming of multicore processors, vector data types, libraries, or intrinsics to utilize the SIMD units of CPU.

Early work that proposed the compilation of a data-parallel programming model to hardware implementations was presented with FCUDA by Papakonstantinou et al. [PGS+09]. Here, CUDA, heavily driven by the GPU industry and the main inspiration for the OpenCL standard, was adapted to serve as a source language. Employing a source-to-source compiler, CUDA thread blocks are transformed into parallel C code for AutoPilot [ZFJ+08], which is the predecessor of Xilinx' Vivado HLS. The Throughput Oriented Performance Porting (TOPP) framework [PCH+13] optimizes memory access by decreasing the lifetime of variables and removing synchronization points using FCUDA annotations. Shagrithaya, Kępa, and Athanas [SKA13] and Kępa, Soni, and Athanas [KSA15] use Xilinx Vivado HLS for kernel code generation and focus on designing application-specific kernel interfaces for fine-grained parallelism. On the other hand, Intel FPGA SDK for OpenCL targets royalty-free OpenCL to extract the thread-level parallelism that the developer specifies. Its compiler uses the LLVM infrastructure to transform kernels into dedicated hardware pipeline circuits that can be replicated many times [CAD+]. Intel FPGA SDK for OpenCL implements a complete heterogeneous system by facilitating all required memory and host interfaces for the targeted device via QSYS and Quartus integration tools. Silicon-OpenCL (SOpenCL) [OBD+11] is another LLVM-based compiler that automatically adjusts the parallelism level of a kernel for a specified FPGA using hardware templates. Gurumani et al. [GCL+13] use application-specific knowledge as well as analytical models for design space exploration, which is too complex in the case of multiple communicating kernels.

Despite the significant improvements in HLS methodologies and tools, hardware

design expertise is still a game changer in terms of good results. Similar to Intel FPGA SDK for OpenCL, users of a Xilinx Vivado HLS must write very specific code to obtain efficient hardware implementations through synthesis. For instance, the wait at the initial latency and stalling of the input should be manually described for a convolution. Therefore, Xilinx optimization guide [Xil17c] recommends and shows example codes tuned to resemble already to the structure of a circuit, unfortunately. Another solution could be using the OpenCV library of Vivado HLS [Xil23], which restricts a developer to use a pre-defined set of computer vision algorithms. However, the border handling implementation in [Xil23] applies padding, thus increases latency. A similar approach to our metaprogramming library, which provides template functions for point and local operators, is proposed in [SAH+14]. Neither the discussed solutions provided by Vivado nor the work in [SAH+14] facilitates any parallelization. Moreover, our implementations use fewer resources and achieve lower latencies than both, even without parallelization.

One promising direction to eliminate the required platform expertise for achieving good quality implementations without having to deal with low-level implementation details is developing DSLs that lift the description of a design to a higher level for a restricted domain of applications, to capture the parallelism and typically to map an application to different target platforms. For instance, TensorFlow supports both TPUs and GPUs. Hipacc [MRH+16; RÖM+17b] and Halide [RBA+13] are image processing DSLs that generate optimized code for CPUs and GPUs. These languages provide readability, portability, and modularity, hiding target- and device-specific optimizations.

PARO [HRD+08; Han09] and the FPGA version of Spiral [MFH+12] are prominent approaches of generating hardware accelerators from a DSL. PARO [HRD+08; Han09] is an HLS tool for generating highly parallel hardware accelerators from a high-level functional language for a broad variety of multi-dimensional dataflow dominant applications that can be described by nested for loop programs. PARO can be seen as the first DSL approach to synthesize processor array accelerators after initial attempts using polyhedral loop models such as [Tei93]. Spiral [MFH+12] generates HDL for a variety of digital signal processing applications. In the domain of image processing, previous work includes Darkroom [HBD+14], Rigel [HDD+16], and RIPL [SMB+16]. These approaches are able to generate HDL for an image processing DSL without using any HLS tool, but lacking the state-of-the-art scheduling, allocation, and binding methods of modern HLS tools. The Vivado HLS backend developed for Hipacc by Reiche et al. [RSH+14] is a prominent example of using a modern HLS tool for generating hardware accelerators from an image processing DSL. As part of this thesis, we improved its performance by extending [RSH+14] with the loop coarsening and image processing applications proposed in [ÖRH+17b] (see Chapter 3). Among other contributions, our work [ÖRH+16] is unique for being the first DSL approach that implements a complete heterogeneous system based on OpenCL (i.e., to the best

of our knowledge, our work [ÖRH+16] is the first DSL backend that targets Intel FPGA SDK for OpenCL.). Pu et al. [PBY+17] and Chugh et al. [CVP+16] followed our approach for the Halide [RBA+13], and the PolyMage [MVB15] DSLs, respectively. However, they only support generating hardware accelerators using Vivado HLS (i.e., cannot target Intel FPGAs and do not generate OpenCL host code). The Halide backend [PBY+17] generates software code to control the hardware accelerators but only for a specific device whereas generating OpenCL code as in our work allows creating a complete heterogeneous system, which is not specific to a device (i.e., our work leverages the industrial OpenCL standard API for host-device communication). Furthermore, to the best of our knowledge, already mentioned existing approaches (DSLs, application libraries tuned for HLS) do not provide different implementations depending on parameter of a library function offered for an algorithm specification. In this sense, our approach is more comprehensive, where we choose a implementation optimized for the given set of template parameters (that are different specifications of an algorithm).

4.8 Conclusion

In this chapter, we presented a source-to-source compiler approach for 2D image processing algorithms based on a DSL, more specifically to target Intel FPGA SDK for OpenCL using Hipacc DSL. Moreover, we introduced an image processing library tuned for Xilinx Vivado HLS, which is used to alleviate the tasks of Hipacc's Vivado HLS backend. The developed library can be utilized and extended by HLS users without modifying the Hipacc compiler. It provides high-level abstractions for image processing operators (i.e., point, local, and global operators), common memory instances such as sliding window and line buffers, and data structures for streaming. The library is implemented with application-specific deeply pipelined hardware descriptions optimized at bit-level. Furthermore, it supports parallelization of applications by the loop coarsening techniques introduced in Chapter 3.

Our approach leverages algorithm descriptions to a higher level to generate highly optimized target-specific code for Intel FPGA SDK for OpenCL and Xilinx Vivado HLS. We were able to show that our generic approach can lead to results close to those of hand-optimized applications provided by Intel. Despite comparable results, the generative approach remains superior as core fragments of the implementation (such as boundary conditions or the loop coarsening factor) can be changed without severe modifications. Furthermore, a server-grade GPU is outperformed in terms of throughput for various image filter algorithms. In conclusion, our proposed code generation from a DSL raises the abstraction level in the C-based HLS tools and, thus, eases the burden on the developer of writing target-specific code.

5 HipaccVX: Wedding of OpenVX and DSL-based Code Generation

Our DSL-based and library-based approaches presented in Chapter 4 have several advantages over typical high-level libraries such as OpenCV. These include (i) allowing users to describe their algorithms by using an orthogonal set of abstractions instead of restricting them to a predefined set of image processing functions. (ii) introducing inter-kernel optimizations as part of code generation transformations, such as eliminating unnecessary memory copies between image processing functions. (iii) eliminating repetitive work by reducing code replication and increasing modularity. Furthermore, our approach allows generating highly-optimized target-specific code for different computing platforms (e.g., CPUs, GPUs) from the same application description (written in Hipacc). However, our DSL-based approach requires more effort than users writing their application using the OpenCV library interface since the application developer has to learn a non-standard language (e.g., the Hipacc DSL) and understand the abstractions of this DSL (e.g., how to describe image processing using image processing operators).

OpenVX [The14] makes an important contribution to enable system-level optimization possibilities that are not available in traditional libraries such as OpenCV and allows *performance portability* across different computing platforms (e.g., CPU, GPUs). It is the first industrial standard for a graph-based specification of computer vision algorithms. However, OpenVX' algorithm space is constrained to a small set of vision functions. This limitation hinders describing custom kernels for accelerating computations that are not included in the standard. Table 5.1 provides a comparison between OpenVX and Hipacc, which is selected as an example DSL.

In this chapter, we leverage the best of both worlds by coupling the Hipacc DSL to OpenVX and providing language constructs to the programmer for the definition of so-called »user-defined nodes« [ÖOQ+21]. In this way, we support the acceleration of OpenVX' user-defined kernels for various computing platforms and enable optimizations that cannot be detected with standard OpenVX application descriptions. Our approach leverages the OpenVX' industrial standard for the graph-based description of image processing algorithms, allows users to use OpenVX' computer

Table 5.1: Available features in OpenVX (VX), DSL compiler Hipacc (H), and our joint approach HipaccVX (HVX).

Features	VX	H	HVX
Industrial standard (open, royalty-free)	✓	✗	✓
Community driven open-source implementations	✗	✓	✓
Well-known CV functions (e.g., optical flow)	✓	✗	✓
High-level abstractions that adhere to distinct memory access patterns (e.g., local)	✗	✓	✓
Custom node execution on accelerator devices (i.e., OpenCL)	✓	✗	✓
Acceleration of the custom nodes that are based on high-level abstractions	✗	✓	✓

vision functions without learning a non-standard language, and provides the benefits of DSL-based code generation.

5.1 Introduction

The emergence of cheap, low-power cameras and embedded platforms have boosted the use of intelligent systems with CV capabilities in a broad spectrum of markets, ranging from consumer electronics, such as mobile, to real-time automotive applications and industrial automation, e.g., semiconductors, pharmaceuticals, packaging. The global machine vision market size was valued at $16.0 billion already in 2018, and yet, it is expected to reach a value of $24.8 billion by 2023 [BCC18]. A CV application might be implemented on a great variety of hardware architectures ranging from GPUs to FPGAs depending on the domain and the associated constraints (e.g., performance, power, energy, and cost). For sophisticated real-life applications, the best trade-off is often achieved by heterogeneous systems incorporating different computing components that are specialized for particular tasks.

In 2014, the Khronos Group released OpenVX as a C-based API to facilitate cross-platform portability not only of the code but also of the performance for CV applications [The14]. This is momentous since OpenVX is the first (royalty-free) standard for a graph-based specification of CV algorithms, enabling system-level optimizations such as eliminating unnecessary memory copies. Yet, OpenVX' algorithm space is constrained to a relatively small set of vision functions. Users are allowed to instantiate additional code in the form of custom nodes, but these cannot be analyzed at the system level by the graph-based optimizations applied from an OpenVX back end. Furthermore, writing a custom node requires users to optimize their code for a specific platform. Standard programming languages such as OpenCL should not be used for writing custom nodes since they do not offer performance portability across different computing platforms [SFL+15; DWL+12]. Table 5.1 summarizes deficiencies in OpenVX standard.

A solution to the problems mentioned above is offered by the community working

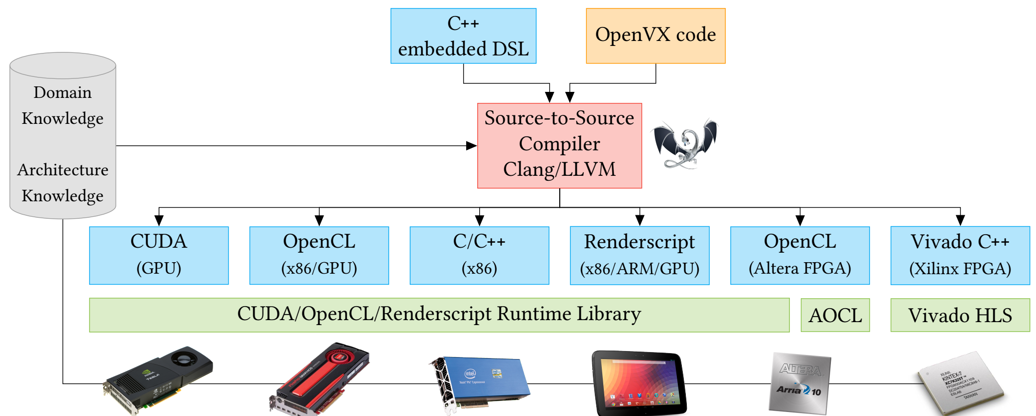


Figure 5.1: HipaccVX overview. (Figure reprinted from [ÖOQ+21], © CC BY)

on DSLs for image processing. These DSLs are able to generate code from a set of *algorithmic abstractions* that lead to high-performance execution for diverse types of computing platforms [RBA+13; MRH+16; MVB15]. However, existing DSLs lack formal verification, hence they do not ensure the safe execution of a user application whereas OpenVX is an industrial standard.

In this thesis, we couple the advantages of DSL-based code generation with OpenVX (summarized in Table 5.1). We present a set of abstractions that are used as basic building blocks for expressing OpenVX' standard CV functions. These building blocks are suitable for generating optimized, device-specific code from the same functional description. As a result, we achieve performance portability not only for OpenVX' CV functions but also for user-defined kernels[1] that are expressed with these computational abstractions.

In summary, the contributions of this chapter are as follows[2]:

- We systematically categorize and specify OpenVX' CV functions by high-level abstractions that adhere to distinct memory access patterns (see Section 5.4.1).
- We propose a framework called HipaccVX, which is an OpenVX implementation that achieves high performance for a wide variety of target platforms, namely, GPUs, CPUs, and FPGAs (see Section 5.5). It uses hipacc's backend for code generation, as shown in Figure 5.1.
- HipaccVX supports the definition of custom nodes (i.e., user-defined kernels) based on the proposed abstractions (see Section 5.5.2).
- To the best of our knowledge, our approach is the first one that allows for graph-based optimizations that incorporate not only standard OpenVX CV nodes but also

[1]A kernel in OpenVX is the abstract representation of a computer vision function [The19].
[2]The contents of this chapter are based on and partly published in [ÖOQ+21], which has appeared in Journal of Real-Time Image Processing.

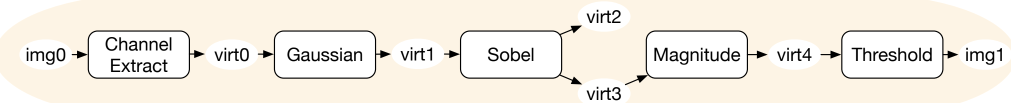


Figure 5.2: The graph representation for the OpenVX code shown in Listing 5.1. The output image (*img1*) contains solely the horizontal edges extracted from the input image (*img0*). The *virt2* image is defined only because OpenVX' Sobel function returns both horizontal and vertical edges. This redundant computation is eliminated during the optimization passes of our HipaccVX compiler framework (see Section 5.5.3). (Figure reprinted from [ÖOQ+21], © CC BY)

user-defined custom nodes (see Section 5.5.3), i.e., optimizations across standard and custom nodes.

5.2 OpenVX programming model

OpenVX is an open, royalty-free C-based standard for the cross-platform acceleration of computer vision applications. The specification does not mandate any optimizations or requirements on device execution; instead, it concentrates on software abstractions that are freed from low-level platform-specific declarations. The OpenVX API is opaque; that is, the memory hierarchy and device synchronization are hidden from the user. Typically, platform experts of the individual hardware vendors provide optimized implementations of the OpenVX API [The18a].

Listing 5.1 shows an example OpenVX code for a simple edge detection algorithm, for which the application graph is shown in Figure 5.2. An application is described as a DAG, where nodes represent CV functions (see Lines 14 to 18) and data objects, i.e., images, scalars (see Lines 4 to 12), while edges show the dependencies between nodes. All OpenVX *objects* (i.e., *graph*, *node*, *image*) exist within a *context* (Line 1). A *context* keeps track of the allocated memory resources and promotes implicit freeing mechanisms at `release` calls (Line 24). A *graph* (Line 2) solely operates on the data objects attached to the same context.

The data objects used only for the intermediate steps of a calculation should be specified as *virtual* by the users. These are considered as graph edges between CV function nodes and are inaccessible for the rest of the application. For instance, *virtual images* defined in Lines 9 to 12 are declared with null image sizes and undefined data types. They cannot be accessed via read/write operations. This paves the way for system-level optimizations applied in a platform-specific back end, i.e., host-device data transfers or memory allocations [RVD+14].

Listing 5.1: OpenVX code for an edge detection algorithm. The application graph derived for this OpenVX program is shown in Figure 5.2.

```
1  vx_context context = vxCreateContext();
2  vx_graph graph = vxCreateGraph(context);
3
4  vx_image img[] = {
5    vxCreateImage(context, width, height, VX_DF_IMAGE_UYVY),
6    vxCreateImage(context, width, height, VX_DF_IMAGE_U8)};
7
8  vx_image virt[] = {
9    vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
10   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
11   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT),
12   vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_VIRT)};
13
14 vxChannelExtractNode(graph, img[0], VX_CHANNEL_Y, virt[0]);
15 vxGaussian3x3Node(graph, virt[0], virt[1]);
16 vxSobel3x3Node(graph, virt[1], virt[2], virt[3]);
17 vxMagnitudeNode(graph, virt[3], virt[3], virt[4]);
18 vxThresholdNode(graph, virt[4], thresh, img[1]);
19
20 status = vxVerifyGraph(graph);
21 if (status == VX_SUCCESS)
22   status = vxProcessGraph(graph);
23
24 vxReleaseContext(&context);
```

An OpenVX graph must be *verified* (Line 20) before it is *executed* (Line 22). Therefore, the execution is not eager. The verification ensures the safe execution of a graph description and resolves the implementation types of virtual data objects. The OpenVX standard mandates that a verification procedure must, at the minimum, (*i*) validate the node parameters (i.e., presence, directions, data types, range checks), and (*ii*) assure the graph connectivity (detection of cycles), [The18b].

An OpenVX backend performs optimizations during the verification phase. Its graph-based description allows for addressing system-level issues such as accelerator communications, memory allocations, and data transfers. Correspondingly, an application graph might be restructured before the execution [RVD+14]. Therefore, verification could be slow, but it is considered to be an initialization procedure. Ideally, a verified OpenVX graph is executed repeatedly for different input parameters (i.e., a new frame in video processing).

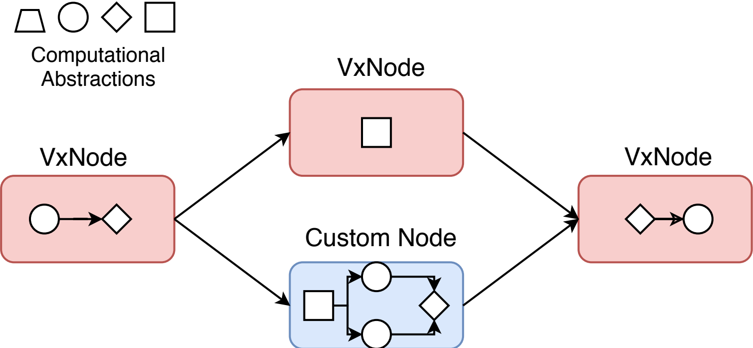


Figure 5.3: HipaccVX enables performance portability for user-defined code by representing OpenVX' CV functions and custom nodes by a small set of computational abstractions. (Figure reprinted from [ÖOQ+21], © CC BY)

5.3 Deficiencies of OpenVX

In OpenVX, the smallest component to express a computation is a graph node (e.g., `vxGaussian3x3Node`) from its base CV functions. However, these CV functions are restricted to a small set since OpenVX has a tight focus on cross-platform acceleration [The19]. Custom nodes can be added to extend this functionality[3], but they leave the following issues unresolved: (*i*) Users will be responsible for the performance of a custom node, who supposedly should not consider performance optimizations. (*ii*) Performance of a custom node will be tuned for a specific platform (*iii*) The graph optimization routines will not be able to analyze custom nodes.

For instance, consider Figure 5.3 that depicts an OpenVX application graph with three CV function nodes (red) and a user-defined kernel node (blue). A GPU back end would offer optimized implementations of the vxNodes (e.g., Gauss), but the user code (custom node) is a black box for the graph optimizations.

Programming models such as OpenCL can be used to implement custom nodes. This enables functional portability across a great variety of computing platforms. However, the user should have *expertise* in the target architecture to optimize an implementation for high performance. Furthermore, OpenCL cannot assure the portability of the performance since the code needs to be tuned according to the target device, i.e., usage of device-specific synchronization primitives, exploitation of texture memory if available, usage of vector operations, or different numbers of hardware threads [SFL+15; DWL+12]. In fact, an OpenCL code optimized for an ISA has to be ultimately rewritten for an FPGA implementation in order to deliver

[3]The support for the execution of a user code (custom node) as part of an application graph on an accelerator device was introduced in August 2019 with the release of OpenVX v1.3 [The19]. Previous versions [The18b] constraint the usage of the user-defined kernels to the host platform and required them to be implemented as C++ kernels.

high-performance [ÖRH+16].

5.4 Our Approach: DSL-based Code Generation for OpenVX

As a solution to the challenges posed in Section 5.3, we propose introducing an orthogonal set of so-called *computational abstractions* that enables high-performance implementations for a variety of computing platforms (such as CPUs, GPUs, FPGAs), similar to Hipacc (see Chapter 4). We suggest using these abstractions to implement OpenVX' CV functions and, at the same time, to serve users for the description of custom nodes. First, we explain these computational abstractions in Section 5.4.1, and then, summarize the further advantages of our approach in Section 5.4.2.

5.4.1 Computational Abstractions

We have analyzed OpenVX' CV functions and categorized them into the computational abstractions summarized in Table 5.2. The categorization is mainly based on three groups of operators:

(*i*) *point operators* that compute an output from one input pixel,

(*ii*) *local operators* depend on neighbor pixels over a certain region, and

(*iii*) *global operators* where the output might depend on the whole input image (presented in Figure 5.4).

We have identified the following patterns for the global operators:

(*a*) *reduction*: traverses an input image to compute one output (e.g., max, mean),

(*b*) *histogram*: categorizes (maps) input pixels to bins according to a binning (reduce) function,

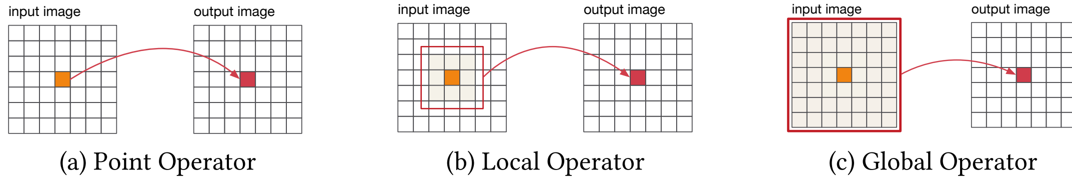
(*c*) *scaling*: downsizes or expands input images by interpolation,

(*d*) *scan*: each output pixel depends on the previous output pixel.

(*e*) *Warp*, *transpose*, and *matrix multiplication* are denoted as global operator blocks.

Table 5.2: Categorization of the OpenVX Kernels according to data access patterns

OpenVX Kernels	HipaccVX Abstractions	Hipacc Abstractions
AbsDiff, Copy, Add, Subtract, And, Xor, Or, Not, ChannelCombine, ChannelExtract, ColorConvert, ConvertDepth, Magnitude, Phase, Multiply, ScaleImage, Threshold, TensorAdd, TensorSubtract, TensorConvertDepth, TensorMultiply, ScalarOperation, Select, Remap	point	Kernel
NonMaxSuppression, Dilate3x3, Erode3x3, NonLinearFilter, Median3x3, BilateralFilter, Sobel3x3, Box3x3, Convolve, Gaussian3x3, LBP, FastCorners	local	Kernel
MinMaxLoc, MeanStdDev, Min, Max	reduce (global)	Reduction
Histogram	histogram (global)	Histogram
scale-image	scale (global)	Interpolation
GaussianPyramid, LaplacianPyramid, LaplacianReconstruct	pyramid (global)	Pyramid
IntegralImage	scan (global)	Software
WarpAffine, WarpPerspective	warp (global)	Software
TensorTranspose, TensorMatrixMultiply	(global) transpose, matrixMult	Software
HarrisCorners	point + local + *custom*	Kernel, Software
EqualizeHist	histogram + point	Kernel, Histogram
OpticalFlowPyrLK	point + local + pyramid + *custom*	Kernel, Pyramid, Software
HOGCells	*custom* + local + histogram	Kernel, Software
CannyEdge	point + local + *custom*	Kernel, Software



(a) Point Operator (b) Local Operator (c) Global Operator

Figure 5.4: The considered computational abstractions (listed in Table 5.2) are based on three groups of operators. (Figure reprinted from [ÖOQ+21], © CC BY)

5.4.2 Advantages of using Computational Abstractions

Using the proposed set of abstractions reduces code duplication compared to typical approaches, where the libraries are implemented using hand-written CV functions. For instance, 36 out of OpenVX' CV functions can be implemented solely with the description of point and local operators as shown in Table 5.2; that is, a few highly optimized building blocks for a target platform (e.g., GPU) can be reused.

Implementing both the OpenVX CV functions and the custom nodes using the same language constructs allows constructing a graph using same computational patterns. Figure 5.3 illustrates this benefit. Remember that red and blue nodes in Figure 5.3 represent OpenVX' CV functions and custom nodes, respectively. The geometric shapes represent the computational abstractions explained in Section 5.4.1. It can be seen that the implementation of an application solely contains the same type of building blocks, where user code is not left as a "black box".

Our approach sacrifices flexibility by restricting users to a given set of DSL abstractions for defining custom nodes. In return, it is able to accelerate user code for different computing platforms. Hereby, users do not need to understand potential optimizations for a target platform to achieve high performance.

Memory access patterns of our abstractions entail system-level optimization strategies motivated by the OpenVX standard, such as image tiling [THM+18] and hardware-software partitioning [THB+18]. Furthermore, by introducing the node-internal computational abstractions, our approach enables additional optimizations that manipulate the computation (see Sections 5.5.2 and 5.5.3). This is illustrated in Figure 6.1. Assume that all the CV functions of the OpenVX code in Listing 5.1 are implemented by using the *point* and *local* operators. Then, its application graph (Figure 5.2) is transformed into the implementation graph shown in Figure 6.1. This allows for analyzing the described application at a finer level. For instance, the vertical derivative function in `vxSobel3x3` does not contribute to the resulting image; hence the redundant computation is eliminated. This would not be possible in a setting the whole computation is offloaded to a device from a typical library implementation. Thanks to code generation abilities, the locality is increased by fusing the CV nodes.

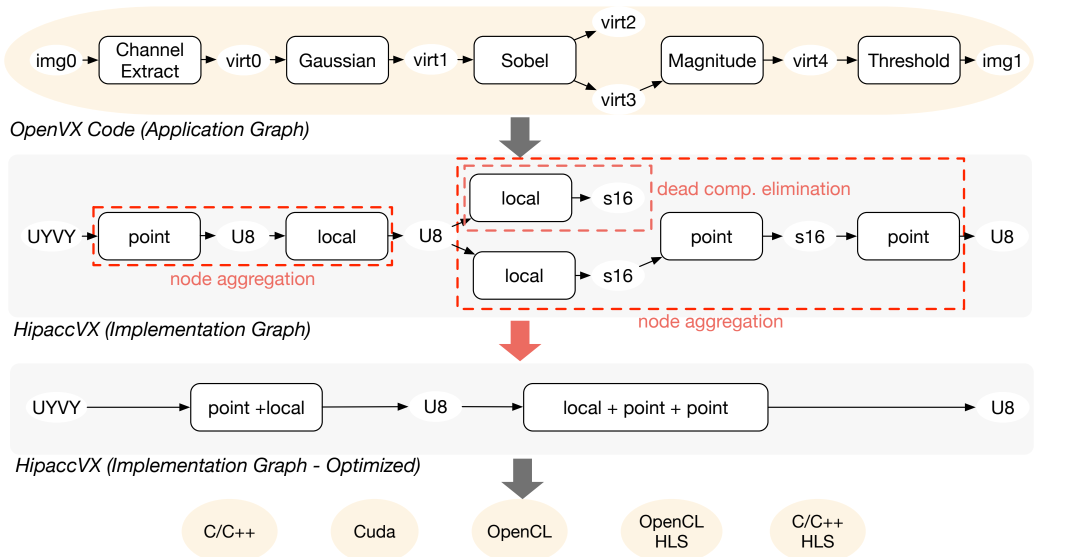


Figure 5.5: Given an application graph consisting of five consecutive CV functions. A typical OpenVX implementation will have a specific implementation for each CV function despite these algorithms having similar patterns. In our approach, we describe every CV function using high-level abstractions called point and local. Therefore, the implementation graph only contains two types of operators. This allows generating target-specific code using a DSL for various devices, including CPUs, GPUs, and FPGAs. Furthermore, our approach enables additional optimizations such as dead computation elimination and node aggregation (see Sections 5.5.2 and 5.5.3). (Figure reprinted from [ÖOQ+21], © CC BY)

5.5 Implementation: The HipaccVX Framework

In this thesis, we developed a framework called HipaccVX, a DSL-based implementation of OpenVX [ÖOQ+21]. We extended OpenVX specification by Hipacc code interoperability (see Section 5.5.2) such that programmers are allowed to register Hipacc kernels as custom nodes to OpenVX programs. The HipaccVX framework consists of an OpenVX graph implementation and optimization routines that verify and optimize input OpenVX applications (see Section 5.5.3). Ultimately, it generates a device-specific code for the target platform using hipacc's code generation. The tool flow is presented in Figure 5.1.

5.5.1 Image Processing DSLs

Recently proposed DSL compilers for image processing, such as Halide [RBA+13], Hipacc [MRH+16], and PolyMage [MVB15], enable the portability of high-performance

across varying computing platforms. All of them take as input a high-level, functional description of the algorithm and generate platform-specific code tuned for the target device. In this work, we use Hipacc to present our approach. An overview of Hipacc DSL is given below (see Section 4.2 for more details).

Hipacc provides language constructs that are embedded into C++ for the description of computations on image objects. Applications are defined in a single program, multiple data (SPMD) context, similar to kernels in CUDA and OpenCL. For instance, Listing 5.2 shows the description of a discrete Gaussian blur filter application. First, a `Mask` is defined in Line 7 from a constant array. Then, input and output `Image`s are defined as C++ objects in Lines 12 and 13, respectively. *Clamping* is selected as the image boundary handling mode for the input image in Line 16. The whole input and output images are defined as ROI by the `Accessor` and `IterationSpace` objects that are specified in Lines 17 and 20, respectively. Finally, the Gaussian kernel is instantiated in Line 23 and executed in Line 24.

Listing 5.3 describes the actual *operator kernel* for the Gaussian shown in Listing 5.2. The `LinearFilter` is a user-defined class that is derived from Hipacc's `Kernel` class, where the *virtual* `kernel` method is overridden (according to C++ polymorphism rules). There, a user describes a convolution as a lambda function using the `convolve()` construct, which computes an output pixel (`output()`) from an input window (`input(mask)`). Hipacc's compiler utilizes Clang's AST to specialize the lambda function according to the selected platform and generates device-specific code that provides high-performance implementations when compiled with the target architecture compiler. We refer to Section 4.2 and [MRH+16] for more detailed explanations and other programming language constructs of Hipacc.

5.5.2 DSL Back End and User-Defined Kernels

OpenVX mandates the verification of parameters and the relationship between input and output and parameters as presented in Listing 5.4. There, first, a user kernel and all of its parameters should be defined (lines 6 to 22). Then, a custom node should be created by `vxCreateGenericNode` (Line 27) after the user kernel is finalized by a `vxFinalizeKernel` call (Line 24). The kernel parameter types are defined, and the node parameters are set by `vxAddParameterToKernel` (lines 16 to 22) and `vxSetParameterByIndex` (lines 28 to 30), respectively.

We extended OpenVX by a `vxHipaccKernel` function (Line 6) to instantiate a Hipacc kernel as an OpenVX kernel. The Hipacc kernels should be written in a separate file and added as a generic node according to the OpenVX standard [The19]. Programmers do not have to describe the dependency between Hipacc kernels as in Listing 5.2. Instead, they write a regular OpenVX program to describe an application graph. This sustains the custom node definition procedure of OpenVX. Ultimately, during the `vxVerifyGraph` call, the HipaccVX framework checks the correctness

Listing 5.2: Hipacc application code for a Gaussian filter. The instantiated `LinearFilter` Kernel is given in Listing 5.3.

```
// filter mask for Gaussian blur filter
const float filter_mask[3][3] = {
  { 0.057118f, 0.124758f, 0.057118f },
  { 0.124758f, 0.272496f, 0.124758f },
  { 0.057118f, 0.124758f, 0.057118f }
};
Mask<float> mask(filter_mask);

// input and output images
size_t width, height;
uchar *image = read_image(&width, &height, "input.pgm");
Image<uchar> in(width, height, image);
Image<uchar> out(width, height);

// reading from in with clamping as boundary condition
BoundaryCondition<uchar> cond(in, mask, Boundary::CLAMP);
Accessor<uchar> acc(cond);

// output image (region of interest is the whole image)
IterationSpace<uchar> iter(out);

// instantiate and launch the Gaussian blur filter
LinearFilter Gaussian(iter, acc, mask, 3);
Gaussian.execute();
```

Listing 5.3: Hipacc kernel code for an FIR filter.

```
class LinearFilter: public Kernel <uchar > {
  // ...
  public:
    LinearFilter(Accessor<uchar> &input, // input image
                 IterationSpace<uchar> &out, // output image
                 Mask<float> &mask)          // mask
                 : {/* ... */}

  void kernel() { // convolve -> local operator
    output() = convolve(mask, Reduce::SUM, [&] () -> uchar {
          return mask() * input(mask);
    });
  }
};
```

of the application graph (i.e., verifies that the graph does not have a cycle, node parameters have the correct type, and none of the parameters have `NULL` value), optimizes the given OpenVX application, generates the corresponding Hipacc code,

Listing 5.4: DSL code interoperability extension (only Line 6).

```
vx_node vxGaussian3x3Node(vx_graph graph,
                          vx_image arr,
                          vx_image out) {

  // Extension: An OpenVX kernel from a Hipacc kernel
  vx_kernel cstmk = vxHipaccKernel("gaussian3x3.cpp");

  /*** The code below is the standard OpenVX API ***/
  // Create vx_matrix for mask
  const float coeffs[3][3] = /* ... */;
  vx_matrix mask = vxCreateMatrix(context, VX_TYPE_FLOAT32, 3, 3);
  vxCopyMatrix(mask, (void*)coeffs, VX_WRITE_ONLY, VX_MEMORY_TYPE_HOST);

  // Set input/output parameters for a kernel
  vxAddParameterToKernel(cstmk, 0, VX_OUTPUT, VX_TYPE_IMAGE,
                         VX_PARAMETER_STATE_REQUIRED);

  vxAddParameterToKernel(cstmk, 1, VX_INPUT, VX_TYPE_IMAGE,
                         VX_PARAMETER_STATE_REQUIRED);

  vxAddParameterToKernel(cstmk, 2, VX_INPUT, VX_TYPE_MATRIX,
                         VX_PARAMETER_STATE_REQUIRED);

  vxFinalizeKernel(cstmk);

  // Create generic node
  vx_node node = vxCreateGenericNode(graph, cstm_k);
  vxSetParameterByIndex(node, 0, (vx_reference) out);
  vxSetParameterByIndex(node, 1, (vx_reference) arr);
  vxSetParameterByIndex(node, 2, (vx_reference) mask);

  return node;
}
```

and employs Hipacc framework for device-specific code generation.

OpenVX' CV functions are implemented as a library by using our extension for Hipacc code instantiation. For instance, the HipaccVX implementation of the `vxGaussian3x3Node` API is shown in Listing 5.4. Users can simply use these CV functions as in Listing 5.1. A minority of OpenVX functions are implemented as OpenCV kernels since they cannot be fully described in Hipacc. These are listed in Table 5.2 with a *Software* label instead of a Hipacc abstraction type. As future work, we can extend Hipacc to support these functions.

Optimizations Based on Code Generation

By implementing a Hipacc back end for OpenVX, many device-specific optimization techniques are inherited from Hipacc. Hipacc internally applies several optimizations for the code generation from its DSL abstractions. These include memory padding, constant propagation, utilization of textures, loop unrolling, kernel fusion, thread-coarsening, implicit use of unified CPU/GPU memory, and the integration with CUDA Graph [MRH+16; RKH+17; QRH+19; QÖT+20a]. At the same time, Hipacc targets Intel and Xilinx FPGAs using their HLS tools. There, an input application is implemented through application circuits derived from the DSL abstractions and optimized by hardware techniques such as pipelining and loop coarsening [RÖM+17b; ÖRH+16; ÖRH+17b].

5.5.3 OpenVX Graph and System-Level Optimizations

As mentioned before, an OpenVX application is represented by a DAG $G_{app} = (V, E)$, where V is a set of vertices, and E is a set of edges $E \subseteq V \times V$ denoting data dependencies between nodes. The set of vertices V can further be divided into two disjoint sets D and N ($V = D \cup N$, $D \cap N = \emptyset$) denoting data objects and CV functions, respectively.

Both data (i.e., `Image`, `Scalar`, `Array`) and node (i.e., CV functions) objects are implemented as C++ classes that inherit the OpenVX `Object` class. Vertices $v \in V$ of our OpenVX graph implementation consist of OpenVX `Object` pointers. The verification phase first checks if an application graph G_{app} (derived from the user code, see, e.g., Listing 5.1) does not contain any cycles. Then, it verifies that the description is a bipartite graph, i.e., $\forall(v, w) \in E: \ v \in D \wedge w \in N \vee v \in N \wedge w \in D$. Finally, the verification phase applies the following optimizations:

Reduction of Data Transfers

Data nodes of an application graph that are not virtual must be accessible to the host. In contrast, the intermediate (virtual) points of a computation should be stored in the device memory. We distinguish these two data node types by the set of non-virtual data nodes D_{nv} and the set of virtual data nodes D_v, where $D = D_{nv} \cup D_v$, $D_{nv} \cap D_v = \emptyset$. HipaccVX keeps this information in its graph implementation and determines the subgraphs between non-virtual data nodes, which can be kept in the device memory. In this way, data transfers between the host and device are avoided.

Elimination of Dead Computations

An application graph may consist of nodes that do not affect the results. Inefficient user code or other compiler transformations might cause such dead code. A less

Algorithm 3: Graph Analysis for Dead Computation Elimination

input : G_{app} – application graph
D_{nv} – set of are non-virtual data nodes
output: G_{filt} – optimized application graph

```
1  function eliminate_death_nodes(G_app, D_nv)
       /* Find candidate non-virtual roots and leaves */
2      D_in ← ∅, D_out ← ∅
3      forall v ∈ D_nv do
4          if deg⁻ (v) = 0 then
5              D_in ← D_in ∪ v                 // input non-virtual data nodes
6          end
7          else if deg⁺ (v) = 0 then
8              D_out ← D_out ∪ v               // out non-virtual data nodes
9          end
10         else
11             D_in ← D_in ∪ v
12             D_out ← D_out ∪ v
13         end
14     end
       /* Mark the nodes between roots and leaves as alive */
15     G_trans ← transpose_graph (G_app)
16     V_alive ← ∅
17     forall v_start ∈ D_out do
18         V_v ← depth-first_visit (v_start, D_in, G_trans)
19         V_alive ← V_alive ∪ V_v
20     end
       /* Filter, keep only the alive nodes and their edges */
21     G_filt ← filter_graph (G_app, KEEP_EDGES, V_alive)
22     return G_filt
23 end
```

apparent reason could be the usage of OpenVX compound CV functions for smaller tasks. Consider `Sobel3x3` as an example, which computes two images, one for the horizontal and one for the vertical derivative of a given image. As the OpenVX API does not offer these algorithms separately, programmers have to call `Sobel3x3`, even when they are only interested in one of the two resulting images. Our implementation is based on abstractions and allows a better analysis of the computation compared to OpenVX' CV functions, i.e., the Sobel API is implemented by two parallel local operators, as shown in Figure 6.1. HipaccVX optimizes a given application graph using the procedure described in Algorithm 3. Conventional compilers do not analyze

this redundancy if utilizing the host/device execution paradigm (e.g., OpenCL, CUDA), where OpenVX kernels are offloaded to an accelerator device, and the host executes device kernels according to the application dependency (see Section 5.6.2).

Algorithm 3 assumes that the non-virtual data nodes whose input and output degrees are zero must be the inputs (D_{in}) and the results (D_{out}) of an application, respectively. Other non-virtual data nodes could be input, output, or intermediate points of an application, depending on the number of connected virtual data nodes. These are initialized in Line 2. Then, all of the nodes in the same component between the node v_{start} and the set V_{in} are traversed via the *depth-first visit* function (Line 18) and marked as alive (Lines 2 to 20). Finally, in Line 21, a filtered view of an application graph is created from the set of alive nodes.

The complexity of the functions transpose (Line 15) and depth-first visit (Line 18) are $O(|V| + |E|)$ and $O(|E|)$, correspondingly. The filter graph function (Line 21) is only an adaptor that requires no change in the application graph [SLL02]. In the worst case, the graph has $|V| - 2$ output data nodes. That is, the complexity of Algorithm 3 becomes $O(|V|^2 + |E|)$ in time and $O(|V| + |E|)$ in space.

5.6 Evaluation and Results

We present results for a Xilinx Zynq ZYNQ-zc706 FPGA using Xilinx Vivado HLS 2019.1 and an Nvidia GeForce GTX 680 with CUDA driver 10.0. We evaluate the following applications:

- As image smoothers, we consider a Gaussian blur (*Gauss*) and a *Laplacian* filter with a 5×5 and 3×3 local node, respectively.
- The filter chain (*FChain*) is an image pre-processing algorithm consisting of three convolution (local) nodes.
- The *SobelX* determines the horizontal derivative of an input image using the OpenVX `vxSobel` function.
- The edge detector in Figure 5.2 (*EdgFig5.2*) finds horizontal edges in an input image, while *Sobel* computes both horizontal and vertical edges using three CV nodes.
- The *Unsharp* filter sharpens the edges of an input image using one Gauss node and three point operator nodes.
- Both *Harris* and *Tomasi* detect corners of a given image using 13 (4 local + 9 point) and 14 (4 local + 10 point) CV nodes, respectively.

These applications are representative to show the optimization techniques discussed in this chapter. The performance of a simple CV application (e.g., *Gauss*) solely depends on the quality of code generation, while graph-based optimizations can further optimize the performance of more complex applications (e.g., *Tomasi*). *Laplacian* uses the OpenVX' custom convolution API and *EdgFig2* consists of redundant kernels.

5.6.1 Acceleration of User-Defined Nodes

User-defined nodes can be accelerated on a target platform (e.g., GPU accelerator) when they are expressed using HipaccVX' abstractions (see Section 5.5.2). A standard C++ implementation of these custom nodes results in executing them on the host device. This is illustrated in Figure 5.6 for a corner detection algorithm that consists of nine kernels. The CPU codes for these custom nodes are also acquired using Hipacc. As seen in Figure 5.6, HipaccVX provides the same performance invariant to the number of user-defined nodes, whereas using the OpenVX API leads to a severe decrease in throughput since each user-defined node has to be executed on the host CPU.

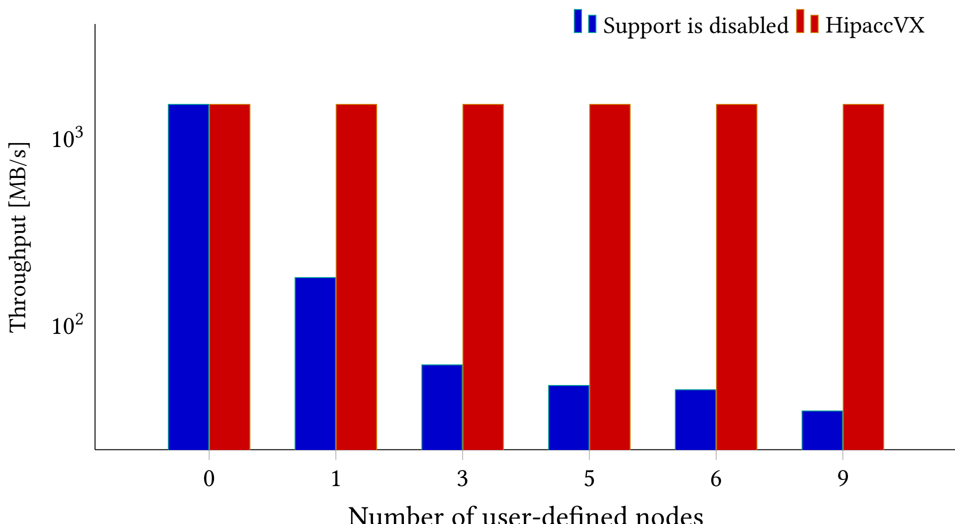


Figure 5.6: Throughput for different versions of the same corner detection application (consisting of 9 kernels) on the Nvidia GTX680 (higher is better). The blue bars denote an increasing number of CV functions implemented as user-defined nodes using C++. In OpenVX, these user-defined functions have to be executed on the host CPU, which leads to performance degradation; whereas HipaccVX accelerates all user-defined nodes on the GPU. (Figure reprinted from [ÖOQ+21], © CC BY)

5.6.2 System-Level Optimizations based on OpenVX Graph

Reduction of Data Transfers HipaccVX eliminates the data transfers between the execution of subsequent functions on a target accelerator device, as explained in Section 5.5.3. This is disabled for *naive* implementations. The improvements for the two applications are shown in Figure 5.7. HipaccVX' throughput optimizations reach a speedup of 13.5.

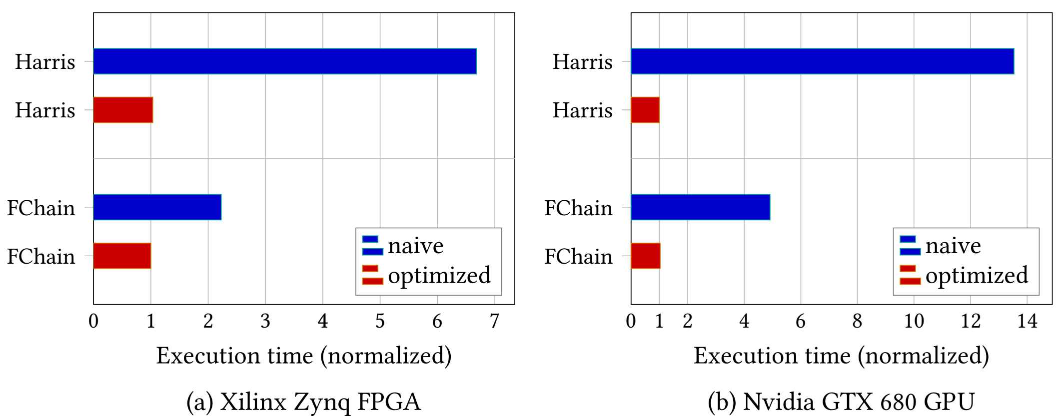


Figure 5.7: Normalized execution time (lower is better) for 1024 × 1024 images. HipaccVX eliminates redundant transfers by analyzing OpenVX' graph-based application code. (Figure reprinted from [ÖOQ+21], © CC BY)

Elimination of Dead Computation HipaccVX eliminates the computations that do not affect the results of an application (see Section 5.5.3). This is illustrated in Figure 5.8. HipaccVX improves the throughput by a factor of 2.1 on the GTX 680. The throughput for the Zynq FPGA is only slightly improved since the applications fit into the target device, thus, run in parallel. Yet, HipaccVX' FPGA implementation for the same application reduces the number of FPGA resources (elementary programmable logic blocks called *slices* and on-chip block RAMs, short *BRAMs*) significantly (around 50% for SobelX) on the Zynq (see Figure 5.9).

5.6.3 Evaluation of the Performance

In Figure 5.10, we compare HipaccVX with the VisionWorks (v1.6) [NVI20] provided by Nvidia, which provides an optimized commercial implementation of OpenVX. HipaccVX, as well as typical library implementations, exploit the graph-based OpenVX API to apply system-level optimizations [RVD+14], such as reduction of data transfers (see Section 5.5.3). Additionally, HipaccVX generates code that is specific to target GPU architectures and applies optimizations such as constant propagation,

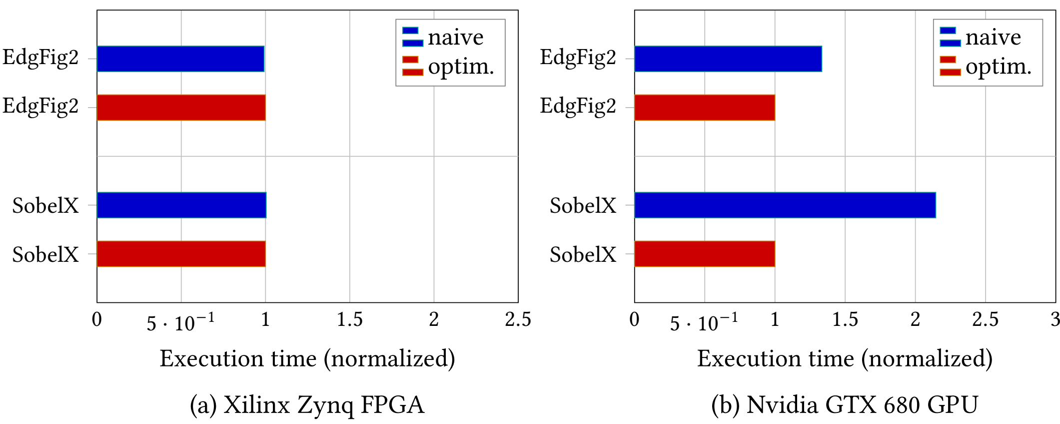


Figure 5.8: Normalized execution time (lower is better) for 1024 × 1024 images. (Figure reprinted from [ÖOQ+21], © CC BY)

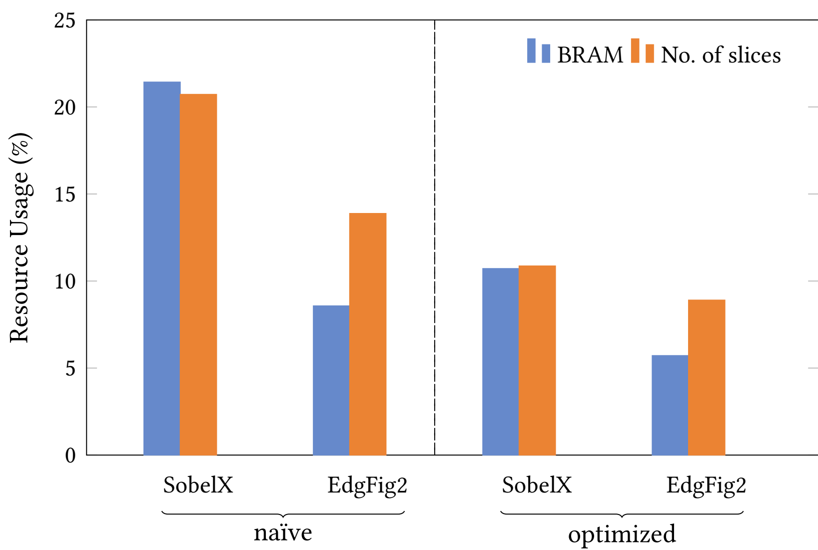


Figure 5.9: The post place and route (PPnR) results for the Xilinx Zynq FPGA. Elimination of dead computation significantly reduces the percentage of resources used. (Figure reprinted from [ÖOQ+21], © CC BY)

thread coarsening, multiple program, multiple data (MPMD) [MRH+16]. As shown in Figure 5.10, HipaccVX can generate implementations that provide higher throughput than VisionWorks. Here, the speedups for applications that are composed of multiple kernels (Harris, Tomasi, Sobel, Unsharp) are higher than the ones solely consisting of one OpenVX CV function (Gauss and Laplacian). This performance boost is, to a large extent, due to the locality optimization achieved by fusing consecutive kernels at the

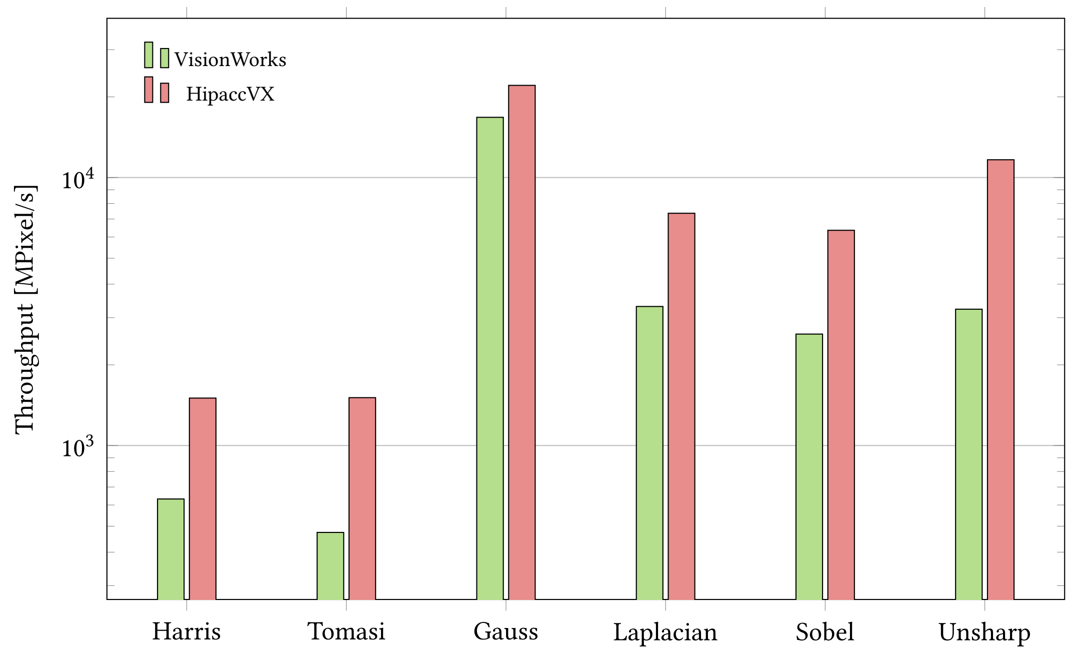


Figure 5.10: Comparison of Nvidia VisionWorks v1.6 and HipaccVX on the Nvidia GTX 680. Image sizes are 2048×2048. (Figure reprinted from [ÖOQ+21], © CC BY)

Table 5.3: PPnR results for the Xilinx Zynq for images of 1020 × 1020 and T_{target} = 5 ns (corresponds to f_{target} = 200 MHz).

App	v		BRAM	SLICE	DSP	Latency [cyc.]
Gauss	1	HipaccVX	8	473	16	1044500
		Halide-HLS	8	1823	50	1052673
	4	HipaccVX	16	1519	64	261649
		Halide-HLS	16	4112	180	266241
Harris	1	HipaccVX	20	1457	34	1042466
		Halide-HLS	16	2688	35	1052673
	2	HipaccVX	20	2326	68	521756
		Halide-HLS	16	4011	70	528385

compiler level [QRH+19]. This requires code rewriting and the resource analysis of the target GPU architectures.

There was no publicly available FPGA implementation of OpenVX at the time

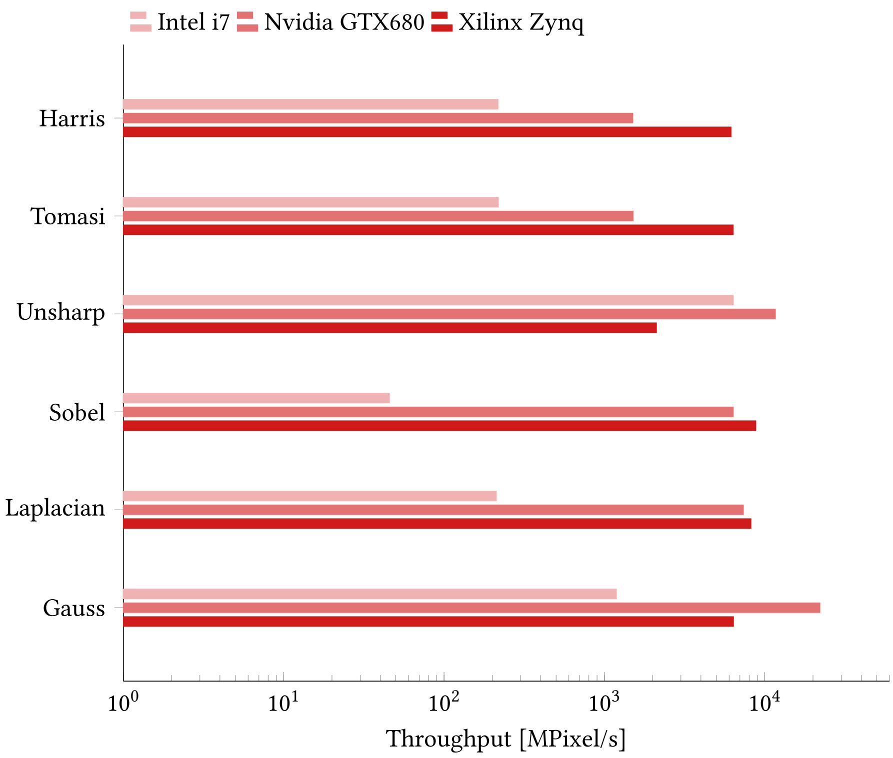


Figure 5.11: Comparison of throughput for the Nvidia GTX680, Xilinx Zynq, and Intel i7-4790 CPU. The same OpenVX application code is used to generate different accelerator implementations. The HipaccVX framework allows for both code and performance portability by generating optimized implementations for various accelerators. (Figure reprinted from [ÖOQ+21], © CC BY)

this thesis was written. Therefore, in Table 6.2, we compare HipaccVX with Halide-HLS [PBY+17], which is a state-of-the-art DSL targeting Xilinx FPGAs. As can be seen, HipaccVX uses fewer resources and achieves a higher throughput for the benchmark applications. HipaccVX transforms a given OpenVX application into a streaming pipeline by replacing virtual images with FIFO semantics. Thereby, it uses an internal representation in SSA form. Furthermore, it replicates the innermost kernel to achieve higher parallelism for a given factor v. For practical purposes, we present results only for Xilinx technology. Chapter 4 shows that Hipacc can achieve a performance similar to handwritten examples provided by Intel for image processing. This also indicates that the memory abstractions given in Table 5.2 are suitable to generate optimized code for HLS tools.

Figure 5.11 compares the throughputs that were achieved from the same OpenVX application code for different accelerators. Here, we generated OpenCL, CUDA,

and Vivado HLS (C++) code to implement a given application on an Intel i7-4790 CPU, an Nvidia GTX680 GPU, and a Xilinx Zynq FPGA, respectively. GPUs and FPGAs can exploit data-level parallelism by processing a significantly higher number of operations in parallel compared to CPUs. This makes them very suitable for computer vision applications. Modern GPUs operate on a higher clock frequency compared to existing FPGAs. Therefore they could provide higher throughput for abundantly parallel applications. This is the case for Gauss and Unsharp. However, FPGAs can exploit temporal locality by using *pipelining* and eliminate unnecessary data transfers to global memory between consecutive kernels. Therefore, all the FPGA implementations in Figure 5.11 achieve a similar throughput.

5.7 Related Work

The OpenVX specification is not constrained to a particular memory model as OpenCL and OpenMP. Therefore, it provides a better performance portability than traditional libraries such as OpenCV [RVD+14]. It has been implemented by a few major vendors, including Nvidia, Intel, AMD, and Synopsys [The18a]. The authors of [EYA15; YAY+18; MWS+16; ZTS18; THM+18] focus on graph scheduling and design space exploration for heterogeneous systems consisting of GPUs, CPUs, and custom instruction-set architectures. Unlike the prior work, [THM+16] suggests static OpenVX compilation for low-power embedded systems instead of runtime library implementations. Our work is similar to this since we statically analyze a given OpenVX application and combine the benefits of domain-specific code generation approaches [RBA+13; MRH+16; MVB15; RÖM+17b; PBY+17; CVP+16].

Halide [RBA+13], Hipacc [MRH+16], and PolyMage [MVB15] are image processing DSLs that provide language constructs and scheduling primitives to generate optimized code for the target device, i.e., CPUs, GPUs. Halide [RBA+13] decouples the algorithm description from scheduling primitives, i.e., vectorization, tiling, while Hipacc [MRH+16] and PolyMage [MVB15] implicitly apply these optimizations on a graph-based description similar to OpenVX. CAPH [Sér+13], RIPL [SMB+16], and Rigel [HDD+16] are image processing DSLs that generate optimized code for FPGAs. Hipacc-FPGA [RÖM+17b] supports HLS tools of both Xilinx and Intel, while Halide-HLS [PBY+17], PolyMage-HLS [CVP+16], and RIPL only target Xilinx devices. CAPH relies upon the actor/dataflow model of computation to generate VHDL or SystemC code. Our approach could also be used to implement OpenVX by these image processing DSLs.

To the best of our knowledge, there is no publicly available OpenVX implementation for Xilinx FPGAs. Intel OpenVino [Int18] provides a few example applications that are specific to Arria-10 FPGAs. Taheri et al. [THB+18] provide some initial results for FPGAs, where the primary attention is the scheduling of statistical kernels (i.e., histogram). The image processing DSLs in [RÖM+17b; CVP+16] use similar tech-

niques to implement user applications as a streaming pipeline. Section 5.5.3 shows how to instrument these techniques for the OpenVX API. Omidian et al. [OL18] present a heuristic algorithm for the design space exploration of OpenVX graphs for FPGAs. This algorithm could be simplified by using HipaccVX' abstractions (see Section 5.4.1) instead of OpenVX' CV functions. Then it could be used in conjunction with HipaccVX to explore the design space of hardware/software platforms. Moreover, Omidian et al. [OIL18] suggest an overlay architecture for FPGA implementations of OpenVX. The proposed overlay implementation requires the optimized implementation of OpenVX' CV functions, which DomVX could generate. Furthermore, an overlay architecture based on HipaccVX's abstractions, a smaller set of functions compared to OpenVX CV functions, could reduce resource usage in [OIL18].

Intel's OpenVX implementation [AB17] is the first work extending the OpenVX standard with an interoperability API for OpenCL. This is supported in OpenVX v1.3 [The19]. Yet, performance portability still cannot be assured for the custom nodes. An OpenCL code tuned for a specific CPU might perform very poorly on FPGAs and GPU architectures [SFL+15; DWL+12]. Contrarily to our approach, the performance of this approach relies on the user code.

5.8 Conclusion

In this chapter, we proposed implementing the OpenVX standard by a set of computational abstractions that adhere to distinct memory access patterns and allow code generation for different computing platforms. We presented HipaccVX, an implementation for OpenVX using the proposed abstractions to generate code for GPUs, CPUs, and FPGAs. In this way, we get the best of both worlds (OpenVX and DSL works). HipaccVX uses Hipacc for code generation, but our approach is not restricted to a specific DSL.

Our approach relies on OpenVX' industry-standard graph specification and enables DSL-based code generation. Users are offered well-known CV functions as well as DSL elements (i.e., programming constructs, abstractions) for the description of custom nodes. As a result, programmers are allowed to write functional descriptions for custom nodes without having concerns about the performance; and, as a consequence, are allowed to write performance-portable OpenVX programs for a larger algorithm space.

Our findings show that our approach can facilitate optimizations that are beyond the scope of traditional OpenVX graph implementations that rely solely on standard computer vision functions. These optimizations can double the throughput on an Nvidia GTX GPU and decrease the resource usage on a Xilinx Zynq FPGA by 50% for our benchmarks. We showed that our proposed compiler framework could achieve

better results than the state-of-the-art approaches Nvidia VisionWorks [NVI20] and Halide-HLS [PBY+17].

Part III

High-Level Synthesis with Partial Evaluation

6 AnyHLS: High-Level Synthesis using Partial Evaluation

In previous chapters, we used template metaprogramming and developed a compiler backend for an image processing DSL to design high-level abstractions. These techniques are not easy to use – that is, modifying a compiler or implementing a well-designed generic C++ library require advanced programming skills. What is more, metaprogramming and deeply embedded DSLs do not preserve well-typedness of the generated program.

In this chapter, we present AnyHLS [ÖPM+20a; ÖPM+18], an approach to synthesize FPGA designs in a modular and abstract way. AnyHLS is able to raise the abstraction level of existing HLS tools by resorting to programming language features such as types and higher-order functions as follows: It relies on partial evaluation to specialize and to optimize high-level abstractions. Then, it generates vendor-specific HLS code for Intel and Xilinx FPGAs. This is much more productive than current C/C ++-based approaches and even DSL design techniques since no modification of a compiler is required.

AnyHLS provides a high level of portability by avoiding vendor-specific pragmas at the source code and by generating target-specific code (e.g., OpenCL or C ++) as input to existing HLS tools. Writing pragma-annotated C/C++ programs prevents portability across different vendors. Furthermore, pragmas are not first-class citizens in the language. This makes it hard to use them in a modular way or utilize them in high-level abstractions. Instead, AnyHLS use partial evaluation to apply code transformations that depend on compile-time input parameters. One example is loop unrolling shown in Section 6.3.2, which replicates its body function by specializing it for compile-time parameters, e.g., the loop index. Similarly, we define array-like memory abstractions (`Regs1D`) concisely from the desired small partitions instead of the current way of declaring an array and partitioning by a pragma (see Section 3.2.4).

As a case study, in Section 6.4, we present a library for the domain of image processing to demonstrate high productivity, modularity, and portability gains achieved by AnyHLS.

6.1 Introduction

There is an ongoing discussion whether C-based languages are good candidates for HLS [Edw06; San06; CLN+11; BRS13; KFP+18]. Yet, most commonly used HLS compilers (e.g., Vivado HLS, Altera SDK for OpenCL (AOCL), Catapult, LegUp) are based on C-based languages [NSP+15; CLN+11; Xil17c; Int17]. The modularity and readability of C/C++ or OpenCL descriptions often conflict with best coding practices of HLS compilers [EZI+19; dSBL19]. Furthermore, the lack of standardization in HLS languages and compilers hinders the portability of code across them. Often, the code optimized for one HLS tool must significantly be changed to target another HLS tool even when the same FPGA design is described. In this chapter, we advocate describing FPGA designs using functional abstractions and using partial evaluation to generate optimized HLS code.

6.1.1 Raising the Abstraction Level in HLS

As shown in Chapter 4, the abstraction level in HLS can be raised by designing libraries, DSLs or source-to-source compilers, and thus low-level implementation details can be hidden from users with no hardware design knowledge. These approaches improve the modularity and reduce code duplication but are hard to develop and maintain when well-typedness of programs are preserved. Several works [dSBL19; RAK18; EZI+19; ÖRH+17a] make extensive use of C++ template metaprogramming to provide libraries that are optimized for Vivado-HLS. Generic programs can be optimized for compile-time known values using metaprogramming techniques, but it has the following drawbacks: (i) The well-typedness of the generated program cannot be guaranteed in metaprogramming. This makes it difficult to understand error messages. (ii) Metaprograms are hard to develop, maintain, and understand since the meta language is different from the core language (C++ core vs. C++ template language). For this reason, code cannot be easily moved between the core and the meta language. (iii) Lambda expressions are not allowed to be used as template arguments in C++. We refer to [LBH+18] for more details. In particular, [RAK18; EZI+19] explain the challenges of implementing higher-order algorithms in C++ for Vivado-HLS. OpenCL C does not support template metaprogramming, thus forces users to use preprocessor macros for generic library design. Therefore, libraries developed by using C++ template metaprogramming have to be rewritten completely for OpenCL C, that is, for AOCL.

DSLs use domain-specific knowledge to parallelize algorithms and generate low-level, optimized code [ORS+13]. Programming accelerators using DSLs is thus easier, in particular for FPGAs, because the compiler performs scheduling (see Chapter 4). Other examples include Lift that targets FPGAs via algorithmic patterns [KBS+19] and Tiramisu [BRR+19] for data-parallel algorithms on dense arrays. Tiramisu takes

as input a set of scheduling commands from the user and feeds it to the polyhedral analysis of the compiler. However, a considerable portion of these scheduling primitives remains platform-specific [DBA+18]. Spatial [KFP+18] is a language for programming coarse-grained reconfigurable architectures (CGRAs) and FPGAs. Spatial provides language constructs to express control, memory, and interfaces of hardware implementation.

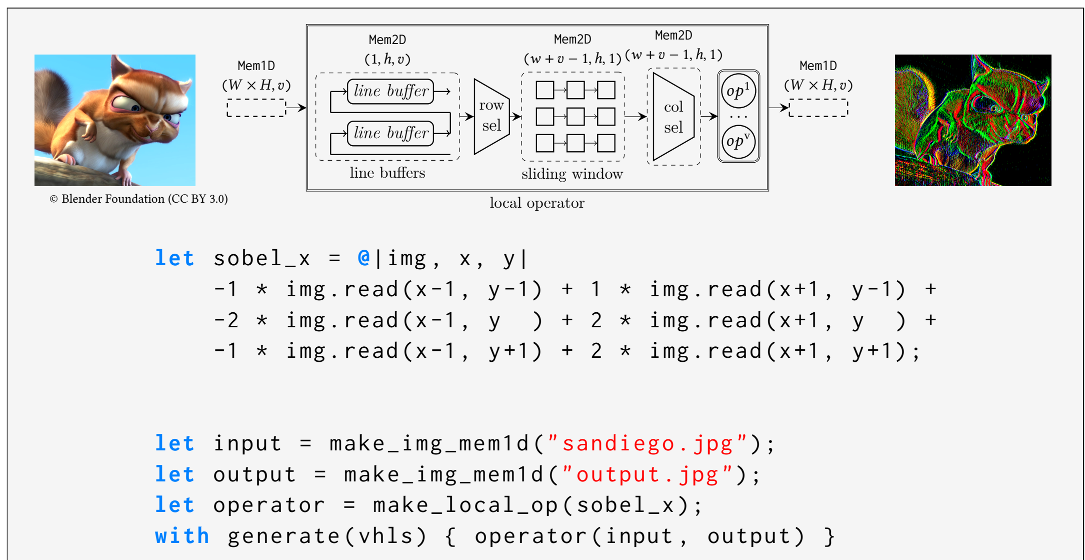
6.1.2 Main Contributions

In this chapter, it is shown that the described need to raise the abstraction level in HLS can be accomplished by using recent compiler technology, in particular by exploring the concepts of *partial evaluation* and *high-order functions*. Unlike the aforementioned DSL compilers, AnyHLS allows programmers to build the basic blocks and abstractions necessary for their application domain by themselves (see Section 6.3). AnyHLS is thereby built on top of AnyDSL [LBH+18] (see Section 6.2.1). AnyDSL offers partial evaluation to enable *shallow* embedding [LBH+15] without the need for modifying a compiler. This means that there is no need to change the compiler when adding support for a new application domain, since programmers can design custom control structures. Partial evaluation specializes algorithmic variants of a program at compile-time. Compared to metaprogramming, partial evaluation operates in a single language and preserves the well-typedness of programs [LBH+18]. Furthermore, different combinations of static/dynamic parameters can be instantiated from the same code. Please refer to Leißa et al. [LBH+18] for more details and a thorough comparison to prior techniques.

Consider Figure 6.1 for an example from image processing: With a functional language, we separate the description of the `sobel_x` operator from its realization in hardware. The hardware realization `make_local_op` is a function that specifies the data path, the parallelization, and the memory architecture. Thus, the algorithm and hardware architecture descriptions are specified by a set of higher-order functions. A partial evaluator ultimately combines these functions to generate an HLS code that delivers high-performance circuit designs when compiled using HLS tools. Since the initial descriptions are high-level, compact, and functional, they are reusable and distributable as a library. We leverage the AnyDSL compiler framework [LBH+18] to perform partial evaluation and extend it to generate input code for HLS tools targeting Intel and Xilinx FPGA devices. We claim that this approach leads to a modular and portable code other than existing HLS methods, and is able to produce highly efficient hardware implementations.

In summary, this chapter makes the following contributions based on the publications[1]:

[1]The contents of this chapter are based on and partly published in [ÖPM+20a], which has appeared in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, and [ÖPM+18],




```
let sobel_x = @|img, x, y|
    -1 * img.read(x-1, y-1) + 1 * img.read(x+1, y-1) +
    -2 * img.read(x-1, y  ) + 2 * img.read(x+1, y  ) +
    -1 * img.read(x-1, y+1) + 2 * img.read(x+1, y+1);


let input = make_img_mem1d("sandiego.jpg");
let output = make_img_mem1d("output.jpg");
let operator = make_local_op(sobel_x);
with generate(vhls) { operator(input, output) }
```

Figure 6.1: AnyHLS example: The description of the `sobel_x` algorithm is decoupled from the description of its hardware realization. The function `make_local_op` describes the hardware realization, including important transformations for exploiting parallelism and memory architecture specification. The function `generate(vhls)` selects the backend for code generation, which is Vivado HLS in this case. Ultimately, an optimized input code for HLS is generated by partially evaluating the algorithm and realization of the functions. (Figure reprinted from [ÖPM⁺20a], © 2020 IEEE)

- We present AnyHLS[2], raising the abstraction level in HLS by using *partial evaluation* of *higher-order functions* as a core compiler technology. It guarantees the *well-typedness* of the residual program and offers considerably higher productivity than existing DSL design techniques and C/C++-based approaches (see Section 6.2).
- AnyHLS offers unprecedented target independence, and thus portability, across different HLS tools by avoiding tool-specific pragma extensions and generating target-specific OpenCL or C++ code as input to existing HLS tools (see Section 6.3).
- Productivity, modularity, and portability gains are demonstrated by presenting

which has appeared in Proceedings of the Fifth International Workshop on FPGAs for Software Programmers (FSP).

[2]https://github.com/AnyDSL/anyhls

an image processing library *as a case study* in Section 6.4. For this domain, we show that a competitive performance in terms of throughput and resource usage can be achieved in comparison with existing state-of-the-art DSLs (see Section 6.5).

6.2 Overview, Background, and Related Work

In the following, we briefly discuss prior work and fundamental concepts of AnyDSL (Section 6.2.1).

In order to achieve good quality of results (QoR), HLS languages often demand programmers to specify the hardware architecture of an application instead of just its algorithm. For this reason, HLS languages offer hardware-specific pragmas. This ad-hoc mix of software and hardware features makes it difficult for programmers to optimize an application. In addition, most HLS tools rely on their own C dialect, which prevents code portability. For example, Xilinx Vivado HLS [Xil17c] uses C++ as base language while Intel SDK [Int17] (formerly Altera) uses OpenCL C. These severe restrictions make it hard to use existing HLS languages in a portable and modular way.

6.2.1 AnyDSL Compiler Framework

AnyDSL[3] [LBH+15; LBH+18] is a compiler framework for designing high-performance, domain-specific libraries. It provides the imperative and functional language Impala. Impala's syntax is inspired by Rust. We will now briefly discuss Impala's most important features that we rely on in AnyHLS.

Partial Evaluation

Partial evaluation is a technique for program optimization by specialization of compile-time known values. Assume that each input of a program F is classified as either static s or dynamic d, and values for all of the static inputs are given. Then, partial evaluation produces an optimized (residual) program F_s such that

$$[[Fs]](d) = [[F]](s, d) \tag{6.1}$$

and running F_s on the dynamic inputs produces the same result as running the original program F on all of the inputs [JGS93]. Compiler techniques such as constant propagation, loop unrolling, or inlining are examples where partial evaluation can be successfully applied. Typically, the user has no control when these optimizations are applied from a compiler.

[3] https://anydsl.github.io

Impala allows programmers to partially evaluate [Fut82] a program at compile time. Programmers control the partial evaluator via *filters* [Con88]. These are Boolean expressions of the form `@(expr)` that annotate function signatures. *Each* call site instantiates the callee's filter with the corresponding argument list. The call is specialized when the expression evaluates to `true`. The expression `?expr` yields `true`, if `expr` is known at compile-time; the expression `$expr` is never considered constant by the evaluator. For example, the following `@(?n)` filter will only specialize calls to `pow` if `n` is statically known at compile-time:

Listing 6.1: Description of a power function in AnyDSL. The compiler will generate an optimized code accordingly after evaluating the input parameters x and n. For instance, a compile-time constant will be generated when both x and n are known at compile time.

```
1  fn @(?n) pow(x: int, n: int) -> int {
2    if n == 0 {
3      1
4    } else {
5      if n % 2 == 0 {
6        let y = pow(x, n / 2);
7        y * y
8      } else {
9        x * pow(x, n - 1)
10     }
11   }
12 }
```

Thus, the calls

```
let z = pow(x, 5);                let z = pow(3, 5);
```

will result in the following equivalent sequences of instructions after specialization:

```
let y = x * x;                    let z = 243;
let z = x * y * y;
```

The `@` is a syntactic sugar (i.e., available as shorthand for `@(true)`) sets the partial evaluator to always specialize the annotated function.

As a hardware circuit description must be static (e.g., memory size) for being synthesizable, types, loops, functions, and interfaces must be resolved at compile-time [ÖRH+17a; ÖRH+16; RAK18; EZI+19]. Partial evaluation has many advantages compared to metaprogramming as discussed in Section 6.1.1. Hence, Impala's partial evaluation is particularly useful to optimize HLS descriptions.

Generators

Impala provides the following `for` syntactic sugar[4] for describing iterations as higher-order functions. The loop

```
for var1, ..., varn in iter(arg1, ..., argn) { /* ... */ }
```

translates to

```
iter(arg1, ..., argn, |var1, ..., varn| { /* ... */ });
```

The body of the `for` loop and the iteration variables constitute an anonymous function

```
|var1, ..., varn| { /* ... */ }
```

that is passed to `iter` as the last argument. We call functions that are invokable like this *generators*. Domain-specific libraries implemented in Impala make busy use of these features as they allow programmers to write custom generators that take advantage of both domain knowledge and certain hardware features, as we will see in the next section.

Generators are particularly powerful in combination with partial evaluation. Consider the following functions:

```
type Body = fn(int) -> ();
fn @(?a & ?b) unroll(a: int, b: int, body: Body) -> () {
  if a < b { body(a); unroll(a+1, b, body) }
}
fn @ range(a: int, b: int, body: Body) -> () {
  unroll($a, b, body)
}
```

Both generators iterate from a (inclusive) to b (exclusive) while invoking body each time. The filter `unroll` implies the partial evaluator to completely unroll the recursion if both loop bounds are statically known at a particular call site.

6.3 The AnyHLS Library

Efficient and resource-friendly FPGA designs require application-specific optimizations. These optimizations and transformations are well known in the community. For example, de Fine Licht, Meierhans, and Hoefler [dMH18] discuss the key transformations of HLS codes such as loop unrolling and pipelining. They describe the whole hardware design from the low-level memory layout to the operator implementations with support for low-level loop transformations throughout the design. In our setting, the programmer defines and provides these abstractions using AnyDSL

[4] In computer science, *syntactic sugar* is a term used for a programming language syntax that allows expressing a verbose form in an alternative style that is more clear and concise. The name is coined by Landin [Lan65] in 1964 as part of a simple ALGOL-like programming language.

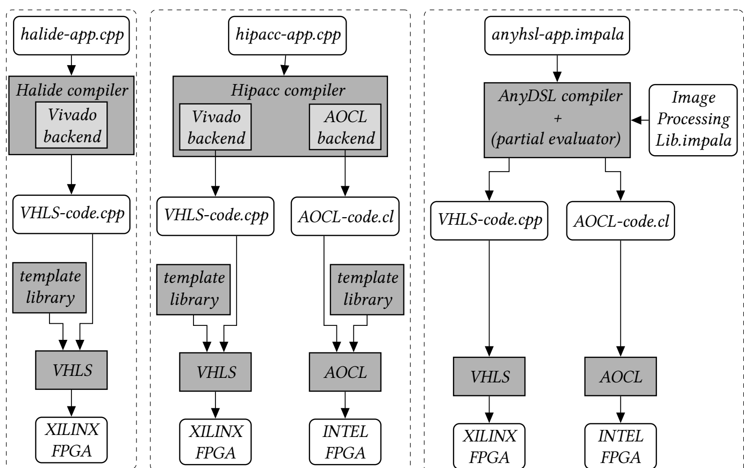


Figure 6.2: FPGA code generation flows for Halide, Hipacc, and AnyHLS (from left to right). VHLS and AOCL are used as acronyms for Vivado HLS and Intel FPGA SDK for OpenCL, respectively. Halide and Hipacc rely on domain-specific compilers for image processing. They use template libraries (developed by using C-based metaprogramming techniques) to alleviate the tasks of their source-to-source compiler backends (see Section 4.5 for more details). AnyHLS allows defining all abstractions for the domain of two-dimensional image processing algorithms in a language called Impala and relies on partial evaluation for code specialization. This ensures maintainability and extensibility of the provided compilation flow for a domain, image processing in this thesis. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

for the domain of 2D image processing algorithms in the form of a library. We rely on partial evaluation to combine those abstractions and to remove overhead associated with them. Ultimately, the AnyDSL compiler synthesizes optimized HLS code (C++ or OpenCL C) from a given functional description of an algorithm as shown in Figure 6.2. The generated code goes to the selected HLS tool. This is in contrast to other domain-specific approaches like Halide-HLS [PBY+17] or Hipacc [RÖM+17b], which rely on domain-specific compilers to instantiate predefined templates or macros. Hipacc makes use of two distinct libraries to synthesize algorithmic abstractions to Vivado-HLS and Intel AOCL, while AnyHLS uses the same image processing library that is described in Impala.

6.3.1 HLS Code Generation

For HLS code generation, we implemented an intrinsic named `vhls` in AnyHLS to emit Vivado HLS and an intrinsic named `opencl` to emit AOCL:

```
with vhls() { body() }                with opencl() { body() }
```

With `opencl`, we use a grid and block size of `(1, 1, 1)` to generate a single work-item kernel, as the official AOCL documentation recommends [Int17]. We extended AnyDSL's OpenCL runtime by the extensions of Intel OpenCL SDK. To provide an abstraction over both HLS backends, we create a wrapper `generate` that expects a code generation function:

```
type Backend = fn(fn() -> ()) -> ();
fn @ generate(be: Backend, body: fn() -> ()) -> () {
  with be() { body() }
}
```

Switching backends is now just a matter of passing an appropriate function to `generate`:

```
let backend = vhls; // or opencl
with generate(backend) { body() }
```

6.3.2 Building Abstractions for FPGA Designs

In the following, we present abstractions for the key transformations and design patterns that are common in FPGA design. These include (a) important loop transformations, (b) control flow and data flow descriptions such as reductions, (c) FSMs, and (d) the explicit utilization of different memory types. Approaches like Spatial [KFP+18] expose these patterns within the language—new patterns require dedicated support from the compiler. Hence, these languages and compilers are restricted to a specialized application domain they have been designed for. In AnyHLS, Impala's functional language and partial evaluation allow us to design the abstractions needed for FPGA synthesis in the form of a library. New patterns can be added to the library without dedicated support from the compiler. This makes AnyHLS easier to extend compared to the approaches mentioned afore.

Loop Transformations

C++ compilers usually provide certain preprocessor directives to trigger particular code transformations. A common feature is to unroll loops (see Listings 6.2 and 6.3).

Listing 6.2: Loop unrolling using a pragma

```
1 for (int i=0; i<N/W; ++i) {
2   for (int w=0; w<W; ++w) {
3     #pragma unroll
4     body(i*W + w);
5   }
6 }
```

Listing 6.3: Loop unrolling using the loop abstractions in AnyHLS

```
1  for i in range(0, N/W) {
2    for w in unroll(0, W) {
3
4      body(i*W + w);
5    }
6  }
```

Such pragmas are built into the compiler. The Impala version (shown in Listing 6.3) uses generators that are entirely implemented as a library. Partial evaluation optimizes Impala's `range` and `unroll` abstractions as well as the input body function according to their static inputs, i.e., `N`, `W`. The residual program consists of the consecutive *body* function according to the value of the `W` as shown in Figure 6.3. This generates a concise and clean code for the target HLS compiler, which is drastically different from using a pragma.

Generators, unlike C++ `pragmas`, are first-class citizens of the Impala language. This allows programmers to implement sophisticated loop transformations. For example, the function `tile` shown in Listing 6.4 returns a new generator. It instantiates a tiled loop nest of the specified tile `size` with the `Loops` `inner` and `outer`.

Listing 6.4: Implementation of loop tiling abstraction for AnyHLS

```
1  type Loop = fn(int, int, fn(int) -> ()) -> ();
2  fn @ tile(size: int, inner: Loop, outer: Loop) -> Loop {
3    @|beg, end, body| outer(0, (end-beg)/size,
4      |i| inner(i*size + beg, (i+1)*size + end, |j| body))
5  }
6
7  let schedule = tile(W, unroll, range);
8  for i in schedule(0, N) {
9    body(i)
10 }
```

Passing `W` for the tiling `size`, `unroll` for the inner loop, and `range` for the outer loop yields a generator that is identical to the loop nest at the beginning of this paragraph. With this description, we can reuse or explore iteration techniques without touching the actual body of a **`for`** loop. For example, consider the processing options for a two-dimensional loop nest as shown in Figure 6.3: When just passing `range` as `inner` and `outer` loop, the partial evaluator will keep the loop nest and, hence, not unroll body and instantiate it only once. Unrolling the inner loop replicates body and increases the bandwidth requirements accordingly. Unrolling the outer loop also replicates `body`, but in a way that benefits data reuse from the temporal locality of an iterative algorithm. Unrolling both loops replicate `body` for increased bandwidth and data reuse for the temporal locality.

C/C++-based HLS solutions often use a `pragma` to mark a loop amenable for pipelining. This means parallel execution of the loop iterations in hardware. For example,

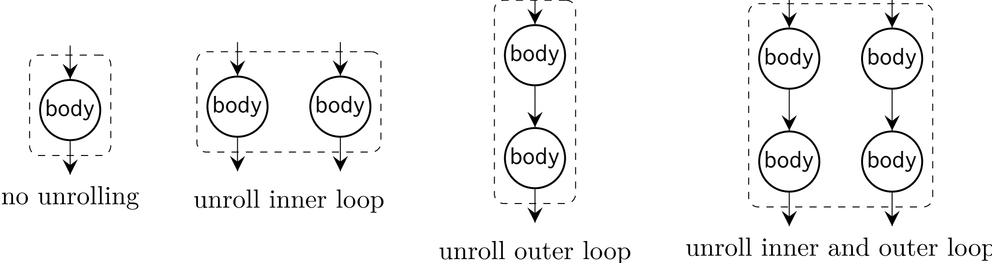


Figure 6.3: Parallel processing. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

the code in Listing 6.6 uses an initiation interval (`II`) of `3`:

Listing 6.5: Loop tiling pragma in C-based HLS

```
for (int i=0; i<N; ++i) {
  #pragma HLS pipeline II=3
  body(i);
}
```

Instead of using a pragma as in Listing 6.6, AnyHLS uses the intrinsic generator `pipeline` as shown in Listing 6.6. Unlike the above loop abstractions (e.g., unroll), Impala then emits a tool-specific pragma to implement/refine the `pipeline` abstraction. This provides portability across different HLS tools. Furthermore, it allows the programmer to invoke and pass around `pipeline`—just like any other generator.

Listing 6.6: Loop tiling in AnyHLS

```
let II = 3;
for i in pipeline(II, 0, N) {
  body(i)
}
```

Reductions

Reductions are useful in many contexts. The `reduce` function in Listing 6.7 takes an array of values, a range within, and an operator:

Listing 6.7: Implementation of the reduction abstraction in AnyHLS

```
type T = int;
fn @(?beg & ?end) reduce(beg: int, end: int, input: &[T],
                         op: fn(T, T) -> T) -> T {
  let n = end - beg;
  if n == 1 {
    input(beg)
  } else {
```

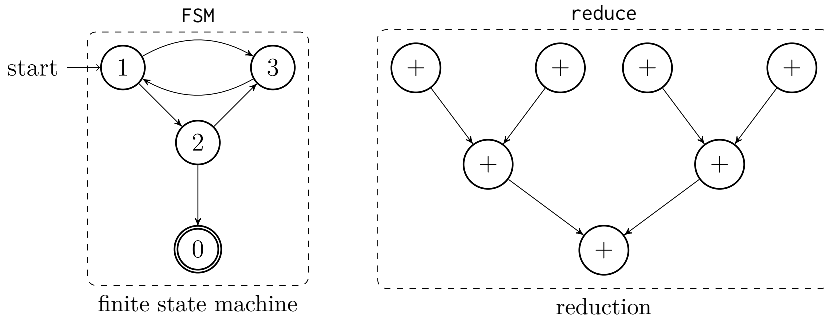


Figure 6.4: Multiplexers, finite state machines, and reductions. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

```
8       let m = (end + beg) / 2;
9       let a = reduce(beg, m, input, op);
10      let b = reduce(m, end, input, op);
11      op(a, b)
12    }
13  }
```

The recursion in the above filter, will be completely unfolded if the range is statically known. Thus,

```
reduce(0, 4, [a, b, c, d], |x, y| x + y)
```

will yield:

```
(a + b) + (c + d)
```

Finite State Machines

AnyHLS models computations that depend not only on the inputs but also on an internal state with an FSM. To define an FSM, programmers need to specify states and a transition function that determines when to change the current state based on the machine's input. This is especially beneficial for modeling control flow. To describe an FSM in Impala, we introduce types to represent the states and the machine as shown in Listing 6.8:

Listing 6.8: Implementation of the FSM state in AnyHLS

```
1  type State = int;
2  struct FSM {
3    add: fn(State, fn() -> (), fn() -> State) -> (),
4    run: fn(State) -> ()
5  }
```

An object of type FSM provides two operations: adding one state with add or running the computation. The add method takes the name of the state, an action to be performed for this state, and a transition function associated with this state. Once all states are added, the programmer runs the machine by passing the initial state as an input parameter. The example in Listing 6.9 adds 1 to every element of an array.

Listing 6.9: An example description of an FSM in AnyHLS

```
1 let buf = /*...*/;
2 let mut (idx, pixel) = (0, 0);
3 let fsm = make_fsm();
4 fsm.add(Read, || pixel = buf(idx),
5               || if idx>=len { Exit } else { Compute });
6 fsm.add(Compute, || pixel += 1,           || Write);
7 fsm.add(Write,   || buf(idx++) = pixel, || Read );
8 fsm.run(Read);
```

Similar to the other abstractions introduced in this section, the constructor for an FSM is not a built-in function of the compiler but a regular Impala function. In some cases, we want to execute the FSM in a pipelined way. For this scenario, we add a second method run_pipelined. As all the methods, e.g., make_fsm, add, run, are annotated for partial evaluation (by @), input functions to these methods will be optimized according to their static inputs. Ultimately, AnyHLS will emit the states of an FSM as part of a loop according to the selected run method.

Memory Types and Memory Abstractions

FPGAs have different memory types of varying sizes and access properties. Impala supports three memory specific to hardware design (see Figure 6.5): global memory, on-chip memory, registers. In addition, it provides a memory abstraction for using HLS streams. Global memory (typically DRAM) is allocated on the host using our runtime and accessed through regular pointers. On-chip memory (e.g., BRAM or M10K/M20K) for the FPGA is allocated using the reserve_onchip compiler intrinsic. Memory accesses using the pointer returned by this intrinsic will map to on-chip memory. Standard variables are mapped to registers, and a specific **stream** type is available to allow for the communication between FPGA kernels. Memory-wise, a **stream** is mapped to registers or on-chip memory by the HLS tools. These FPGA-specific memory types in Impala will be mapped to their corresponding tool-specific declarations in the residual program (on-chip memory will be defined as local memory for AOCL, whereas it will be defined as an array in Vivado HLS).

Memory partitioning An array partitioning pragma must be defined as follows to implement a C array with hardware registers using Vivado HLS [Xil17c]:

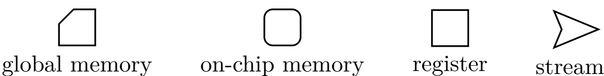


Figure 6.5: Memory types provided for FPGA design. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

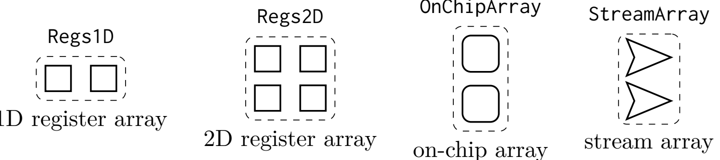


Figure 6.6: Memory abstractions. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

Listing 6.10: A typical way of partitioning an array by using pragmas in existing HLS tools.

```
1  typedef int T;
2  T Regs1D[size];
3  #pragma HLS variable=Regs1D array_partition dim=0
```

Other HLS tools offer similar pragmas for the same task. Instead, AnyHLS provides a description of a register array by the recursive declaration of registers as in Listing 6.11, and without using any tool-specific pragma.

Listing 6.11: Recursive description of a register array using partial evaluation instead of declaring an array and partitioning it by HLS pragmas.

```
1  type T = int;
2  struct Regs1D {
3    read:  fn(int) -> T,
4    write: fn(int, T) -> (),
5    size:  int
6  }
7  fn @ make_regs1d(size: int) -> Regs1D {
8    if size == 0 {
9      Regs1D {
10       read:  @|_| 0,
11       write: @|_, _| (),
12       size:  size
13     }
14   } else {
15     let mut reg: T;
16     let others = make_regs1d(size - 1);
17     Regs1D {
18       read:  @|i|    if i+1 == size { reg }
19                       else { others.read(i) },
```

```
20        write: @|i, v| if i+1 == size { reg = v }
21                       else { others.write(i, v) },
22        size:  size
23      }
24    }
25  }
```

When the size is not zero, each recursive call to this function allocates a register variable named reg, and creates a smaller register array with one element less named others. The read and write functions test if the index i is equal to the index of the current register. In the case of a match, the current register is used. Otherwise, the search continues in the smaller array. The generator (make_regs1d) returns an Impala variable that can be read and written by index values (regs in the following code), similar to C arrays.

```
let regs = make_regs1d(size);
```

However, it defines size number of registers in the residual program instead of declaring an array and partitioning it by tool-specific pragmas as in Listing 6.10. The generated code does not contain any compiler directives; hence it can be used for different HLS tools (e.g., Vivado HLS, AOCL). Since we annotated make_regs1d, read, and write for partial evaluation, any call to these functions will be inlined recursively. This means that the search to find the register to read to or write from will be performed at compile time. These registers will be optimized by the AnyDSL compiler, just like any other variables: unnecessary assignments will be avoided, and an HLS code will be generated.

Correspondingly, AnyHLS provides generators (similar to Listing 6.11) for one and two-dimensional arrays of on-chip memory (e.g., line buffers in Section 6.4), global memory, and streams (as illustrated in Figure 6.6) instead of using memory partitioning pragmas encouraged in existing HLS tools (as in Listing 6.10).

6.4 A Library for Image Processing on FPGA

AnyHLS allows for defining domain-specific abstractions and optimizations that are used and applied prior to generating customized input to existing HLS tools. In this section, we introduce a library that is developed to support HLS for the domain of image processing applications. It is based on the fundamental abstractions introduced in Section 6.3.2. Our low-level implementation is similar to existing domain-specific languages targeting FPGAs [HDD+16; RÖM+17b]. For this reason, we focus on the interface of our abstractions as seen by the programmer.

We design applications by decoupling their algorithmic description from their schedule and memory operations. For instance, typical image operators, such as the following Sobel filter, just resort to the make_local_op generator.

Listing 6.12: Sobel filter as example local operator application described by using our library.

```
1  let sobel_x = stencil(@|img, x, y|
2      -1 * img.read(x-1, y-1) + 1 * img.read(x+1, y-1) +
3      -2 * img.read(x-1, y  ) + 2 * img.read(x+1, y  ) +
4      -1 * img.read(x-1, y+1) + 2 * img.read(x+1, y+1),
5      extents(1, 1));    // stencil pattern for 3x3 filter
6
7  let img = create_host_image(input.png);
8  let dx  = create_host_image(output.png);
9  let vect_factor = 4;
10 let operator = make_local_op(vect_factor, sobel_x, mirror, mirror);
11
12 with generate(vhls) { operator(img, dx); }
```

Similarly, we implement a point operator for RGB-to-gray color conversion as follows (Listing 6.13):

Listing 6.13: RGB-to-gray color conversion as example point operator application described by using our library.

```
1  let img  = create_host_image(input.png);
2  let gray = create_host_image(output.png);
3  let vect_factor = 4;
4
5  let rgb2gray = make_point_op(@ |pix| {
6      let r = pix & 0xFF;
7      let g = (pix >>  8) & 0xFF;
8      let b = (pix >> 16) & 0xFF;
9      (r + g + b) / 3
10   });
11
12 with generate(aocl) { rgb2gray(vect_factor, img, gray); }
```

The image data structure is opaque. The target platform mapping determines its layout. AnyHLS provides common border handling functions as well as point and global operators such as reductions (see Section 6.3.2). These operators are composable to allow for more sophisticated ones.

6.4.1 Vectorization

Image processing applications consist of loops that possess a very high degree of spatial parallelism. This should be exploited to reach the bandwidth speed of memory technologies. A resource-efficient approach, so-called *vectorization* or *loop coarsening*, is to aggregate the input pixels to vectors and process multiple input data at the same time to calculate multiple output pixels in parallel [SRH+15; ÖRH+17b; SGE+18].

This replicates only the arithmetic operations applied to data (so-called datapath) instead of the whole accelerator, similar to SIMD architectures. Vectorization requires a control structure specialized to a considered hardware design. We support the automatic vectorization of an application by a given factor `v` when using our image processing library. In particular, our library uses the vectorization techniques proposed in Chapter 3 [ÖRH+17b]. For example, the `make_local_op` function has an additional parameter to specify the desired vectorization and will propagate this information to the functions it uses internally: `make_local_op(op, v)`. For brevity, we omit the parameter for the vectorization factor for the remaining abstractions in this section.

6.4.2 Memory Abstractions for Image Processing

Memory Accessor

An FPGA implementation of an algorithm has to be concise about memory addressing since ambiguous code induces a complicated hierarchy that leads to waste of resources and slow execution times. In particular, commercial HLS tools perform poorly when on-chip-memory blocks are configured from arrays of primitive types [CWY+17]. In order to optimize memory access and encapsulate the contained memory type (on-chip memory, etc.) into a data structure, we decouple the data transfer from the data use via the following memory abstractions:

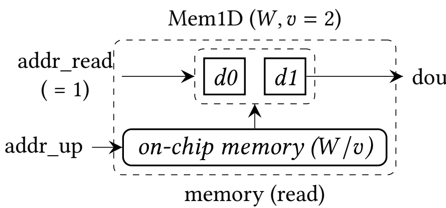
Listing 6.14: One-dimensional memory accessor structure in AnyHLS

```
1 struct Mem1D {
2   read:   fn(int) -> T,
3   write:  fn(int, T)->(),
4   update: fn(int) -> (),
5   size:   int
6 }
```

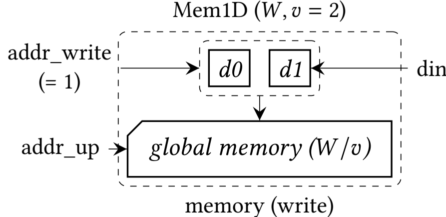
Listing 6.15: Two-dimensional memory accessor structure in AnyHLS

```
1 struct Mem2D {
2   read:   fn(int, int) -> T,
3   write:  fn(int, int, T)->(),
4   update: fn(int, int) -> (),
5   width:  int, height: int
6 }
```

Similar to hardware design practices, these memory abstractions require the memory address to be `update`d before the `read`/`write` operations. The `update` function transfers data from/to the encapsulated memory to/from staging registers using vector data types. Then, the `read`/`write` functions access an element of the vector. This increases data reuse and DRAM-to-on-chip memory bandwidth [CCF+16].



(a) An `update` call loads the *addr-up*th data-beat from the memory to the registers, whereas a `read` outputs a datum from the *addr-read*th register.



(b) An `update` call loads a data-beat stored in registers to the *addr-up*th address of the memory, whereas a `write` inputs a datum to the *addr-write*th register.

Figure 6.7: Memory accessor structure is used to abstract the memory type. It decouples the index arithmetic from memory access. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

For example, consider Figure 6.7 where two memory accessors are used to encapsulate the contained memory types (on-chip memory, etc.) into an opaque data structure. This makes it suitable as a gluing data type between the library components.

Stream Processing

Inter-kernel dependencies of an algorithm should be accessed on-the-fly in combination with fine-granular communication in order to pipeline the full implementation with a fixed throughput. That is, as soon as a block produces one data, the next block consumes it. In the best case, this requires only a single register of a small buffer instead of reading/writing to temporary images:

We define a stream between two kernels as shown in Listing 6.16.

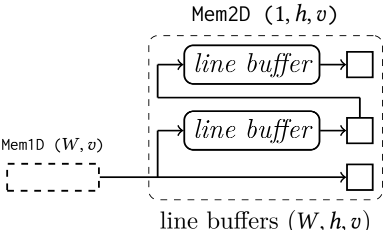
Listing 6.16: Creating a memory accessor for an HLS stream in AnyHLS

```
1 fn make_mem_from_stream(size: int, data: stream) -> Mem1D;
```

Line Buffers

Storing an entire image to on-chip memory before execution is not feasible since on-chip memory blocks are limited in FPGAs. On the other hand, feeding the data on demand from main memory is extremely slow. Still, it is possible to leverage fast

on-chip memory by using it as FIFO buffers containing only the necessary lines of the input images (W pixels per line).



This enables parallel reads at the output for every pixel read at the input. We model a line buffer as shown in Listing 6.18.

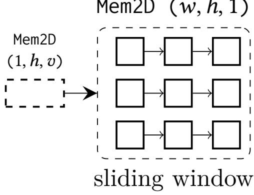
Listing 6.17: Creating a line buffer in AnyHLS

```
1  type LineBuf1D = fn(Mem1D) -> Mem1D;
2  fn make_linebuf1d(width: int) -> LineBuf1D;
3  // similar for LineBuf2D
```

Akin to `Regs1D` (see Section 6.3.2), a recursive call builds an array of line buffers (each line buffer will be declared by a separate memory component in the residual program similar to the on-chip array in Figure 6.6).

Sliding Window

Registers are the most amenable resources to hold data for highly parallelized access. A sliding window of size $w \times h$ updates the constituting shift registers by a new column of h pixels and enables parallel access to $w \cdot h$ pixels.



This provides high data reuse for temporal locality and avoids waste of on-chip memory blocks that might be utilized for a similar data bandwidth. Our implementation uses `make_regs2d` for an explicit declaration of registers and supports pixel-based indexing at the output.

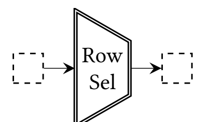
Listing 6.18: Creating a sliding window in AnyHLS

```
1  type Swin2D = fn(Mem2D) -> Mem2D;
2  fn @ make_sliding_window(w: int, h: int) -> Swin2D {
3    let win = make_regs2d(w, h);
4    // ...
5  }
```

This will instantiate $w \cdot h$ registers in the residual program, as explained in Section 6.3.2.

1D Data Selection for Border Handling

We provide border handling abstractions that select data beside the sliding window to implement one-dimensional multiplexer arrays [ÖRH+17b]. Their hardware implementation is explained in Chapter 3. Column and row selection apply the border handling only in x- and y–direction, respectively. The sliding window expects the row selection as input, while the column selection returns a `Mem2D`: We refer to [ÖPM+18] for more details on these data selection abstractions and their use for image border handling.



(a) Row selection is a function that inputs a data assignment function.

(b) Col selection creates a `Mem2D` structure at the output.

Figure 6.8: 1D data selection functions to implement border handling. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

6.4.3 Loop Abstractions for Image Processing

Point Operators

Algorithms such as image scaling and color transformation calculate an output pixel for every input pixel. The point operator abstraction (see Listing 6.19) in AnyHLS yields a vectorized pipeline over the input and output image. This abstraction is parametric in its vector factor `v` and the desired operator function `op`.

Listing 6.19: Implementation of the point operator abstraction.

```
1  type PointOp = fn(Mem1D) -> Mem1D;
2  fn @ make_point_op(v: int, op: Op) -> PointOp {
3    @ |img, out| {
4       for idx in pipeline(1, 0, img.size) {
5         img.update(idx);
6         for i in unroll(0, v) {
7           out.write(i, op(img.read(i)));
8         }
9         out.update(idx);
10      }
```

```
11    }
12 }
```

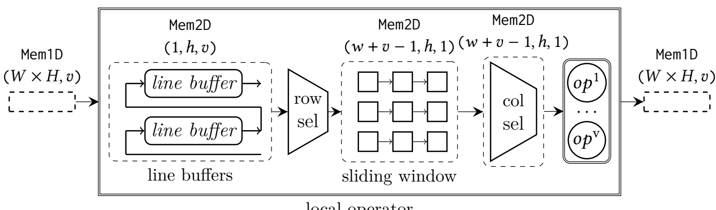
The total latency is

$$L = L_{arith} + \lceil W/v \rceil \cdot H \text{ cycles} \tag{6.2}$$

where W and H are the width and height of the input image, and L_{arith} is the latency of the data path.

Local Operators

Algorithms such as Gaussian blur and Sobel edge detection calculate an output pixel by considering the corresponding input pixel and a certain neighborhood of it in a local window. Thus, a local operator with a $w \times h$ window requires $w \cdot h$ pixel reads for every output. The same $(w-1) \cdot h$ pixels are used to calculate results at the image coordinates (x, y) and $(x+1, y)$. This spatial locality is transformed into temporal locality when input images are read in raster order for burst mode, and subsequent pixels are sequentially processed with a streaming pipeline implementation. The local operator implementation in AnyHLS (shown in Listing 6.20) consists of line buffers and a sliding window to hold dependency pixels in on-chip memory and calculates a new result for every new pixel read.



This provides a throughput of v pixels per clock cycle at the cost of an initial latency (v is the vectorization factor)

$$L_{initial} = L_{arith} + (\lfloor h/2 \rfloor \cdot \lceil W/v \rceil + \lfloor \lceil w/v \rceil / 2 \rfloor) \tag{6.3}$$

that is spent for caching neighboring pixels of the first calculation. The final latency is thus:

$$L = L_{initial} + (\lceil W/v \rceil \cdot H) \tag{6.4}$$

Compared to the local operator in Figure 6.1, we also support boundary handling. We specify the extent of the local operator (filter size / 2) as well as functions specifying the boundary handling for the lower and upper bounds. Then, row and column selection functions apply border handling correspondingly in x- and y–directions by using one-dimensional multiplexer arrays similar to Özkan et al. [ÖRH+17b].

Listing 6.20: Implementation of the local operator abstraction.

```
type LocalOp = fn(Mem1D) -> Mem1D;
fn @ make_local_op(v: int, op: Op, ext: Extents,
                   bh_lower: FnBorder,
                   bh_upper: FnBorder) -> LocalOp {
  @ |img, out| {
    let mut (col, row, idx) = (0, 0, 0);
    let wait = /* initial latency */
    let fsm = make_fsm();
    fsm.add(Read, || img.update(idx), || Compute);
    fsm.add(Compute, || {
      line_buffer.update(col);
      sliding_window.update(row);
      col_sel.update(col);
      for i in unroll(0, v) {
        out.write(i, op(col_sel.read(i)));
      }
    }, || if idx > wait { Write } else { Index });
    fsm.add(Write, || out.update(idx-wait-1), || Index);
    fsm.add(Index, || {
      idx++; col++;
      if col == img_width { col=0; row++; }
    }, || if idx < img.size { Read } else { Exit });
    fsm.run_pipelined(Read, 1, 0, img.size);
  }
}
```

6.5 Evaluation and Results

In the following, we compare the quality of synthesized circuits using AnyHLS and two other state-of-the-art domain-specific approaches including Halide-HLS [PBY+17] and Hipacc [RÖM+17b]. The generated HLS codes are compiled using Intel FPGA SDK for OpenCL 18.1 and Xilinx Vivado HLS 2017.2 targeting a Cyclone V GT 5CGTD9 FPGA and a Zynq XC7Z020 FPGA, respectively.

The generated hardware designs are evaluated for throughput, latency, and resource utilization. FPGAs possess two types of resources: (i) computational: LUTs and DSP blocks; (ii) memory: FFs and on-chip memory (BRAM/M20K). A SLICE/ALM is comprised of look-up tables (LUTs) and flip flops, thus indicate the resource usage when considered with the DSP block and on-chip memory blocks.

The implementation results presented for Vivado HLS feature only the kernel logic, while those by Intel OpenCL include PCIe interfaces. The execution time of an FPGA circuit (Vivado HLS implementation) equals to T_{clk} · latency, where T_{clk} is the period of the maximum achievable clock (lower is better). We measured the timing results for Intel OpenCL by executing the applications on a Cyclone V GT

5CGTD9 FPGA. This is the case for all analyzed applications. We have no intention nor license rights [Xil14, §4] [Int09, §2] to benchmark and compare the considered FPGA technologies or HLS tools.

6.5.1 Applications

In our experimental evaluation, we consider the following applications:

- ***Gaussian (Gauss)*** blurring an image with a 5×5 integer kernel
- ***Harris corner detector (Harris)*** consisting of 9 kernels that resort to integer arithmetic and horizontal/vertical derivatives
- ***Jacobi*** smoothing an image with a 3×3 integer kernel
- ***filter chain (FChain)*** consisting of 3 convolution kernels as a pre-processing algorithm
- ***bilateral filter (Bilateral)***, a 5×5 floating-point kernel as an edge-preserving and noise-reducing function based on exponential functions
- ***mean filter (MF)***, a 5×5 filter that determines the average within a local window via 8-bit arithmetic
- ***SobelLuma***, an edge detection algorithm provided as a design example by Intel. The algorithm consists of RGB to Luma color conversion, Sobel filters, and thresholding

6.5.2 Optimizations

AnyHLS exploits stream processing and performs implicit parallelization. The following subsections show the impact of those optimizations applied as a set of functions using partial evaluation of an initial specification.

Stream Processing

Memory transfers between FPGA's programmable logic and external memory are one of the most time-consuming parts of many image processing applications. AnyHLS streaming pipeline optimization passes dependency pixels directly from the producer to the consumer kernel, as explained in Section 6.4.2. This allows pipelined kernel execution and makes intermediate images between kernels superfluous. The more intermediate images are eliminated, the better the performance of the resulting designs. For example, this eliminates 8 intermediate images in Harris corner and 2 in filter chain, see Figure 6.9 for the performance impact.

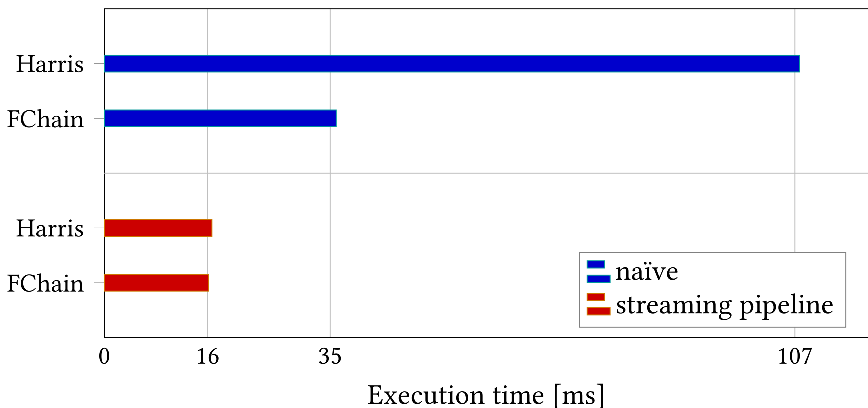


Figure 6.9: Execution time for naïve and streaming pipeline implementations of the Harris and FChain for an Intel Cyclone V FPGA for images of size 1024 × 1024. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

The throughput of both streaming pipeline implementations is indeed determined by their slowest individual kernel, which is a local operator. Consider Table 6.1, which displays the Vivado HLS reports. The latency results correspond exactly to the values computed using Eq. (6.4).

Table 6.1: Streaming pipeline implementations of Harris and FChain on a Xilinx Zynq. Data is transferred to the FPGA only once, thus similar throughputs are achieved. Images sizes are 1024 × 1024, $v = 1$, $f_{target} = 200$ MHz.

App.	Largest mask	Sequential Dependency	Latency [cyc.]	Throughput [MB/s]
FChain	5 × 5	local + local + local	1050649	821
Harris	3 × 3	local + local + point	1049634	825

Vectorization

Many FPGA implementations benefit from parallel processing to exploit available memory bandwidth. AnyHLS implicitly parallelizes a given image pipeline by a *vectorization factor* v. As an example, Figure 6.10 shows the PPnR results, along with the achieved memory throughput for different vectorization factors for the mean filter on a Cyclone V. The speedup is almost linear, whereas resource utilization is sub-linear to the vectorization factor, as Figure 6.10 depicts. AnyHLS exploits the data reuse between consecutive iterations of the local operators. Data is read and written with the vectorized data types. The line buffers and the sliding window are extended to hold dependency pixels for vectorized processing. Thus, only the datapath is replicated instead of the whole accelerator implementation (see Section 6.4.1). All the

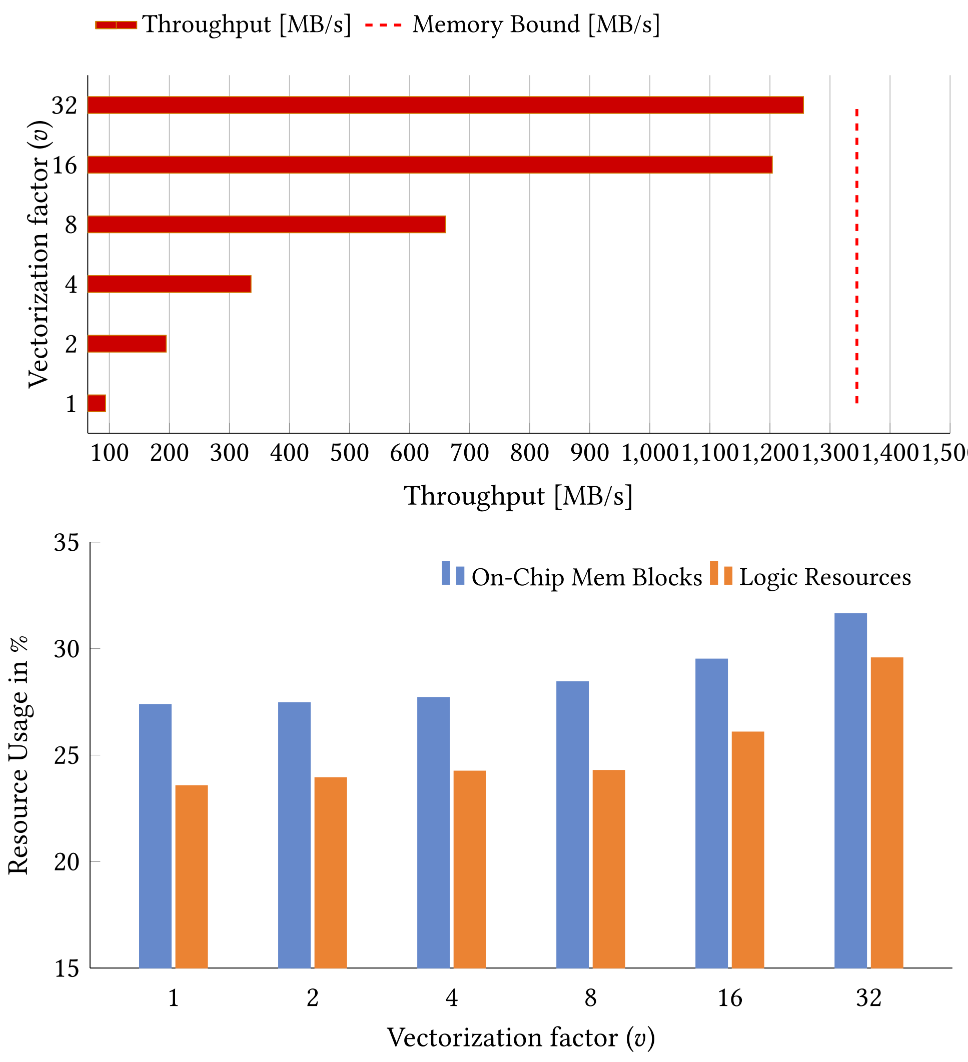


Figure 6.10: post place and route results of AnyHLS's mean filter implementation on an Intel Cyclone V FPGA. The memory bound of the device for our setup is 1344.80 MB/s (measured by Intel's diagnosis tool). (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

considered applications except Bilateral in Figure 6.12 reach the memory bound of the Cyclone V FPGA. Bilateral is compute-bound due to its large number of floating-point operations.

6.5.3 Hardware Design Evaluation

We evaluate the generated hardware designs based on their throughput, latency, and resource utilization. As a reference, we use the designs generated by Halide-HLS [PBY+17] and Hipacc [RÖM+17b], two state-of-the-art image processing DSLs that generate better results than previous approaches (e.g., Xilinx OpenCV). In contrast to these, which implement dedicated HLS code generators, AnyHLS is essentially implemented as a library within the AnyDSL framework, as illustrated in Figure 6.2. Our focus is to show that higher-order abstractions, together with partial evaluation, are powerful enough to design a library targeting different HLS compilers.

Experiments using Xilinx Vivado HLS

We evaluate the results of circuits generated using AnyHLS in comparison with the domain-specific language approaches Hipacc and Halide-HLS. We consider two representative applications from the Halide-HLS repository with different configurations (border handling mode and vectorization factor): Gauss and Harris. These DSLs have been developed by FPGA experts and perform better than many other existing libraries. The applications are rewritten for Hipacc and AnyHLS by respecting their original descriptions. This ensures that Halide-HLS applications have been implemented with adequate scheduling primitives. Hipacc and AnyHLS implementations require only the algorithm descriptions as input.

For almost all applications and configurations in Tables 6.2 and 6.3, AnyHLS implementations demand fewer resources and deliver higher performance. Of course, this improvement mainly stems from our library implementation. AnyHLS achieves a lower latency mainly because of the following reasons:

i) The latency of a local operator generated using the AnyHLS' image processing approach corresponds to the theoretical latency given in Eq. (6.4), which is $L = L_{arith} + 1.042.442$ clock cycles for Gauss when $v = 1$. $L_{arith} = 14$ for AnyHLS' Gauss implementation as shown in Table 6.2.

ii) Halide-HLS pads input images according to the selected border handling mode (even when no border handling is defined). This increases the input image size from (W, H) to $(W + w - 1, H + h - 1)$, thus the latency.

iii) Hipacc does not pad input images, but executes $(H + \lfloor h/2 \rfloor \cdot (W + \lfloor w/2 \rfloor))$ loop iterations for a $(W \times H)$ image and $(w \times h)$ window. This is similar to the convolution example in the Vivado Design Suite User Guide [Xil17c], but not optimal.

Table 6.2: post place and route results for the Xilinx Zynq board for images of size 1020×1020 and T_{target} = 5 ns (corresponds to f_{target} = 200 MHz). Border handling is undefined.

App	v		#BRAM	#SLICE	#DSP	Latency [cyc.]	Throughput [MB/s]
Gauss	1	AnyHLS	8	463	16	1042456	828.2
		Halide-HLS	8	1823	50	1052673	438.2
		Hipacc	8	473	16	1044500	764.7
	4	AnyHLS	16	1441	80	260626	3041.4
		Halide-HLS	16	4112	180	266241	1640.1
		Hipacc	16	1519	64	261649	3064.6
Harris	1	AnyHLS	20	1405	22	1041450	829.0
		Halide-HLS	16	2688	35	1052673	464.0
		Hipacc	20	1457	34	1042466	828.2
	2	AnyHLS	20	2513	44	520740	1450.4
		Halide-HLS	16	4011	70	528385	895.0
		Hipacc	20	2326	68	521756	1637.8

Table 6.3: post place and route results for the Gaussian blur with clamping at the borders. Image sizes are 1024×1024, $v = 1$, f_{target} = 200 MHz.

Framework	#BRAM	#SLICE	#DSP	Latency [cyc.]	Throughput [MB/s]
AnyHLS	8	1646	16	1050641	801.8
Halide-HLS	16	2096	50	1060897	458.7
Hipacc	8	1709	16	1052693	820.1

The execution time of an implementation equals $T_{clk} \cdot latency$, where T_{clk} $(1/f_{clk})$ is the period of the maximum achievable clock frequency (lower is better). Overall, AnyHLS processes a given image faster than the other DSL implementations.

Halide-HLS uses more on-chip memory for line buffers (see Section 6.4.3) compared to Hipacc and AnyHLS because of its image padding for border handling. Let us consider the number of BRAMs utilized for the Gaussian blur: The line buffers need to hold 4 image lines for the 5×5 kernel. The image width is 1024 and the pixel size is 32 bits. Therefore, AnyHLS and Hipacc use eight 18K BRAMs as shown in Table 6.2. However, Halide-HLS stores 1028 integer pixels, which require 16 18K BRAMs to buffer four image lines. This doubles the number of BRAMs usage (see Table 6.3).

AnyHLS use the vectorization architecture proposed in [ÖRH⁺17b]. This improves the use of the registers compared to Hipacc and Halide.

The performance metrics and resource usage reported by Vivado HLS correlate

with our Impala descriptions, hence we claim that the HLS code generated using the AnyHLS' image processing approach does not entail severe side effects for the synthesis using Vivado HLS. Hipacc and Halide-HLS have dedicated compiler backends for HLS code generation. These can be improved to achieve similar performance to AnyHLS. However, this is not a trivial task and prone to errors. The advantage of AnyDSL's partial evaluation is that the user has control over code generation. Extending AnyHLS' image processing library only requires adding new functions in Impala (see Figure 6.2). Our intention to compare AnyHLS with these DSLs is to show that we can generate equally good designs without creating an entire compiler backend.

Experiments using Intel FPGA SDK for OpenCL (AOCL)

Table 6.4 presents the implementation results for an edge detection algorithm provided as a design example by Intel. The algorithms consist of RGB to Luma color conversion, Sobel filters, and thresholding. Intel's implementations consist of a single-work item kernel that utilizes shift registers according to the FPGA design paradigm. These types of techniques are recommended by Intel's optimization guide [Int17] despite that the same OpenCL code performs drastically bad on other computing platforms.

Table 6.4: post place and route results of an edge detection application for the Intel Cyclone V. Image sizes are 1024×1024. None of the implementations use DSPs.

v	Framework	#M10K	#ALM	#DSP	Throughput [MB/s]
1	Intel's Imp.	290	23830	0	419.5
	AnyHLS	291	23797	0	422.5
	Hipacc	318	25258	0	449.1
16	Intel's Imp.	-	-	0	-
	AnyHLS	337	29126	0	1278.3
	Hipacc	362	35079	0	1327.7
32	Intel's Imp.	-	-	0	-
	AnyHLS	401	38069	0	1303.8
	Hipacc	421	44059	0	1320.0

We described Intel's handwritten *SobelLuma* example using Hipacc and AnyHLS. Both Hipacc and AnyHLS provide a higher throughput even without vectorization. In order to reach memory-bound, we would have to rewrite Intel's hand-tuned design

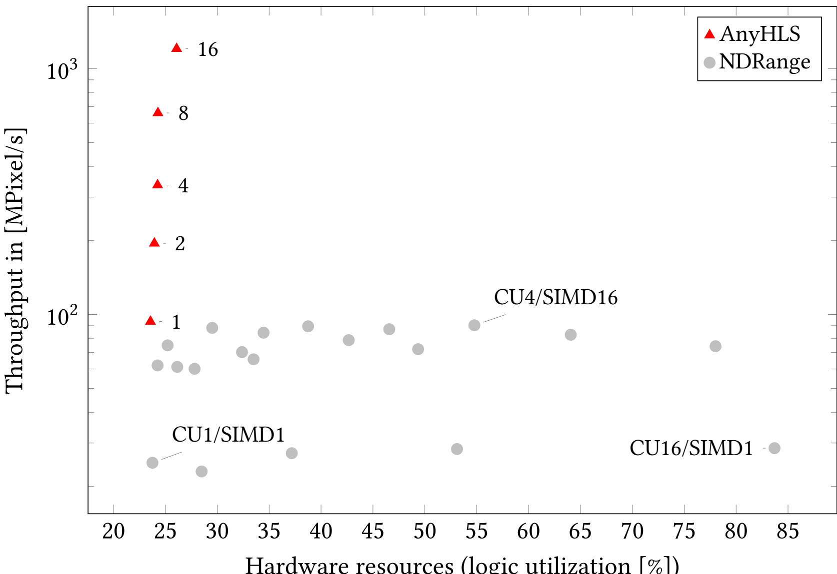


Figure 6.11: Design space for a 5×5 mean filter using an NDRange kernel (using the `num_compute_units` / `num_simd_work_items` attributes) and AnyHLS (using the vectorization factor v) for an Intel Cyclone V FPGA. (Figure reprinted from [ÖPM⁺20a], © 2020 IEEE)

example to exploit further parallelism. AnyHLS uses slightly less resources, whereas Hipacc provides slightly higher throughput for all the vectorization factors. Similar to Figure 6.10, both frameworks yield throughputs very close to the memory bound of the Intel Cyclone V.

The OpenCL NDRange kernel paradigm conveys multiple concurrent threads for data-level parallelism. OpenCL-based HLS tools exploit this paradigm to synthesize hardware. AOCL provides attributes for NDRange kernels to transform their iteration space. The `num_compute_units` attribute replicates the kernel logic, whereas `num_simd_work_items` vectorizes the kernel implementation[5]. Combinations of those provide a vast design space for the same NDRange kernel. However, as Figure 6.11 demonstrates, AnyHLS achieves implementations that are orders of magnitude faster than using attributes in AOCL.

Finally, Table 6.5 and Figure 6.12 present a comparison between AnyHLS and the AOCL backend of Hipacc [ÖRH⁺16]. As shown in Figure 6.2, Hipacc has an individual backend and template library written with preprocessor directives to generate high-performance OpenCL code for FPGAs. In contrast, the application and library code in AnyHLS stays the same. The generated AOCL code consists of a loop that iterates over the input image. Compared to Hipacc, AnyHLS achieves a

[5]These parallelization attributes are suggested in [Int17] for NDRange kernels, not for the single-work item kernels using shift registers such as the edge detection application shown in Table 6.4.

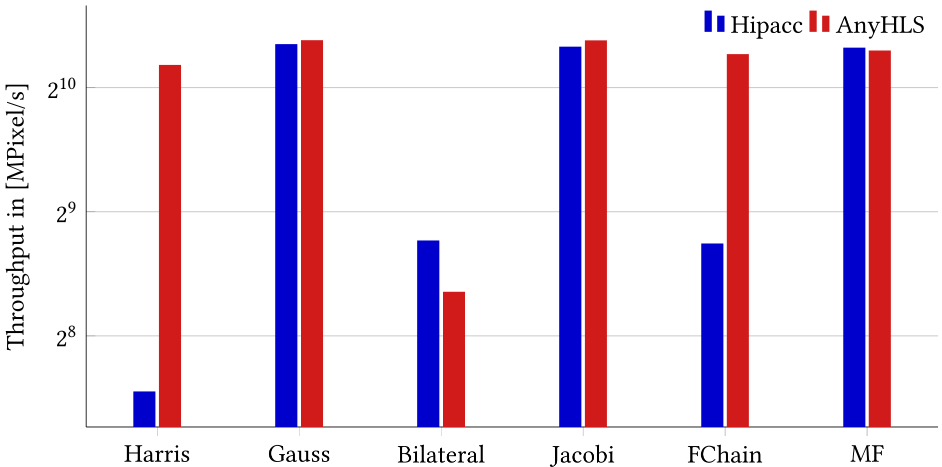


Figure 6.12: Throughput measurements for an Intel Cyclone V for the implementations generated from AnyHLS and Hipacc. Resource utilization for the same implementations are shown in Table 6.5. (Figure reprinted from [ÖPM+20a], © 2020 IEEE)

Table 6.5: PPnR for the Intel Cyclone V. Missing numbers (-) indicate that the generated implementations do not fit the board.

App	v	Framework	#M10K	#ALM	#DSP	Throughput [MB/s]
Gauss	16	AnyHLS	401	37509	0	1330.1
	16	Hipacc	402	35090	0	1301.2
Jacobi	16	AnyHLS	370	31446	0	1328.8
	16	Hipacc	372	30296	0	1282.9
Bilat.	1	AnyHLS	399	79270	153	326.6
	1	Hipacc	422	79892	159	434.7
MF	16	AnyHLS	400	39266	0	1255.68
	16	Hipacc	-	-	-	-
	8	Hipacc	351	31796	0	1275.9
FChain	8	AnyHLS	418	44807	0	1230.6
	8	Hipacc	645	64225	0	427.4

similar performance in terms of throughput. This shows that AnyHLS optimizes the inter-kernel dependencies better than Hipacc (see Section 6.4.2).

6.6 Conclusions

In this chapter, we advocate the use of modern compiler technologies for high-level synthesis. We combine functional abstractions with the power of partial evaluation to decouple a high-level algorithm description from its hardware design that implements the algorithm. This process is entirely driven by code refinement, generating input code to HLS tools, such as Vivado HLS and AOCL, from the same code base. To specify important abstractions for hardware design, we have introduced a set of basic primitives. Library developers can rely on these primitives to create domain-specific libraries. As an example, we have implemented an image processing library for synthesis to both Intel and Xilinx FPGAs. Finally, we have shown that our results are on par or even better in performance compared to state-of-the-art approaches.

7 Conclusion and Future Directions

The trend in computer architecture is moving towards specialization of hardware (for an application or a class of applications) and software (for target hardware) to overcome the limitations posed by the power constraints and the scaling of current semiconductor technology (see Section 1.2) [HP19; Hor14; CCF+10; LTE+20]. Modern processor dies are being built with a GPU and a number of specialized accelerators designed for, e.g., image, audio, and video processing. FPGAs are commercially being used in data centers to accelerate various tasks, e.g., by Microsoft [PCC+14; ORK+15], or Baidu [OLQ+14]. Amazon rents various FPGAs in the cloud as part of their AWS service [Ama22]. Programming such heterogeneous systems is challenging; programmers must write device-specific code to achieve high performance and efficiency. Such optimized code is usually not portable, and rewriting programs for each target is tedious. Improving compilers to automatically parallelize, optimize, and generate target-specific code from an application description written in a programming language initially developed for a sequential (or multi-threaded) execution of older computer architectures has been proven to be infertile [CCF+10]. The problem gets even more complex for designing custom hardware specialized to a set of tasks.

Designing application-specific hardware removes the inefficiencies caused by unnecessary logic, complex control flow, and a fixed memory system (see Section 1.2.3), thus providing better energy efficiency. However, the same customization abilities, which can facilitate orders-of-magnitude performance/watt gains, make hardware design (for an FPGA or ASIC) a complex task that exposes low-level concerns. This complexity poses a severe problem to the widespread adoption of FPGAs (despite being easily accessible from a server) and increases the non-recurring engineering (NRE) costs of ASIC design. Furthermore, the gap between algorithm development and hardware design concerns prevents exploring algorithms that are optimized for custom hardware design. That is, exploring/optimizing algorithms for FPGA/ASIC requires hardware design knowledge, whereas modifying algorithms for a more efficient hardware implementation requires algorithm knowledge.

HLS, an NP-complete problem, has taken much attention over the last five decades as a remedy to the issues mentioned above. The latest generation of tools shows promising results, especially for DSP and data-path-oriented applications, and of-

fer system integration tools to enable a software-like development experience by automatically synthesizing control interface and driver support. Nevertheless, HLS is still not embraced by the community (not by algorithm/software developers as well as hardware designers), and neither has it become an integral part of hardware design. Modern HLS tools require hardware design knowledge to achieve good results. They offer a language that forces users to describe hardware implementation using a mix of software and hardware design abstractions. The optimizations are hard to understand and tool-specific. Users must think about hardware-centric optimizations while writing software that looks like a program developed for an ISA. This burdens HLS users and compilers, where the compiler has to extract a control path and data path from an ambiguous description, e.g., spoiled with CPU-centric style and hardware-specific compiler pragmas.

7.1 Summary

We have argued that the next step in HLS requires a clear separation of algorithmic concerns from its implementation. We advocate that the design flow in HLS must start from a purely behavioral description of an application free from hardware design concerns. At the same time, it should allow hardware designers to describe their implementation concisely using hardware design abstractions (but above RT-level and in a productive way). Developing such an HLS tool is a challenging problem when a general-purpose language is used for a design entry. However, we claim that it is achievable when the algorithm description is constrained to a class of applications, and domain knowledge is used to generate a concise description of the behavior of an application-specific hardware as input to HLS. In this way, application developers do not require hardware design knowledge to achieve good HLS results.

In this thesis, we used image processing as the domain of interest to test the benefits of our approach and provide proof-of-concept implementations. In order to leverage decades of research on HLS and thus use state-of-the techniques developed for the tasks of hardware synthesis (e.g., scheduling, allocation, binding), we used two commercially available tools, namely Xilinx Vivado HLS and Intel OpenCL SDK for FPGAs. Our experiments show that these tools are able to synthesize *highly optimized* hardware circuits that can rival with RTL implementations when the input code describes an application-specific hardware concisely. That is, our application is written in C++ or OpenCL but by using arrays and registers to describe a memory hierarchy, loops to describe a pipelined hardware schedule. We also show that writing NDRange code in OpenCL provides poor HLS results, where the NDRange paradigm is used for the multi-threaded and multicore execution. Hence, despite the different language syntaxes (OpenCL or C++), both of the HLS tools perform best for similar input descriptions, where the main program uses one top-level loop and a single thread.

Our approach requires three main tasks for a framework developer: (*i*) identifying performance-relevant and algorithmic-level (high-level) abstractions by investigating the considered application domain, (*ii*) investigating implementation techniques for the considered applications, and (*iii*) using code-generation techniques to provide a productive, modular, and portable description for the application developers in the form of a DSL or a function library.

We built our solutions by using high-level abstractions based on point, local, and global operators. In Chapter 3, we proposed novel implementation techniques for implementing image border handling and accelerating stencil-based image processing applications using a technique called *loop coarsening*. These techniques require modifying memory hierarchy, data path, and control path of a synthesized hardware, and thus are hard to be inferred by HLS compilers. We showed order of magnitude improvements in throughput and up to 80% less resource usage compared to compiler-based automatic parallelization techniques of a modern HLS tool.

We then presented three distinct solutions based on different code generation techniques to raise the abstraction level in HLS and to hide implementation details from application developers:

(i) A novel source-to-source compiler (based on Clang/LLVM compiler infrastructure) that generates input code for Intel's HLS tool (OpenCL SDK for FPGAs) from the Hipacc image processing DSL [MRH+16] (see Chapter 4). Our backend applies various transformations using domain knowledge, such as creating a streaming pipeline and parallelizing image processing operators using our implementation techniques introduced in Chapter 3. We demonstrated that implementations produced by our compiler backend are on par with the hand-written applications provided by Intel (when not parallelized) and significantly better when compared with the parallelization intrinsics provided by the Intel HLS compiler. Since Hipacc is able to generate high-performance code for CPUs, and GPUs as well, we show that our approach allows using the same application description to target drastically different computing platforms.

(ii) Using the C++ metaprogramming language to build a modular and highly parameterizable function library for describing image processing applications as stream-based data flow graphs of point, local, and global operators (see Chapter 4). It is highly optimized with hardware design techniques such as bit-level optimizations, deep pipelining, and our novel parallelization techniques introduced in Chapter 3. Our library increases the productivity of HLS users by providing high-level abstractions and key hardware design elements to describe their hardware in a modular way. Moreover, we alleviated the tasks of Hipacc's HLS backends by integrating this library as part of its code generation flow.

(iii) Introducing AnyHLS to advocate the benefits of using partial evaluation to

raise the abstraction level in HLS (see Chapter 6). AnyHLS allows designing and extending a library of domain-specific abstractions for C-based HLS as easy as writing higher-order functions in a functional language. Our approach is significantly more effortless than developing a source-to-source compiler and guarantees the well-typedness of a generated program (unlike C++ metaprogramming). We showed the productivity, modularity, and portability gains by presenting an image processing library as a case study.

In addition, we presented the HipaccVX framework to show the benefits of using code generation techniques and domain-specific abstractions used in this thesis to implement OpenVX (see Chapter 5). OpenVX is an industrial standard built for achieving performance-portability across different computing platforms. It allows for applying system-level optimizations between standard computer vision functions (e.g., Gaussian, Harris corner) by offering a graph-based API where edges show dependency between data and computation. This graph-based description of CV functions is partially similar to Hipacc. However, the OpenVX standard does not provide a protocol to accelerate user-defined kernels as part of an application graph. Thus, it restricts users from describing their algorithms using a small set of computer vision functions. As a remedy, HipaccVX extends the OpenVX standard to use Hipacc's domain-specific abstractions in a user-defined node to support writing custom applications. Our approach reduced code repetition (e.g., replicated implementation of convolution functions for Gaussian and Sobel algorithms) and enabled further optimizations while allowing users to describe their algorithm in a standard language.

We envision a sea of domain-specific languages and application libraries built to raise the abstraction level in HLS (ideally, one DSL or a library for one application domain). Choosing the right software and hardware abstractions, as well as increasing productivity of designing and extending these libraries while guaranteeing the correctness of a generated program are extremely important for realizing this vision. Industrial practice to hide implementation details from users is to use metaprogramming tools of a standard language to provide function libraries tailored for one specific HLS tool. While this has the advantage of using a well-known language, programmers suffer from using a metalanguage that optimizes the actual program as a data structure (e.g., C++ templates and C++ core are different languages), and entails writing programs that are hard to understand. Developing a DSL and a source-to-source compiler provides more flexibility to compiler developers where the same application code can be used for targeting different computing platforms (such as CPUs, GPUs, and FPGAs), and the correctness of a generated program can be guaranteed. However, this is a complex task that is time-consuming and error-prone, thus not extension friendly. We showed that partial evaluation is a powerful technique that significantly increases developers' productivity, especially when used for specializing inputs of higher-order functions. Partial evaluation was not offered in a

standard programming language when this thesis was written[1]. Hence, we believe our contributions play a key role in convincing HLS tool developers.

This thesis shows that domain-specific solutions are able to bridge the gap between algorithm developers and hardware designers and, even more, provide portability across different computing platforms without sacrificing performance and efficiency (e.g., of power and resource usage). These solutions restrict users to a specific application domain and expect them to learn a new language (or a library). The former limitation can even play a good role by saving algorithm developers (with no digital circuit design expertise) from using FPGAs when they are not advantageous over cheaper and more flexible ISAs. In other words, restricting input language by offering a set of domain-specific high-level abstractions (that yield performance, power, or energy gains when implemented for the target platform) restricts the way input algorithms can be described. This restriction provides a safe space for algorithm developers to explore variations of an input application without worrying about making an implementation drastically worse for the target platform. We showed (by HipaccVX framework) that the latter limitation (i.e., the requirement of learning a new language) can be circumvented by implementing industrial standard languages using domain-specific solutions.

7.2 Future Work

Many improvements could be made to increase the productivity of mapping algorithms to specialized circuits and heterogeneous systems in general. More specifically, extending our solutions with additional application domains, hardware-centric optimization techniques, and also for other FPGA-based systems (i.e., FPGAs with HBM memory) as well as for other HLS tools would be beneficial. Ultimately, our approach helps profiling and mapping algorithms to heterogeneous systems when used as part of design space exploration and partitioning algorithms of a system-level design approach.

We argue that HLS tools should provide a language above RTL but allow for concise descriptions of a hardware implementation using hardware abstractions. Designing such a language is challenging and requires investigating several application domains (in addition to image processing) to explore a correct set of hardware abstractions. For this purpose, supporting language features that allow for a productive, modular, extensible, and portable way of describing high-level abstractions (such as partial evaluation, and zero-cost higher-order functions) is crucial. AnyHLS provides a good base for this work since it can be extended easily and already supports two

[1] We used Impala, a language forked from Rust, to implement a state-of-the-art partial evaluation technique as a core compiler technology [LBH+18]. However, Impala misses several fundamental programming features, such as polymorphism.

commercial HLS tools (same code, different HLS tools). Developing libraries for other application domains, such as machine learning, security, linear algebra, filters, and frequency domain applications in digital signal processing, would be very beneficial. Similarly, Hipacc can be extended for 3D-stencil applications, computer vision, and a significant portion of computer graphics applications (especially the pre-and post-processing parts).

Another important direction is integrating additional hardware-centric optimizations to our frameworks, which would be hidden from application developers. We have shown in a recent publication [SÖK+22a] that the efficiency of DSP utilization can be increased in a generic way for low-precision arithmetic operations by packing multiple operations into single DSP units. Similarly, automatically applying multiplierless multiple constant multiplication (MCM) algorithms (such as [VP07]) from a high-level application reduces the number of resources required for implementing a set of constant multiplications. At best, implementing a design space exploration algorithm for mapping operations to DSP or logic resources (such as LUTs) by also considering the techniques mentioned above would increase the quality of synthesized hardware in many application domains. Applying these techniques before HLS allows for using domain knowledge and does not burden HLS compilers; thus, they are good optimization techniques showing the benefits of our approach.

In 2021, Xilinx open-sourced the front end of the Vitis HLS tool. Modifying Hipacc and AnyHLS to directly emit LLVM code instead of generating C++ code as input to its front end could help decrease the unwanted effects of syntactic variations. Furthermore, a clean-slate HLS language (e.g., originated from AnyHLS) could be developed as a final product. Extending Hipacc and AnyHLS for other open-source HLS tools, even better, providing a standard way of using these HLS tools would significantly benefit the community for cumulative progress.

One exciting and promising research direction is to use DSL-based code generation techniques as part of a system-level design methodology. Hipacc is able to generate highly optimized code for CPUs, GPUs, and FPGAs. Similarly, we used AnyHLS to develop an FPGA branch for Stincilla DSL[2], which was formerly able to target CPUs and GPUs. Developing a top-level ESL algorithm that leverages domain knowledge to partition an input algorithm into a heterogeneous system automatically and generates code using Hipacc or Stincilla would be very valuable.

[2] FPGA branch of https://github.com/AnyDSL/stincilla

German Part

Deklarative Programmierungstechniken zur Hardware-Synthese von Bildverarbeitungsanwendungen

Zusammenfassung

Herkömmliche Hardwarebeschreibungssprachen (HDLs) wie VHDL und Verilog werden häufig für den Entwurf digitaler elektronischer Schaltungen, z. B. anwendungsspezifischer integrierter Schaltungen (ASICs), oder für die Programmierung feldprogrammierbarer Gate-Arrays (FPGAs) verwendet. Die Verwendung von HDLs für die Implementierung komplexer Algorithmen oder die Aufrechterhaltung großer Projekte ist jedoch selbst für Experten mühsam und zeitaufwändig. Dies verhindert auch den weit verbreiteten Einsatz von FPGAs. Als Lösung wird seit Jahrzehnten die High-Level-Synthese (HLS) erforscht, um die Produktivität zu erhöhen, indem letztlich eine Verhaltensbeschreibung eines Algorithmus (was macht die Schaltung?) als Design-Eingabe genommen und automatisch eine Register-Transfer-Level (RTL)-Implementierung erzeugt wird. Kommerzielle HLS-Tools gehen von bekannten Programmiersprachen aus (z. B. C, C++ oder OpenCL), die ursprünglich für programmierbare Prozessoren mit einer Befehlssatzarchitektur (ISA) entwickelt wurden. Diese Werkzeuge liefern jedoch nur dann eine zufriedenstellende Qualität der Hardwaresynthese-Ergebnisse, wenn die Programmierer für ihre Anwendungen hardwarefreundliche Implementierungen beschreiben (wie die Schaltung aufgebaut ist?), die z. B. eine bestimmte Speicherarchitektur, einen Steuerpfad und einen Datenpfad nutzen. Dies erfordert ein tiefgehendes Verständnis der Prinzipien des Hardware-Designs. Um Software-Programmiersprachen für den Hardware-Entwurf zu übernehmen, verwendet jedes HLS-Tool seinen eigenen Sprachdialekt und führt einen nicht standardisierten Satz von Pragmas ein. Die gemischte Verwendung von Software- und Hardwaresprachabstraktionen behindert ein rein verhaltensorientiertes Design und macht Optimierungen schwer verständlich, da der erwartete Code weder eine reine Hardwarebeschreibung noch eine reguläre Softwareimplementierung ist. Darüber hinaus muss ein Code, der für ein HLS-Tool optimiert wurde, erheblich geändert werden, um für ein anderes HLS-Tool geeignet zu sein, und ist auf einer ISA schlecht zu handhaben. Unserer Überzeugung nach, wird der nächste Entwicklungsschritt bei HLS auf sprachlicher Seite liegen, um Produktivitäts-, Portabilitäts- und Leistungshürden zu überwinden, die durch Unzulänglichkeiten des Verhaltensdesigns bestehender Werkzeuge verursacht werden.

In dieser Dissertation werden drei verschiedene Lösungen vorgestellt und evaluiert,

um die Beschreibung des Verhaltens (was?) eines Algorithmus von seiner Implementierung (wie?) zu trennen und gleichzeitig qualitativ hochwertige Hardwaresyntheseergebnisse für die Klasse der Bildverarbeitungsanwendungen zu liefern. Dies wird durch die Generierung von einem hochoptimiertem, zielspezifischem Eingabecode für kommerzielle HLS-Tools aus High-Level-Abstraktionen erreicht, die Parallelität, Lokalität und Speicherzugriffsinformationen einer Eingabeanwendung erfassen. Bei diesen Ansätzen wird eine Bildverarbeitungsanwendung als eine Reihe grundlegender Bausteine beschrieben, nämlich Punkt-, lokale und globale Operatoren, ohne Berücksichtigung der Implementierung auf niedriger Ebene. Anschließend wird optimierter Eingabecode für das gewählte HLS-Tool (Vivado HLS oder Intel OpenCL SDK for FPGAs) unter Verwendung einer der folgenden unterschiedlichen Programmiertechniken generiert: (i) ein Source-to-Source-Compiler, der für eine domänenspezifische Sprache (DSL) der Bildverarbeitung entwickelt wurde, oder (ii) Template-Metaprogrammierung zur Spezialisierung von C++-Eingabeprogrammen zur Übersetzungszeit, (iii) eine partielle Evaluierungstechnik zur Spezialisierung von Funktionen höherer Ordnung.

Wir stellen ersten den Source-to-Source-Compiler vor, der optimierten Eingabecode für Intel OpenCL SDK für FPGAs aus einer DSL generiert. Wir verwenden das Heterogeneous Image Processing Acceleration (Hipacc) framework, bestehend aus einer Bildverarbeitungs-DSL und einen Source-to-Source-Compiler, der ursprünglich für Grafikprozessoren (GPUs) entwickelt wurde. Die Hipacc-DSL bietet High-Level-Abstraktionen für Punkt-, lokale und globale Operatoren in Form von Sprachkonstrukten. Während der Codegenerierung wandelt das Compiler-Frontend den DSL-Eingabecode in eine abstrakte Syntaxbaum-Darstellung (AST) um und nutzt dabei die Clang/LLVM-Compiler-Infrastruktur. Durch die Nutzung des Domänenwissens, das aus dem eingegebenen DSL-Code gewonnen wird, wendet unser Backend mehrere Transformationen an, um eine Beschreibung einer Streaming-Hardware-Pipeline zu generieren. Im letzten Schritt generiert Hipacc OpenCL-Code als Eingabe für den HLS-Compiler von Intel. Die Qualität unserer Hardware-Synthese-Ergebnisse kann mit denen von Intels handoptimierten OpenCL-Code-Beispielen mithalten, was den Durchsatz und die Ressourcennutzung angeht. Darüber hinaus erreicht Hipaccs Code-Generierung einen signifikant höheren Durchsatz und verbraucht weniger Ressourcen im Vergleich zu Intels Parallelisierungs-Intrinsic.

Zweitens stellen wir einen auf Template-Metaprogrammierung basierenden Ansatz zur Entwicklung modularer und hochgradig parametrisierbarer Funktionsbibliotheken vor, die bei der Kompilierung mit HLS-Tools auch qualitativ hochwertige Hardware-Syntheseergebnisse liefern. Bei diesem Ansatz besteht die Anwendungsprogrammierschnittstelle (API) der Bibliothek aus generischen High-Level-Funktionen zur Deklaration von Bausteinen von Bildverarbeitungsanwendungen, z. B. Punkt-, lokale und globale Operatoren, im Gegensatz zu typischen Bibliotheken, die Funktionen für komplette Algorithmen anbieten, wie z. B. OpenCV. Die Bibliothek ist mit

den Best Practices von Vivado HLS sowie hardware-zentrierten Entwurfstechniken wie Deep Pipelining, Coarse-Level-Parallelisierung und Bit-Level-Optimierungen optimiert. Die Bibliothek enthält mehr als ein Vorlagendesign für jede algorithmische Instanz, um für die Eingabeparameter optimierte Implementierungen nutzen zu können. Sie enthält beispielsweise mehrere Implementierungen für die Behandlung von Bildrändern und Parallelisierungsstrategien auf höherer Ebene, die für verschiedene Eingabeparameter einer lokalen Operatorspezifikation in Betracht gezogen werden. Darüber hinaus wird ein Auswahlalgorithmus zur Kompilierzeit vorgeschlagen, um die am besten geeignete Implementierung anhand eines analytischen Modells für Ressourcennutzung, Geschwindigkeit und Latenzzeit auszuwählen. Auf diese Weise werden Implementierungsdetails auf niedriger Ebene vor den Benutzern verborgen.

Zusätzlich zu den vorgestellten Vorteilen der Verwendung von High-Level-Abstraktionen zur Erhöhung der Abstraktionsebene in HLS zeigen wir, dass dieser Ansatz für die Portabilität der Leistung über verschiedene Computerplattformen von Vorteil ist. Ähnlich wie bei FPGAs kann die Leistungsfähigkeit von CPUs und GPUs nur dann voll ausgeschöpft werden, wenn Anwendungsprogramme mit architekturspezifischen Optimierungen auf niedriger Ebene abgestimmt werden. Diese Optimierungen beruhen auf grundlegend unterschiedlichen Programmierparadigmen und -sprachen. Als Lösung hat Khronos OpenVX als ersten Industriestandard für die graphbasierte Spezifikation von Computer-Vision-Anwendungen (CV) veröffentlicht. Die graphbasierte Spezifikation ermöglicht die Optimierung von Speicherübertragungen zwischen verschiedenen CV-Funktionen von einem gerätespezifischen Backend aus. Außerdem verbirgt der Standard Implementierungsdetails auf niedriger Ebene vor der Algorithmusbeschreibung. So sind beispielsweise die Speicherhierarchie und die Gerätesynchronisation für den Benutzer nicht sichtbar. Der OpenVX-Standard unterstützt jedoch nur eine kleine Anzahl von Computer-Vision-Funktionen und bietet keinen Mechanismus, um Anwendercode als Teil eines OpenVX-Graphen einzubinden. Als nächster Schritt wird HipaccVX als eine OpenVX-Implementierung und -Erweiterung vorgestellt, die die Codegenerierung für eine Vielzahl von Computerplattformen unterstützt. HipaccVX nutzt die Standard-API und Graphen-Spezifikation von OpenVX und bietet gleichzeitig neue Sprachkonstrukte zur Beschreibung von Algorithmen unter Verwendung von High-Level-Abstraktionen, die sich an bestimmte Speicherzugriffsmuster halten (z.B. lokale Operatoren). Somit unterstützt es die Beschleunigung von benutzerdefiniertem Code sowie die CV-Funktionen von OpenVX. Auf diese Weise kombiniert HipaccVX die Vorteile von DSL-Designtechniken mit einer industriellen Standardspezifikation.

Schließlich wird AnyHLS vorgestellt, ein neuartiger Ansatz zur Erhöhung der Abstraktionsebene in HLS durch die Verwendung partieller Evaluierung als zentrale Compilertechnologie. Lediglich eine Sprache und eine Funktionsbibliothek werden verwendet, um einen zielspezifischen Eingabecode für zwei kommerzielle HLS-Tools zu erzeugen, nämlich Xilinx Vivado HLS und Intel FPGA SDK for OpenCL.

Hardware-zentrierte Optimierungen, die Code-Transformationen erfordern, werden als Funktionen höherer Ordnung implementiert, ohne werkzeugspezifische Pragma-Erweiterungen zu verwenden. Die Erweiterung von AnyHLS um neue Funktionen erfordert keine Änderungen an einem Compiler oder einem Codegenerator, der in einer anderen (Host-) Sprache geschrieben ist. Im Gegensatz zur Metaprogrammierung ist die Wohltypisierung eines Restprogramms garantiert. Infolgedessen wird eine wesentlich höhere Produktivität als bei den bestehenden Techniken und ein noch nie dagewesenes Maß an Portabilität zwischen verschiedenen HLS-Tools erreicht. Die Produktivitäts-, Modularitäts- und Portabilitätsgewinne werden anhand einer Bildverarbeitungsbibliothek als Fallstudie demonstriert.

Bibliography

[AB17] Ben Ashbaugh and Ariel Bernal. OpenCL interoperability with OpenVX graphs. In *Proceedings of the 5th International Workshop on OpenCL*, page 26. ACM, 2017.

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, 2001.

[Ama22] Amazon. Amazon EC2 F1 Instances, 2022. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, Atlantic City, New Jersey. Association for Computing Machinery, 1967. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

[Bai11a] Donald G Bailey. *Design for embedded image processing on FPGAs.* John Wiley & Sons, 2011.

[Bai11b] Donald G Bailey. Image border management for FPGA based filters. In *Proceedings of the Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)*, pages 144–149. IEEE, 2011.

[Ban08] Isaac Bankman. *Handbook of medical image processing and analysis.* Elsevier, 2008.

[BC11] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[BCC18] BCC Research. Global Markets for Machine Vision Technologies. Technical report, September 2018.

[BHS09] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[BKL+93] Jörg Biesenack, Michael Koster, Anton Langmaier, Stephane Ledeux, S Marz, Michael Payer, Michael Pilsl, Steffen Rumler, Holger Soukup, Norbert Wehn, et al. The Siemens high-level synthesis system CALLAS. *Transactions on very large scale integration (VLSI) systems*, 1(3):244–253, 1993.

[BOS+95] Reinaldo A Bergamaschi, Richard A O'Connor, Leon Stok, Michael Z Moricz, Shiv Prakash, Andreas Kuehlmann, and D Sreenivasa Rao. High-level synthesis in an industrial environment. *IBM Journal of Research and development*, 39(1.2):131–148, 1995.

[Bou01] Jean-Yves Bouguet. Pyramidal implementation of the affine Lucas Kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[BRR+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA), pages 193–205. IEEE, February 16–20, 2019. DOI: 10.1109/CGO.2019.8661197.

[BRS13] David F Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013. DOI: 10.1145/2436696.2443836.

[BVR+12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation (DAC)* (San Francisco, CA, USA), pages 1212–1221. IEEE, June 3–7, 2012. DOI: 10.1145/2228360.2228584.

[CAD+] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)* (Oslo, Norway), pages 531–534. DOI: 10.1109/FPL.2012.6339272.

[Cam88] Raul Camposano. Design process model in the Yorktown silicon compiler. In *Design Automation Conference*, pages 489–494. IEEE, 1988.

[CCF+10] Calin Cascaval, Siddhartha Chatterjee, Hubertus Franke, Kevin J Gildea, and Pratap Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5–1, 2010.

[CCF+16] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *Proceedings of the 53rd Annual Design Automation (DAC)* (Austin, TX, USA), 109:1–109:6. ACM, June 5–9, 2016. DOI: 10.1145/2897937.2897972.

[CCW+20] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. When HLS meets FPGA HBM: Benchmarking and bandwidth optimization. *arXiv preprint arXiv:2010.06075*, 2020.

[CE14] Sheng Chen and Martin Erwig. Early detection of type errors in C++ templates. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 133–144, 2014.

[CGM+09] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[CLN+11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, 2011. DOI: 10.1109/TCAD.2011.2110592.

[Con88] Charles Consel. New insights into partial evaluation: The SCHISM experiment. In *Proceedings of the 2nd European Symposium on Programming (ESOP)* (Nancy, France), pages 236–246. Springer, March 21, 1988. DOI: 10.1007/3-540-19027-9_16.

[CPR+92] Anantha P Chandrakasan, Miodrag Potkonjak, Jan Rabaey, and Robert W Brodersen. HYPER-LP: A system for power minimization using architectural transformations. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, page 300, 1992.

[CVP+16] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the International on Parallel Architecture and Compilation Techniques (PACT)* (Haifa, Israel), pages 327–338. ACM, September 11–15, 2016. DOI: 10.1145/2967938.2967969.

[CWY+17] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. Bandwidth optimization through on-chip memory restructuring for HLS. In *Proceedings of the 54th Annual Design Automation (DAC)* (Austin, TX, USA), 43:1–43:6. ACM, June 18–22, 2017. DOI: 10.1145/3061639.3062208.

[DBA+18] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco D. Santambrogio. A unified backend for targeting FPGAs from DSLs. In *Proceedings of the 29th Annual IEEE International on Application-specific Systems, Architectures and Processors (ASAP)* (Milan, Italy), pages 41–48. IEEE, July 10–12, 2018. DOI: 10.1109/ASAP.2018.8445108.

[DBB+08] William J Dally, James Balfour, David Black-Shaffer, James Chen, R Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *Computer*, 41(7):27–32, 2008.

[Dec04] Jan Decaluwe. MyHDL: A Python-based hardware description language. *Linux Journal*, (127):84–87, 2004.

[DGY+74] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[DKM+90] Giovanni De Micheli, David Ku, Frédéric Mailhot, and Thomas Truong. The Olympus synthesis system. *Design & Test of Computers*, 7(5):37–53, 1990.

[dMH18] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *The Computing Research Repository (CoRR)*, 2018. arXiv: 1805.08288 [cs.DC].

[DRS+86] Hugo De Man, Jan Rabaey, Paul Six, and Luc Claesen. Cathedral-II: A silicon compiler for digital signal processing. *Design & Test of Computers*, 3(6):13–25, 1986.

[dSBL19] Jeferson Santiago da Silva, François-Raymond Boyer, and JM Pierre Langlois. Module-per-object: a human-driven methodology for C++-based high-level synthesis design. In *27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 218–226. IEEE, 2019.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[Edw06] Stephen A Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers*, 23(5):375–386, 2006. DOI: 10.1109/MDT.2006.134.

[Ell99] John P Elliott. *Understanding behavioral synthesis: A practical guide to high-level design*. Springer Science & Business Media, 1999.

[EYA15] Glenn A. Elliott, Kecheng Yang, and James H. Anderson. Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 273–284. IEEE, 2015. DOI: 10.1109/RTSS.2015.33.

[EZI+19] Haggai Eran, Lior Zeno, Zsolt István, and Mark Silberstein. Design patterns for code reuse in HLS packet processing pipelines. In *27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 208–217. IEEE, 2019.

[Fut82] Yoshihiko Futamura. Parital computation of programs. In *Proceedings of the RIMS Symposia on Software Science and Engineering* (Kyoto, Japan), pages 1–35, 1982. DOI: 10.1007/3-540-11980-9_13.

[GAG+09] Daniel D Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[GCL+13] S. T. Gurumani, H. Cholakkal, Y. Liang, K. Rupnow, and D. Chen. High-level synthesis of multiple dependent CUDA kernels on FPGA. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 305–312, January 22–25, 2013. DOI: 10.1109/ASPDAC.2013.6509613.

[Gel01] Patrick P Gelsinger. Microprocessors for the new millennium: challenges, opportunities, and new frontiers. In *International Solid-State Circuits Conference (ISSCC)*, pages 22–25. IEEE, 2001.

[GKP85] John Granacki, David Knapp, and Alice Parker. The ADAM advanced design automation system: Overview, planner and natural language interface. In *Design Automation Conference*, pages 727–730. IEEE, 1985.

[GPW+16] Suresh V Garimella, Tim Persoons, Justin A Weibel, and Vadim Gektin. Electronics thermal management in information and communications technologies: challenges and future directions. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 7(8):1191–1205, 2016.

[GR94] Daniel D. Gajski and Loganath Ramachandran. Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 11(4):44–54, 1994.

[Han09] Frank Hannig. *Scheduling Techniques for High-Throughput Loop Accelerators.* Dissertation, University of Erlangen-Nuremberg, Germany, August 2009. Verlag Dr. Hut, Munich, Germany.

[HBD+14] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM on Graphics (TOG)*, 33(4):144:1–144:11, 2014. DOI: 10.1145/2601097.2601174.

[HDD+16] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM on Graphics (TOG)*, 35(4):85:1–85:11, 2016. DOI: 10.1145/2897824.2925892.

[Hip22] Hipacc. Hipacc: A domain-specific language and compiler for image processing, November 2022. URL: https://hipacc-lang.org/.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014. DOI: 10.1109/ISSCC.2014.6757323.

[HP19] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[HRD+08] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Reconfigurable Computing: Architectures, Tools and Applications: 4th International Workshop, ARC 2008, London, UK, March 26-28, 2008. Proceedings 4*, pages 287–293. Springer, 2008.

[IDC16] International Data Corporation (IDC). Idc's global datasphere forecast shows continued steady growth in the creation and consumption of data, 2016. URL: https://www.idc.com/getdoc.jsp?containerId=prUS46286020.

[IKK+07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 186–197, 2007.

[Int09] Intel. Intel program license subscription agreement, version Rev. 10/2009, Intel Corporation, October 2009. URL: https://www.intel.com/content/www/us/en/programmable/downloads/software/license/lic-prog_lic.html.

[Int17] Intel. *Intel FPGA SDK for OpenCL: Best Practices Guide*. 2017.

[Int18] Intel. Intel's OpenVX developer guide, May 2018.

[Int23] Intel. Intel fpga design examples, June 2023. URL: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/design-examples-overview.html.

[IRT16] IRTS. International technology roadmap for semiconductors 2.0, 2015 edition, 2016. URL: http://www.itrs2.net/itrs-reports.html.

[JGS93] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[Jon18] Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–167, 2018.

[JPP88] Rajiv Jain, Alice Parker, and Nohbyung Park. Module selection for pipelined synthesis. In *Design Automation Conference*, volume 12 of number 15, pages 542–547. IEEE, 1988.

[KBS+19] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, ARRAY@PLDI 2019, Phoenix, AZ, USA, June 22, 2019.* Pages 35–45, 2019. DOI: 10.1145/3315454.3329957.

[KCG+98] Kayhan Kucukcakar, Chih-Tung Chen, Jie Gong, Wim Philipsen, and Thomas E Tkacik. Matisse: An architectural design tool for commodity ICs. *Design & Test of Computers*, 15(2):22–33, 1998.

[KFP+18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA), pages 296–311. ACM, June 18–22, 2018. DOI: 10.1145/3192366.3192379.

[KMN18] R. Kastner, J. Matai, and S. Neuendorffer. Parallel Programming for FPGAs. *ArXiv e-prints*, May 2018. arXiv: 1805.03648.

[Kna96] David W Knapp. *Behavioral synthesis: Digital system design using the Synopsys Behavioral Compiler*. Prentice-Hall, Inc., 1996.

[KP87] Fadi J Kurdahi and Alice C Parker. REAL: A program for register allocation. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 210–215, 1987.

[KSA15] Krzysztof Kępa, Ritesh Soni, and Peter M. Athanas. Inferring custom architectures from OpenCL. In *Proceedings of the 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 9–16, September 1–4, 2015. DOI: 10.1109/PATMOS.2015.7347581.

[Lan65] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, February 1965. ISSN: 0001-0782. DOI: 10.1145/363744.363749.

[LBH+15] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the International on Generative Programming: Concepts & Experiences (GPCE)* (Pittsburgh, PA, USA), pages 11–20. ACM, October 26–27, 2015. DOI: 10.1145/2814204.2814208.

[LBH+18] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. AnyDSL: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(OOPSLA):119:1–119:30, November 4–9, 2018. DOI: 10.1145/3276489.

[LLQ+19] Yanqiang Liu, Yao Li, Zhengwei Qi, and Haibing Guan. A scala based framework for developing acceleration systems with FPGAs. *Journal of Systems Architecture*, 98:231–242, 2019. DOI: 10.1016/j.sysarc.2019.08.001.

[LTE+20] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), 2020.

[LvMvdW+91] Paul ER Lippens, Jef L van Meerbergen, Albert van der Werf, Wim FJ Verhaegh, BT McSweeney, JO Huisken, and OP McArdle. PHIDEO: A silicon compiler for high speed algorithms. In *Proceedings of the European Conference on Design Automation*, pages 436–441. IEEE, 1991.

[Mar14] Christian Martin. Multicore processors: Challenges, Opportunities, Emerging trends. In *Proceedings of Embedded World Conference*, 2014.

[Mar84] Peter Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *21st Design Automation Conference Proceedings*, pages 587–593. IEEE, 1984.

[MB12] Dejan Marković and Robert W Brodersen. *DSP architecture design essentials*. Springer Science & Business Media, 2012.

[MCC06] R. Mahajan, Chia-pin Chiu, and G. Chrysler. Cooling a microprocessor chip. *Proceedings of the IEEE*, 94(8):1476–1486, 2006. DOI: 10.1109/JPROC.2006.879800.

[MDH+19] Richard Membarth, Hritam Dutta, Frank Hannig, and Jürgen Teich. Efficient Mapping of Streaming Applications for Image Processing on Graphics Cards. In *Transactions on High-Performance Embedded Architectures and Compilers V*. Volume 11225, Lecture Notes in Computer Science (LNCS), pages 1–20. Springer, 2019. ISBN: 978-3-662-58833-8. DOI: 10.1007/978-3-662-58834-5{_}1.

[Mem13] Richard Membarth. *Code generation for GPU accelerators from a domain-specific language for medical imaging*. PhD thesis, Friedrich-Alexander University Erlangen-Nürnberg(FAU), 2013. ISBN: 978-3-8439-1074-3. Verlag Dr. Hut, Munich, Germany.

[MFH+12] Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM on Design Automation of Electronic Systems (TODAES)*, 17(2):15:1–15:33, 2012. DOI: 10.1145/2159542.2159547.

[MHT+12] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proceedingsof the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Shanghai), pages 569–581, New York, NY, USA. IEEE Press, May 25, 2012. ISBN: 978-1-4673-0975-2. DOI: 10.1109/IPDPS.2012.59.

[Mic94] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[MRH+16] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipacc: A domain-specific language and compiler for image processing. *on Parallel and Distributed Systems (TPDS)*, 27(1):210–224, 2016. DOI: 10.1109/TPDS.2015.2394802.

[MS09] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009. DOI: 10.1109/MDT.2009.83.

[MSD17] Paul-Jules Micolet, Aaron Smith, and Christophe Dubach. A study of dynamic phase adaptation using a dynamic multicore processor. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the International on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 429–443. ACM, 2015. DOI: 10.1145/2694344.2694364.

[MWS+16] Jones Yudi Mori, André Werner, Arij Shallufa, Florian Fricke, and Michael Hübner. A design methodology for the next generation real-time vision processors. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*, pages 14–25. Springer, 2016. DOI: 10.1007/978-3-319-30481-6_2.

[NPP+20] Mostafa W Numan, Braden J Phillips, Gavin S Puddy, and Katrina Falkner. Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains. *IEEE Access*, 8:174692–174722, 2020.

[NSP+15] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015. DOI: 10.1109/TCAD.2015.2513673.

[NVI20] NVIDIA. Visionworks, version 1.6, September 2020. URL: https://developer.nvidia.com/embedded/visionworks-archive.

[OBD+11] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from OpenCL programs. In *Proceedings of the 19th Field-Programmable Custom Computing Machines (FCCM)*, pages 186–193. IEEE, 2011.

[OIL18] Hossein Omidian, Nick Ivanov, and Guy G. F. Lemieux. An accelerated OpenVX overlay for pure software programmers. In *Proceedings of the International on Field Programmable Technology (FPT)*, 2018.

[OL18] Hossein Omidian and Guy G. F. Lemieux. JANUS: A compilation system for balancing parallelism and performance in OpenVX. *Journal of Physics:Series*, 1004(1):012011, 2018. DOI: 10.1088/1742-6596/1004/1/012011.

[OLQ+14] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23. IEEE, 2014.

[ORK+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[ORS+13] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the International on Generative Programming: Concepts & Experiences (GPCE)* (Indianapolis, IN, USA), pages 125–134. ACM, October 27–28, 2013. DOI: 10.1145/2517208.2517228.

[PBY+17] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM on Architecture and Code Optimization (TACO)*, 14(3):26:1–26:25, 2017. DOI: 10.1145/3107953.

[PCC+14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[PCH+13] Alexandros Papakonstantinou, Deming Chen, Wen-Mei Hwu, Jason Cong, and Yun Liang. Throughput-oriented kernel porting onto FPGAs. In *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC)* (Austin, TX, USA), 11:1–11:10, May 29–June 7, 2013. DOI: 10.1145/2463209.2488747.

[PGS+09] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the IEEE 7th Symposium on Application Specific Processors (SASP)* (San Francisco, CA, USA), pages 35–42, July 27–28, 2009. DOI: 10.1109/SASP.2009.5226333.

[PK86] Pierre G Paulin and JP Knight. HAL: A multiparadigm approach to automatic data path synthesis. In *23rd Design Automation Conference*, pages 263–270, 1986.

[PK89] Pierre G Paulin and John P Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[PPM86] Alice C Parker, Jorge Pizarro, and Mitch Mlinar. Maha: a program for datapath synthesis. In *23rd ACM/IEEE Design Automation Conference*, pages 461–466. IEEE, 1986.

[PTS+79] Alice C Parker, Donald E Thomas, Daniel P Siewiorek, Mario R Barbacci, A Parker, D Thomas, D Siewiorek, M Barbacci, L Hafer, G Leive, et al. The cmu design automation system: an example of automated data-path design, 1979.

[QÖT+20a] Bo Qiao, M Akif Özkan, Jürgen Teich, and Frank Hannig. The best of both worlds: Combining CUDA graph with an image processing DSL. In *Proceedings of the 57th Annual Design Automation Conference (DAC)* (San Francisco, USA), 2020.

[QRH+19] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (Washington DC, USA), 2019.

[RAK18] Dustin Richmond, Alric Althoff, and Ryan Kastner. Synthesizable higher-order functions for C++. *on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2835–2844, 2018.

[RBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the on Programming Language Design and Implementation (PLDI)* (Seattle, WA, USA), pages 519–530. ACM, June 16–19, 2013. DOI: 10.1145/2491956.2462176.

[RKH+17] Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. Auto-vectorization for image processing DSLs. In *ACM SIGPLAN Notices*, volume 52 of number 5, pages 21–30. ACM, 2017.

[RN16] Mohammad Rafi and Najeeb-ud-Din. A novel arrangement for efficiently handling image border in FPGA filter implementation. In *Proceedings of the 3rd International Conf. on Signal Processing and Integrated Networks (SPIN)*, pages 163–168. IEEE, 2016.

[RÖM+17b] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-based image processing accelerators with Hipacc. In *Proceedings of the International On Computer Aided Design (ICCAD)* (Irvine, CA, USA), pages 1026–1033. IEEE, November 13–16, 2017. ISBN: 978-1-5386-3094-5. DOI: 10.1109/ICCAD.2017.8203894.

[RSH+14] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *2014 international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, pages 1–10. IEEE, 2014.

[RVD+14] Erik Rainey, Jesse Villarreal, Goksel Dedeoglu, Kari Pulli, Thierry Lepley, and Frank Brill. Addressing system-level optimization with OpenVX graphs. In *Proceedings of the on Computer Vision and Pattern Recognition Workshops*, pages 644–649. IEEE, 2014.

[SAH+14] Moritz Schmid, Nicolas Apelt, Frank Hannig, and Jürgen Teich. An image processing library for C-based high-level synthesis. In *Proceedings of the 24th Intl. Conf. on Field Programmable Logic and Applications (FPL)* (Munich, Germany), September 2–4, 2014.

[San06] John Sanguinetti. A different view: Hardware synthesis from SystemC is a maturing technology. *IEEE Design & Test of Computers*, 23(5):387–387, 2006. DOI: 10.1109/MDT.2006.111.

[Sér+13] Jocelyn Sérot et al. CAPH: a language for implementing stream-processing applications on FPGAs. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, 2013.

[SFL+15] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.

[SGE+18] Greg Stitt, Abhay Gupta, Madison N Emas, David Wilson, and Austin Baylis. Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems. In *Proceedings of the ACM/SIGDA InternationalSymposium on Field-Programmable Gate Arrays (FPGA)* (Monterey, CA, USA), pages 173–182. ACM, February 25–27, 2018. DOI: 10.1145/3174243.3174262.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[SJ07] Olli Silven and Kari Jyrkkä. Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal on Embedded Systems*, 2007:1–10, 2007.

[SKA13] Kavya Shagrithaya, Krzysztof Kępa, and Peter M. Athanas. Enabling development of OpenCL applications on FPGA platforms. In *Proceedings of the IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)* (Washington, DC, USA), pages 26–30. IEEE, June 5–7, 2013. DOI: 10.1109/ASAP.2013.6567546.

[SLL02] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: User guide and reference manual.* Addison-Wesley, 2002.

[SMB+16] Robert Stewart, Greg Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. A dataflow IR for memory efficient RIPL compilation to FPGAs. In *Proceedings of the International on Algorithms and Architectures for Parallel Processing (ICA3PP)* (Granada, Spain), pages 174–188. Springer, December 14–16, 2016. DOI: 10.1007/978-3-319-49956-7_14.

[SRH+15] Moritz Schmid, Oliver Reiche, Frank Hannig, and Jürgen Teich. Loop coarsening in C-based high-level synthesis. In *Proceedings of the 26th Annual IEEE International on Application-specific Systems, Architectures and Processors (ASAP)*, pages 166–173. IEEE, 2015.

[ST06] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In *ECOOP 2006–Object-Oriented Programming: 20th European Conference, Nantes, France, July 3-7, 2006. Proceedings 20*, pages 304–327. Springer, 2006.

[Ste04] Fridtjof Stein. Efficient computation of optical flow using the census transform. In *Pattern Recognition.* Volume 3175, Lecture Notes in Computer Science, pages 79–86. Springer, 2004. ISBN: 978-3-540-22945-2. DOI: 10.1007/978-3-540-28649-3_10.

[Tay12] Michael B Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136. IEEE, 2012.

[Tei12] Jürgen Teich. Hardware/software codesign: the past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[Tei93] Jürgen Teich. *A compiler for application specific processor arrays.* Dissertation, Saarland University, Germany, Shaker, 1993, pages 1–230. ISBN: 978-3-86111-701-8.

[THB+18] Sajjad Taheri, Jin Heo, Payman Behnam, Jeffrey Chen, Alexander Veidenbaum, and Alexandru Nicolau. Acceleration framework for FPGA implementation of OpenVX graph pipelines. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–227. IEEE, 2018. DOI: 10.1109/FCCM.2018.00061.

[The14] The Khronos Group. Khronos finalizes and releases OpenVX 1.0 specification for computer vision acceleration. Press Release, October 2014.

[The18a] The Khronos Group. OpenVX resources, August 2018.

[The18b] The Khronos Vision Working Group and others. The OpenVX specification v1.2.1, August 2018.

[The19] The Khronos Vision Working Group and others. The OpenVX specification v1.3, August 2019.

[THM+16] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. Enabling OpenVX support in mW-scale parallel accelerators. In *Proceedings of the International on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2016. DOI: 10.1145/2968455.2968518.

[THM+18] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. Optimizing memory bandwidth exploitation for OpenVX applications on embedded many-core accelerators. *Journal of Real-Time Image Processing*, 15(1):73–92, 2018. DOI: 10.1007/s11554-015-0544-0.

[TM98] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision (ICCV)*, pages 839–846. IEEE, 1998. ISBN: 81-7319-221-9.

[VP07] Yevgen Voronenko and Markus Püschel. Multiplierless multiple constant multiplication. *ACM Transactions on Algorithms (TALG)*, 3(2), 2007.

[WCC09] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: Synthesis, verification, and test.* Morgan Kaufmann, 2009.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. DOI: 10.1145/1498765.1498785.

[Xil14] Xilinx. Core evaluation license agreement, version 2014.06, Xilinx, Inc., June 2014. URL: https://www.xilinx.com/products/intellectual-property/license/core-evaluation-license-agreement.html.

[Xil17a] Xilinx. UltraScale Architecture GTH Transceivers UG576, version v4.0, 2017.

[Xil17b] Xilinx. Vivado AXI Reference Guide UG1037, version v4.0, 2017.

[Xil17c] Xilinx. Vivado Design Suite user guide high-level synthesis UG902, 2017.

[Xil19] Xilinx. Introduction to FPGA design with Vivado HLS UG998, 2019.

[Xil23] Xilinx. Xilinx xfopencv library, June 2023. URL: https://github.com/Xilinx/xfopencv.

[YAY+18] Ming Yang, Tanya Amert, Kecheng Yang, Nathan Otterness, James H Anderson, F Donelson Smith, and Shige Wang. Making OpenVX really “real-time”. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2018.

[YJH+87] Fathy F Yassa, Jeffrey R Jasica, Richard I Hartley, and Sharbel E Noujaim. A silicon compiler for digital signal processing: Methodology, implementation, and applications. *Proceedings of the IEEE*, 75(9):1272–1282, 1987.

[ZFJ+08] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer, 2008. ISBN: 978-1-4020-8587-1. DOI: 10.1007/978-1-4020-8588-8_6.

[ZTS18] Jinghan Zhang, Hamed Tabkhi, and Gunar Schirner. DS-DSE: Domain-specific design space exploration for streaming applications. In *Proceedings of the on Design, Automation and Test in Europe (DATE)*, pages 165–170. IEEE, 2018. DOI: 10.23919/DATE.2018.8341997.

Author’s Own Publications

[HRR+16] Konrad Häublein, Marc Reichenbach, Oliver Reiche, M. Akif Özkan, Dietmar Fey, Frank Hannig, and Jürgen Teich. Hybrid code description for developing fast and resource efficient image processing architectures. In *Proceedings of the 16th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (Island of Samos, Greece), pages 211–218. IEEE, July 18–21, 2016. ISBN: 978-1-5090-3076-7. DOI: 10.1109/SAMOS.2016.7818350.

[ÖRH+16] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. FPGA-based accelerator design from a domain-specific language. In *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*, Lausanne, Switzerland. Institute of Electrical and Electronics Engineers (IEEE), August 29–September 2, 2016. ISBN: 978-2-8399-1844-2. DOI: 10.1109/FPL.2016.7577357.

[ÖRH+17a] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. A highly efficient and comprehensive image processing library for C++-based high-level synthesis. In *Proceedings of the Fourth International Workshop on FPGAs for Software Programmers (FSP)* (Ghent, Belgium), pages 23–32. VDE, September 7, 2017. ISBN: 978-3-8007-4443-5.

[ÖRH+17b] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. Hardware design and analysis of efficient loop coarsening and border handling for image processing. In *Proceedings of the 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (Seattle, WA, USA), pages 155–163. IEEE, July 10–12, 2017. ISBN: 978-1-5090-4825-0. DOI: 10.1109/ASAP.2017.7995273.

[RÖM+17a] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-based image processing accelerators with Hipacc. In *Proceedings of the International Conference On Computer Aided Design (ICCAD)* (Irvine, CA, USA), pages 1026–

1033. IEEE, November 13–16, 2017. ISBN: 978-1-5386-3094-5. DOI: 10.1109/ICCAD.2017.8203894.

[ÖPM+18] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Jürgen Teich, and Frank Hannig. A journey into DSL design using generative programming: FPGA mapping of image border handling through refinement. In *Proceedings of the Fifth International Workshop on FPGAs for Software Programmers (FSP)* (Dublin, Ireland). VDE, August 31, 2018. ISBN: 978-3-8007-4723-8.

[RÖH+18] Oliver Reiche, M. Akif Özkan, Frank Hannig, Jürgen Teich, and Moritz Schmid. Loop parallelization techniques for FPGA accelerator synthesis. *Journal of Signal Processing Systems*, 90(1):3–27, January 2018. ISSN: 1939-8018. DOI: 10.1007/s11265-017-1229-7.

[ÖRQ+19] Mehmet Akif Özkan, Oliver Reiche, Bo Qiao, Richard Membarth, Jürgen Teich, and Frank Hannig. Synthesizing High-Performance Image Processing Applications with Hipacc. In *University Booth at Design, Automation and Test in Europe (DATE)* (Florence), March 25–29, 2019.

[ÖPM+20a] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. AnyHLS: High-Level Synthesis with Partial Evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39, 2020. DOI: 10.1109/TCAD.2020.3012172.

[ÖPM+20b] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. AnyHLS: High-level synthesis with partial evaluation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Virtual Conference), September 20–25, 2020.

[QÖT+20b] Bo Qiao, M. Akif Özkan, Jürgen Teich, and Frank Hannig. The best of both worlds: Combining CUDA graph with an image processing DSL. In *Proceedings of the 57th Annual Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE, July 19–23, 2020. ISBN: 978-1-7281-1085-1.

[QRÖ+20] Bo Qiao, Oliver Reiche, M. Akif Özkan, Jürgen Teich, and Frank Hannig. Efficient parallel reduction on GPUs with Hipacc. In *Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES)* (St. Goar, Germany), pages 58–61. ACM, May 25–26, 2020. ISBN: 978-1-4503-7131-5. DOI: 10.1145/3378678.3391885.

[ÖOQ+21] M. Akif Özkan, Burak Ok, Bo Qiao, Jürgen Teich, and Frank Hannig. HipaccVX: Wedding of OpenVX and DSL-based code generation. *Journal of Real-Time Image Processing*, 18:765–777, 2021. DOI: 10.1007/s11554-020-01015-5.

[SÖK+22a] Jan Sommer, M Akif Özkan, Oliver Keszocze, and Jürgen Teich. DSP-Packing: Squeezing low-precision arithmetic into FPGA DSP blocks. In *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*, Belfast, United Kingdom. Institute of Electrical and Electronics Engineers (IEEE), August 29, 2022.

[SÖK+22b] Jan Sommer, M. Akif Özkan, Oliver Keszocze, and Jürgen Teich. Efficient hardware acceleration of sparsely active convolutional spiking neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):3767–3778, 2022. DOI: 10.1109/TCAD.2022.3197512.

Acronyms

AI Artificial Intelligence . . . 7
ALM Adaptive Logic Module . . . 39
ALU Arithmetic Logic Unit . . . 7
AOCL Altera SDK for OpenCL . . . 184
API Application Programming Interface . . . vi
ASIC Application-Specific Integrated Circuit . . . v
AST Abstract Syntax Tree . . . vi
AVX Advanced Vector Extensions . . . 11

BRAM Block Random Access Memory . . . 39

C&P Calc and Pack . . . 66
CDFG Control and Data Flow Graph . . . 16
CGRA Coarse-Grained Reconfigurable Architecture . . . 185
CLB Configurable Logic Block . . . 39
CPU Central Processing Unit . . . vii
CU Computing Unit . . . 9
CUDA Compute Unified Device Architecture . . . 98
CV Computer Vision . . . vii

DAG Directed Acyclic Graph . . . 99
DDR SDRAM Double Data Rate Synchronous Dynamic Random-Access Memory 44
DFG Data Flow Graph . . . 60
DLP Data-Level Parallelism . . . 7

DMA Direct Memory Access 58
DNN Deep Neural Network 153
DSA Domain-Specific Architecture 7
DSL Domain-Specific Language vi
DSP Digital Signal Processor 7

EDA Electronic Design Automation 13
ESL Electronic System-Level 12

F&C Fetch and Calc 66
FF Flip-Flop 38
FIFO First In First Out 52
FPGA Field-Programmable Gate Array v
FSM Finite State Machine 15
FSMD Finite State Machine with Data Path 12

GOPS giga operations per second 2
GPU Graphics Processing Unit vi

HBM High Bandwidth Memory 58
HDL Hardware Description Language v
Hipacc Heterogeneous Image Processing Acceleration vi
HLS High-Level Synthesis v

ICT Information and Computing Technologies 3
IDC International Data Corporation 2
II Initiation Interval 46
ILP Instruction-Level Parallelism 3
IO Input/Output 40
IR Intermediate Representation 110
ISA Instruction Set Architecture v
ITRS International Technology Roadmap for Semiconductors 3

LAB Logic Array Block 39

LE	Logic Element	39
LLVM	Low Level Virtual Machine	16
LUT	Lookup Table	14
MCM	Multiplierless multiple constant multiplication	220
MPMD	Multiple Program, Multiple Data	175
MPSoC	Multiprocessor System-on-a-Chip	143
MUX	Multiplexer	12
NRE	non-recurring engineering	215
OFDM	Orthogonal Frequency-Division Multiplexing	2
OpenCL	Open Computing Language	95
PCIe	Peripheral Component Interconnect Express	20
PGM	Portable Graymap File Format	107
PPnR	Post Place and Route	175
QoR	Quality of Results	19
RAM	Random-Access Memory	39
RGBA	Red, Green, Blue, Alpha Channel	65
ROI	Region Of Interest	102
RT	Register-Transfer	12
RTL	Register-Transfer Level	v
SIMD	Single Instruction, Multiple Data	7
SMT	Single instruction, multiple threads	8
SoC	System-on-a-Chip	11
SPMD	Single Program, Multiple Data	167
SRAM	Static Random-Access Memory	38
SSA	Single Static Assignment	112
TPU	Tensor Processing Unit	7
UML	Unified Modeling Language	136
VLIW	Very Long Instruction Word	8