TESIS DE LA UNIVERSIDAD DE ZARAGOZA
2022
58

María Ángelica Dávila Gúzman

FPGA Accelerators on Heterogeneous Systems: An Approach Using High Level Synthesis

Director/es
Villarroya Gaudó, María
Suárez Gracia, Darío

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

ISSN 2254-7606

Universidad Zaragoza
1542

Tesis Doctoral

FPGA ACCELERATORS ON HETEROGENEOUS SYSTEMS: AN APPROACH USING HIGH LEVEL SYNTHESIS

Autor

María Ángelica Dávila Gúzman

Director/es

Villarroya Gaudó, María
Suárez Gracia, Darío

UNIVERSIDAD DE ZARAGOZA
Escuela de Doctorado

Programa de Doctorado en Ingeniería de Sistemas e Informática

2022

FPGA Accelerators on Heterogeneous Systems: An Approach Using High Level Synthesis

---

Maria Angélica Dávila Guzmán

November, 2021
Version: Draft

Universidad Zaragoza

Departamento de Informática e Ingeniería de sistemas
Instituto de Investigación es Ingeniería de Aragón
Grupo de Arquitectura de Computadores

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy

FPGA Accelerators on Heterogeneous Systems: An Approach Using High Level Synthesis

Maria Angélica Dávila Guzmán

Supervisors Darío Suárez Gracia
María Villarroya Gaudó

November, 2021

Maria Angélica Dávila Guzmán

FPGA Accelerators on Heterogeneous Systems:
An Approach Using High Level Synthesis

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy , November, 2021

Supervisors: Darío Suárez Gracia and María Villarroya Gaudó

Universidad de Zaragoza
Grupo de Arquitectura de Computadores
Instituto de Investigación es Ingeniería de Aragón
Departamento de Informática e Ingeniería de sistemas
Calle Maria de Luna
50018
Zaragoza

Abstract

The emergence of FPGAs in the High-Performance Computing domain is arising thanks to their promise of better energy efficiency and low control latency, compared with other devices such as CPUs or GPUs.

Albeit these benefits, their complete inclusion into HPC systems still faces several challenges. First, FPGA complexity means its programming more difficult compared to devices such as CPU and GPU. Second, the development time is longer due to the required synthesis effort. And third, working with multiple devices increments the details that should be managed and increase hardware complexity.

This thesis tackles these 3 problems at different stack levels to improve and to make easier the adoption of FPGAs using High-Level Synthesis on HPC systems. At a close to the hardware level, this thesis contributes with a new analytical model for memory-bound applications, an usual situation for HPC applications. The model for HLS kernels can anticipate application performance before place and route, reducing the design development time. Our results show a high precision and adaptable model for external memory technologies such as DDR4 and HBM2, and kernel frequency changes. This solution potentially increases productivity, reducing application development time.

Understanding low-level implementation details is difficult for average programmers, and the development of FPGA applications still requires high proficiency programming skills. For this reason, the second proposal is focused on the extension of a computer vision library to be portable among two of the main FPGA vendors. The template-based library allows hardware flexibility and hides design decisions such as the communication among nodes, the concurrency programming model, and the application's integration in the heterogeneous system, to develop complex vision graphs easily.

Finally, we have transparently integrated the FPGA in a high level framework for co-execution with other devices. We propose a set of high level abstractions covering synchronization mechanism and load balancing policies in a highly heterogeneous system with CPU, GPU, and FPGA devices. We present the main challenges that inspired this research and the benefits of the FPGA use demonstrating performance and energy improvements.

In summary, this thesis contributes to the adoption of FPGAs in HPC domains by offering solutions that help reducing development time and improve performance and energy efficiency.

Resumen

La evolución de las FPGAs como dispositivos para el procesamiento con alta eficiencia energética y baja latencia de control, comparada con dispositivos como las CPUs y las GPUs, las han hecho atractivas en el ámbito de la computación de alto rendimiento (HPC).

A pesar de las inumerables ventajas de las FPGAs, su inclusión en HPC presenta varios retos. El primero, la complejidad que supone la programación de las FPGAs comparada con dispositivos como las CPUs y las GPUs. Segundo, el tiempo de desarrollo es alto debido al proceso de síntesis del hardware. Y tercero, trabajar con más arquitecturas en HPC requiere el manejo y la sintonización de los detalles de cada dispositivo, lo que añade complejidad.

Esta tesis aborda estos 3 problemas en diferentes niveles con el objetivo de mejorar y facilitar la adopción de las FPGAs usando la síntesis de alto nivel(HLS) en sistemas HPC.

En un nivel próximo al hardware, en esta tesis se desarrolla un modelo analítico para las aplicaciones limitadas en memoria, que es una situación común en aplicaciones de HPC. El modelo, desarrollado para kernels programados usando HLS, puede predecir el tiempo de ejecución con alta precisión y buena adaptabilidad ante cambios en la tecnología de la memoria, como las DDR4 y HBM2, y en las variaciones en la frecuencia del kernel. Esta solución puede aumentar potencialmente la productividad de las personas que programan, reduciendo el tiempo de desarrollo y optimización de las aplicaciones.

Entender los detalles de bajo nivel puede ser complejo para las programadoras promedio, y el desempeño de la aplicaciones para FPGA aún requiere un alto nivel en las habilidades de programación. Por ello, nuestra segunda propuesta está enfocada en la extensión de las bibliotecas con una propuesta para cómputo en visión artificial que sea portable entre diferentes fabricantes de FPGAs. La biblioteca se ha diseñado basada en templates, lo que permite una biblioteca que da flexibilidad a la generación del hardware y oculta decisiones de diseño críticas como la comunicación entre nodos, el modelo de concurrencia, y la integración de las aplicaciones en el sistema heterogéneo para facilitar el desarrollo de grafos de visión artificial que pueden ser complejos.

Finalmente, en el runtime del host del sistema heterogéneo, hemos integrado la FPGA para usarla de forma trasparente como un dispositivo acelerador para la co-ejecución en sistemas heterogéneos. Hemos hecho una serie propuestas de alto nivel de abstracción que abarca los mecanismos de sincronización y políticas de balanceo en un sistema altamente heterogéneo compuesto por una CPU, una GPU y una FPGA. Se presentan los principales retos que han inspirado esta investigación y los beneficios de la inclusión de una FPGA en rendimiento y energía.

En conclusión, esta tesis contribuye a la adopción de las FPGAs para entornos HPC, aportando soluciones que ayudan a reducir el tiempo de desarrollo y mejoran el desempeño y la eficiencia energética del sistema.

Contents

1 Introduction **5**
- 1.1 Rationale . . . 5
- 1.2 Objectives and Dissertation Overview . . . 7
- 1.3 Contributions . . . 8
- 1.4 Thesis Project Framework . . . 9

2 FPGAs as Accelerator in Heterogeneous System **11**
- 2.1 Introduction . . . 11
- 2.2 Heterogeneous System . . . 12
- 2.3 FPGA Device . . . 15
  - 2.3.1 FPGA Program Design . . . 19
  - 2.3.2 FPGAs as Accelerator on heterogeneous system . . . 21

3 Experimental Framework **25**
- 3.1 Evaluation Hardware . . . 25
- 3.2 Software Framework . . . 30

4 Analytical Time Estimation of Memory-bound Applications for FPGA Using High-level Synthesis **33**
- 4.1 Introduction . . . 33
- 4.2 Related Work . . . 35
- 4.3 FPGA External Memory and BSP . . . 36
  - 4.3.1 Global Memory Interconnect . . . 37
- 4.4 Performance Estimation for FPGAs . . . 40
- 4.5 Memory Model for FPGA accelerators . . . 41
  - 4.5.1 Burst-Coalesced LSU . . . 45
  - 4.5.2 Atomic-pipelined LSU . . . 48
- 4.6 Methodology . . . 49
- 4.7 Results . . . 51
  - 4.7.1 Microbenchmarks . . . 51
  - 4.7.2 Applications . . . 56
  - 4.7.3 Comparison With Other Models . . . 59
- 4.8 Conclusions . . . 60
- 4.9 Contributions . . . 61

5 FPGA Frameworks to Improve Design Productivity 63
5.1 Introduction . . . 63
5.2 Related Work . . . 64
5.3 OpenVX Programming Flow Alternatives on FPGA . . . 65
5.4 HiFlipVX . . . 67
5.5 Methodology . . . 68
5.6 Tuning HiFlipVX for Intel FPGAs . . . 69
5.6.1 Execution model . . . 69
5.6.2 Kernels . . . 70
5.6.3 Edges . . . 70
5.7 Results . . . 75
5.7.1 HiFlipVX Scalability Analysis . . . 75
5.7.2 OpenVX Application Resource Utilization . . . 76
5.7.3 OpenVX Application Analysis . . . 80
5.7.4 Comparison with Existing Approaches . . . 80
5.7.5 Tiling HiFlipVX for HBM memory . . . 83
5.8 Conclusions . . . 84
5.9 Contributions . . . 85

6 FPGAs on Heterogeneous System for Energy Efficiency 87
6.1 Introduction . . . 87
6.2 Related Work . . . 89
6.3 EngineCL Runtime . . . 90
6.4 Coupling FPGA to EngineCL . . . 92
6.5 Methodology . . . 95
6.6 Results . . . 96
6.6.1 Scheduler Tuning . . . 96
6.6.2 Scheduler comparison . . . 98
6.7 Conclusions . . . 102
6.8 Contributions . . . 103

7 Conclusions and Future Work 105
7.1 Conclusions . . . 105
7.2 Future Work . . . 107

Bibliography 109

8 Appendix: Memory aware co-execution in heterogeneous system with CPU and FPGA with HBM2 memory 123
8.1 Introduction . . . 123

8.2 FPGA with HBM memory in a Heterogeneous system 124

List of Figures **129**

List of Tables **133**

Listings **135**

Abbreviations

FPGA	**F**ield **P**rogrammable **G**ate **A**rray
BSP	**B**oard **S**upport **P**ackage
GMI	**G**lobal **M**emory **I**nterconnect
HBM	**H**igh **M**emory **B**andwidth
DDR	**D**ouble **D**ata **A**rray memory
HLS	**H**igh **L**evel **S**ynthesis
HDL	**H**ardware **D**esign **L**evel
LSU	**L**oad **S**tore **U**nit
IP	**I**ntellectual **P**roperty core
PCIe	**P**eripheral **C**omponent **I**nterconnect **e**xpress
RTL	**R**egister **T**ransfer **L**evel

Publications

Part of this dissertation includes previous results of published or under review papers of my authorship. The works in collaboration are possible thanks to two tools developed in two universities: 1) a runtime for co-execution in heterogeneous system (EngineCL) from the University of Cantabria, and 2) a HLS open source library for Xilinxs FPGAs (HiFlipVX) from the University of Dresden.

The list of publications in chronological order are:

1. **Maria Angélica Dávila-Guzmán**, Raúl Nozal, Rubén Gran, Maria Villaroya-Gaudó, Darío Suárez, and José Luis Bosque. First Steps Towards CPU, GPU, and FPGA Parallel Execution with EngineCL. Proceedings of the 18^{th} International Conference on Computational and Mathematical Method in Science and Engineering. CMMSE 2018.

2. **Maria Angélica Dávila-Guzmán**, Rubén Gran Tejero, María Villarroya-Gaudó, and Darío Suárez Gracia. Caracterización de una FPGA sobre un sistema heterogéneo usando OpenCL. In Jornadas de la sociedad de Arquitectura y Tecnología de Computadoras (SARTECO), pages 75–83, 2018.

3. **Maria Angelica Davila Guzman**, Ruben Gran Tejero, Maria Villarroya Gaudo, and Dario Suarez Gracia. Towards the inclusion of FPGAs on commodity heterogeneous systems. In 2018 International Conference on High Performance Computing Simulation (HPCS), pages 554–556, 2018.
   B CORE 2018

4. **María Angélica Dávila Guzmán**, Raúl Nozal, Rubén Gran Tejero, María Villarroya- Gaudó, Darío Suárez Gracia, and Jose Luis Bosque. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. The Journal of Supercomputing, 75(3): 1732–1746, 2019.
   Q2 JCR 2019

5. **Maria A.Dávila-Guzmán**, Rubén Gran Tejero,María Villarroya-Gaudó,and Darío Suárez Gracia. An analytical model of memory-bound applications

compiled with high-level synthesis. In 2020 IEEE 28^{th} Annual International Symposium on Field- Programmable Custom Computing Machines (FCCM), pages 218–218, 2020. (Poster).
A CORE 2020. HiPEAC paper award

6. **Maria Angélica Dávila-Guzmán**, Rubén Gran Tejero, Maria Villaroya-Gaudó, Darío Suárez, and José Luis Bosque. Memory aware co-execution in heterogeneous systems with CPU and FPGA devices. International Conference on Computational and Mathematical Method in Science and Engineering, CMMSE 2020.

7. **Maria Angélica Dávila-Guzmán**, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, Lester Kalms, and Diana Göhringer. A cross-platform OpenVX library for FPGA accelerators. In 2021 29^{th} Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 75–83, 2021.
C CORE 2021

8. **Maria A.Dávila-Guzmán**,Rubén GranTejero,María Villarroya-Gaudó,and Darío Suárez Gracia. Analytical Model for Memory-centric High Level Synthesis-generated Applications. IEEE Transactions on Computers, 2021.
Q2 JCR 2020

9. **Maria Angélica Dávila-Guzmán**, Rubén Gran Tejero, María Villarroya-Gaudó, Darío Suárez Gracia, Lester Kalms, and Diana Göhringer. A Cross-Platform OpenVX Library for FPGA Accelerators (PDP extension). Journal of Systems Architecture, 2021.
Q2 JCR 2020 (under minor review)

1 Introduction

This chapter presents the reason for including the FPGAs in heterogeneous systems and the uprising programmability challenges. Also, it lists the objectives, the achieved contributions and the project framework that has allowed the development of this thesis.

1.1 Rationale

At the end of Moore's law [108] and Dennard's scaling [35], the emergence of power and temperature constrains in computer system is driving computer hardware architects to new exciting perspectives. Maintain Moore's law was a deliberated choice of chip manufacturers, but in early 2000s, when the transistors size was below 90 nm, the expected benefits of higher computational power with lower energy consumption achieved only by scaling transistors started to fail.

The physical impossibility of placing more transistors in the same area switching at high frequency has forced to stop increasing the clock frequency in processors. To keep moving in the processors Moore's performance curve, during 2010's the number of cores in a processor was increased at lower frequencies, which in turn stopped the excessive power dissipation [104]. Heterogeneous era started with the combination of transistor technologies, evolved to many-core processors, and grew up with the inclusion of specialized processors (accelerators) in the computing systems such as GPUs, FPGAs, DSPs ...

Computer systems with accelerators have demonstrated high energy efficiency and compute power to support the market necessities . Some of these systems have even achieved good positions in the worldwide rankings such as TOP500 [153] of high performance computing and GREEN500 [152] of best energy efficiency in the last years [153].

More devices and hardware heterogeneity come with a burden in programming languages, forcing developers to learn unfamiliar tools and increasing the complexity of software development. One way to reduce heterogeneous system complexity is software abstractions, that can put away the intricacies with high-level frameworks and intelligent runtimes. When they succeed, they enable the computing capabilities of heterogeneous systems without exposing their nuances. The industry is driving initiatives of programming language standardization for heterogeneous systems with languages like OpenCL and SyCL, which abstract hardware architecture details in favor of development productivity. This enables developers to focus on the algorithm development and not on managing system resources. In these terms, the research can be concentrated on how to distribute the work load among the available devices efficiently in a heterogeneous system to improve compute performance and energy efficiency. Still, this approach is not yet broadly supported by industry but has been successfully explored with GPU devices [117, 94, 120, 1, 96, 53, 128, 165], FPGAs devices[53, 5, 160, 141], but it is not fully analyzed with higher heterogeneity including CPU, GPUs and FPGAs[139]. The last part of this dissertation includes one of the first proposals to enable co-execution between the three aforementioned devices.

The FPGAs have recently appeared in the accelerators ecosystem promising high power efficiency and easier programmability. Besides, new programming paradigms try to facilitate and promote their broad adoption. For example, the introduction of High-Level Synthesis has significant importance to improve programmer's productivity using high-level programming languages as C/C++, OpenCL, and recently SYCL[147]. The High-Level Synthesis in FPGAs is not new and has a long history of fails [99], the design space could have thousand of solutions, so it took long time to develop HLS-based tools. The evolution of the transformation from C to RTL was responsible for the take off of the FPGA market in 2004 [99], and in 2015 with the Intel announce of Altera's acquisition, one of the main FPGA manufacturers, the new technological road map was opened with the idea of FPGA integration with Xeon processors [62, 56].

The inclusion of an FPGA accelerator in a heterogeneous system ensuring high programmers productivity is still under development [18]. Although heterogeneous programming languages promise single-source and functional portable programs, the performance portability has not achieved the same level of maturity in industrial framework's yet. FPGA design development has remarkable differences compared

with CPUs and GPUs, even using High-Level Synthesis, since the FPGAs architecture is very flexible and requires the expression of low level details to achieve optimal performance. Consequently, the research community has to develop and support the application frameworks for program optimization and even build the set of benchmarks for testing, which mismatch among FPGA vendors.

In summary, adding higher heterogeneity with new devices as FPGAs requires more than a high-level programming framework; the programmers need to understand the device's architecture and main bottlenecks in an early compilation stage to reduce the development time. This approximation to FPGA optimization could be a long way for custom programs. Relying only on the OpenCL standard is insufficient, since developers require specifications with sufficient low level and near the desired task to easily describe the parallelism with specific FPGA-friendly frameworks, forgetting coarse details as FPGA model features, and focusing on performance. These approaches are covered in this dissertation, particularly in Chapters 4 and 5.

1.2 Objectives and Dissertation Overview

The overall goal of this dissertation is to explore the inclusion of a high energy efficient device as the FPGA in a heterogeneous system. This goal entails their programmability as the main deterrent to be broadly adopted by programmers in a productive way.

This dissertation is organized as follow:

Chapter 2 presents the background on heterogeneous systems and the FPGA architecture, including application design, and FPGA support to be used as an accelerator device. The following Chapter 3 presents the experimental framework.

Chapter 4 explores performance modeling as an option to improve FPGA applications using high-level synthesis. We present a new model for memory bound applications in FPGAs, which estimates the time performance of the applications at an early compilation stage, without the long time required for full compilation.

Chapter 5 explores the portability of FPGA applications among vendors using a high-level framework for computer vision acceleration based on the OpenVX standard.

We propose an extension to support Intel FPGA devices in an open library originally designed for Xilinx devices. Also the framework is extended to be used in FPGAs as a discrete accelerator combining C/C++ and OpenCL programming languages.

Chapter 6 presents the exploration of load balancing algorithms in a system with three different accelerators: CPU, GPU, and FPGA. We integrate the FPGA to an existing runtime library that supports the load balancing algorithm and evaluates the performance in terms of time and energy efficiency.

Finally, with all the contributions to the main objective in previous chapters, in Chapter 7 we conclude and brief possible future research lines.

1.3 Contributions

This thesis includes the following contributions:

- Analysis of the FPGA architecture as an accelerator for one of the main FPGA vendors focused in the memory hierarchy. We determine the most time-critical blocks interacting between kernel memory request and the external memory controller.
- We develop a memory model for FPGAs using high-level synthesis in memory bound applications. We validate the model with a set of benchmarks and comparisons with the state-of-the-art, showing the estimation improvements of this proposal using a standard DDR4 and novel HBM2 memory.
- We propose a set of memory-oriented hints for programmers to guide optimizations using high level synthesis.
- We integrate two of the major FPGA vendors in a library for computer vision applications using the OpenVX standard. This library was carefully ported to Intel FPGA devices and extended to support stream communication among functions to allow fully graph implementations, and extensions to work in FPGA accelerators using OpenCL and C++. These results demonstrate portability and performance improvements.
- We present one of the first results of load balancing in a full heterogeneous system including three different accelerator devices: CPU, GPU, and FPGA.

- We analyze performance results of a heterogeneous system in terms of time and energy efficiency considering all three devices running at the same time. We demonstrate the increase in application performance and energy efficiency with the use of more accelerators.

1.4 Thesis Project Framework

This thesis has been developed at the Computer Architecture Group of Zaragoza (gaZ) of the University of Zaragoza, in the Department of Computer Science and Systems Engineering (DIIS) and the Engineering Research Institute of Aragon (I3A).

The Santander - University of Zaragoza collaboration grant for Latin-American during two years (2017-2019) supported me as PhD Student. I have developed a research internship at the Technical University of Dresden, in the Adaptive Dynamic Systems group under the supervision of Diana Goehringer, thanks to a competitive collaboration grant from HiPEAC.

The research work has been funded by the Spanish National Science Research System, under the project PID2019-105660RB-C21: Jerarquía de memoria, gestión de tareas y optimización de aplicaciones, from the Agencia Estatal de Investigación and TIN2016-76635-C2-1-R: Arquitectura y programación de computadores escalables de alto rendimiento y bajo consumo, from the Spanish Ministry of Economy and Competitive. Both projects are in collaboration with the University of Cantabria which led to a fruitful collaboration in this thesis. Also the Aragón Government has partially founded the work through the Research group recognition: T58_20R research group from Aragón Government and European Social Fund, and (3) 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund. Part of the FPGA board development kits used in this work and Quartus software licenses were a donation from Intel, and Nvidia gave away the GPU.

2 FPGAs as Accelerator in Heterogeneous System

This chapter presents an overview of the heterogeneous systems, their programmability, and how they impulse the use of FPGAs. We discuses synthesis challenges from RTL design to the evolution to high-level synthesize. Finally, we emphasize in the FPGAs on heterogeneous system using high level languages which is the main focus of this dissertation.

2.1 Introduction

Hardware heterogeneity as an alternative for high throughput and energy efficient processing includes accelerators as GPUs and FPGAs. One of the main drawbacks of a heterogeneous system is how to use the available hardware resources efficiently. This problem needs to be addressed in two ways: the first one is how to efficiently distribute the workload among available devices since each architecture performs differently according to algorithm requirements, and accelerators entail different sources of overheads as workload dispatch and communication. The second way is device programmability; the heterogeneous programming languages allow the same language for different device architectures, but even with the modern languages as SYCL [138], the code is functionally portable while performance is not. Then, the implementations and optimizations are different among devices, especially in FPGAs.

FPGAs offer high energy efficiency and low latency compared with CPU and GPU, but entails new challenges in its inclusion as an accelerator due to the long design flow and early support [29]. For this reason, the FPGAs need to be studied in deep detail.

2.2 Heterogeneous System

The Dennard Scaling predictions stopped in early 2000's since it ignores two effects on submicron technologies that condition the current processor development. First, leakage current, which becomes more significant with the reduction of transistor size; second, also ignored, was the threshold voltage at which the transistor switches. Combining these effects, a higher number of transistors per Moore's law [108], increases the power density. As a result, more power needs complex and expensive power dissipation. To make things worst, the dynamic current depends on clock frequency, which was one of the main sources of improvement in the processor in Moore's era.

With the ending of Moore's era in computing, many solutions appear as the multi-core [89, 156], dark-silicon techniques [40] and, the field of our interest, heterogeneous system [78]. The inclusion of accelerators has proven to be a feasible solution; for example, in the Top500 list of supercomputers of June 2021 [153], seven of the top ten list are heterogeneous systems, and in terms of power efficiency in the Green500, the number one is the MN-3 [152]. This heterogeneous architecture combines Xeon CPUs with MN-Cores for deep learning applications. MN-3 is also on the TOP500, but falls to 335th position, showing the difference in performance objectives and power efficiency.

The idea behind a heterogeneous system is the inclusion of accelerators with different specific architectures, more specialized to an application field. When an application is suitable for the processor device, an improvement of performance-per-watt is expected. Multiple devices in a system provide the opportunity to utilize concurrency and parallelism, offering the benefits of flexible, well known, and strong optimized tools of CPU devices, with greater energy efficiency from dedicated hardware. Nowadays, CPUs with multicore processor have high performance supporting out-of-order execution, branch prediction and improved SIMD instructions with up to 16 operations on 128 bits of data in one clock cycle [7]. When a CPU alone is sufficient for a target application, the heterogeneous execution would be a waste of power and performance [52]. Offload processing to an accelerator adds communication overhead caused by the limited bandwidth available among devices. Also the CPU acts as a host of the devices, adding control overhead. Choosing the right combination device-application is a challenge and requires a good understanding of accelerator features.

One of the most popular accelerators are the graphic processing units (GPU). GPUs are a many-core architecture with several processing units, following the SIMD(Single Instruction Multiple Data) paradigm. The GPUs were initially oriented to process computer vision algorithms, however, with the evolution of programming languages, this devices are widely used in many other applications. The GPUs can be discrete, connected through a PCIe or NVLink, or it can be embedded with the CPU. Although a tightly coupled GPU improves the data transfer cost, it has fewer processing units because it has less available area, limiting performance [175].

Other devices that have become popular in the last decade are the FPGAs, which were initially used as accelerator in datacenters [14, 134], FPGA offers dedicated hardware dynamically reconfigured with low latency and present one of the most relevant characteristics: high energy efficiency. As GPUs are specialized for data-parallel application, FPGAs fit with streaming data parallelism [67], but they are less common than GPUs since they are more difficult to program.

Aside from the aforementioned devices, there are many other accelerators such as the DSP (digital signal processor) and ASIC (application-specific integrated circuit). Those target specific niches of applications and are less versatile.

Although accelerators are designed for high performance, if they are difficult to program, there is more risk of programming bugs that slow their adoption by the programmers' community. For this reason, programmability is almost as important as high performance. For GPUs, a set of programming languages are available such as OpenMP, OpenCL, and CUDA. Specially CUDA (Compute Unified Device Architecture) from NVIDIA transforms the adoption of GPUs as a general purpose processor easing programming with a set of compilers, tools, libraries, and applications [54]. In FPGAs the programming is more difficult and traditionally uses the RTL design. Similarly to CUDA, for FPGAs emerges the High-Level Synthesis with languages as C/C++, OpenCL, and SyCL to speed up the hardware development[138].

Hardware heterogeneity entails diversity in programming languages, forcing developers to learn unfamiliar tools per each device. Thus, the goal is to hide heterogeneous system complexity and improve programmability. Software abstraction can put away the complexities by specifying frameworks and runtimes to more fully exploit the computing capabilities of each device in the heterogeneous system [52]. Furthermore, the portability of programming languages is one of the most promising features of programming frameworks, where the idea is defining the compute

kernels to be executed on the devices and also the host code that orchestrates the execution [138].

Industry is one of the main stakeholders in the development of unified programming languages and standards for heterogeneous systems [36]. One of the initiatives is OpenCL[110], it is a standard framework that promises writing only a single kernel, it can run on a wide range of systems, maximizing the portability and with heterogeneous computing in mind [42]. However, all these features come with tedious host code, and the burden must be assumed by the programmer and the runtime. Also, OpenCL cannot mask differences in hardware architectures, as a consequence, programs do not necessarily run at peak performance, and, hence, each program has to be tuned for each architecture [148].

Even although OpenCL is an advance in programming, the exposition of many low level details makes difficult their extended adoption for average programmers [37, 34]. Other high-level frameworks are developed to increase productivity such as CUDA [13] for Nvidia GPUs and OpenACC [126] a more general standard. OpenACC follows the OpenMP philosophy, the applications are migrated to heterogeneous systems inserting compiler directives, reducing the programming migration effort [47, 149]. OpenCL and OpenACC are the independent programming standards to target accelerators [77]. This was valid until 2014 where the Kronos group, the same that proposed OpenCL, published SYCL standard [138] and the first implementations were spread in 2019. SYCL brings OpenCL compute capabilities to C++. It has grown quickly since SYCL integrates high level libraries and is being broadly adopted in many architectures. At least every year, the research community contributes with the framework advances and compilers for heterogeneous platforms such as: OmpSs [37],Halide [136] , VirtCL [173], SnuCL [87], EngineCL [120], . . . In general high level of abstraction provides programmers the ability to orchestrate the available devices to choose or share the workload among the available ones.

The device orchestration, typically called load balancing, is based on the concept of use all the compute capabilities available in the computer system to avoid the waste of compute and power resources. This task is not straightforward since the performance of each device depends on the program requirements. Conventionally, the programmer should know the benefits of each architecture and then choose the right platform to run the application. This problem could have many answers and depends on the performance target (time, power, ...)[39]. The Table 2.1 shows

Table 2.1: CPU, GPU and FPGA characteristics as accelerators in a heterogeneous system, classifying in three categories: best, worst, and intermediate.

Characteristic	CPU	GPU	FPGA
Programability	●	◐	○
Easy optimization	●	◐	○
Flexibility	○	◐	●
Transfer overhead	●	○	○
Compute latency	○	◐	●
Massive Parallel	○	●	◐
High Branch Divergence	●	◐	○
Energy Efficiency	○	◐	●

[n] ●=The best; ◐=Medium; ○=The worst

a summary of the main characteristics of the CPU, GPU, and FPGA to expose the diversity and the multi-objective optimization problem for programmers. Some proposals enable CPU and GPU co-execution, [94, 96, 120, 79, 17, 116] others CPU and FPGA co-execution [133, 10, 37, 53, 141], some others combine GPU and FPGA [160], but no so many are already prepared to include more heterogeneity with CPU, GPU and FPGA devices [33, 139].

Current frameworks concentrate their efforts in the promise of single-source programs to be run on the accelerators and also on the host that orchestrates execution [138], unfortunately the achievement of this objective is still a long way off, specially in FPGAs devices which demands more knowledge in architecture design to get the desired best performance per watt.

In the next section, we describe in detail the FPGA, as it is the most relevant device used in this dissertation. We discuss its architecture and its integration as an accelerator conceived to work with heterogeneous languages.

2.3 FPGA Device

The field programmable gate arrays (FPGA) are today a powerful platform technology thanks to the increase of the number of computational units along with the

number of levels in memory and the processor speed that has occurred in recent years. One of the most significant changes in this time is in the inclusion of the intellectual property (IPs) for digital signal processing (DSP) with floating point operations and the inclusion of modern external memory technologies [168]. The FPGAs are in the middle-ground between general purpose processor(CPU) and pplication-specific integrated circuits (ASICs), often providing better performance per watt in comparison with a CPU, but can be 35× larger and 4× slower than an ASIC implementation [12].

FPGAs are mainly attractive for the programmable routing, which provides a highly flexible architecture for application-specific customized hardware and also allows continuous hardware upgrades. They are composed by columns of reconfigurable logic blocks, DSP blocks, and scalable internal memory [70]. Also, the architecture includes hardware IP units to support peripheral interfaces as: PCIe bus standard, general purpose I/O and external memory. As an example, the Figure 2.1 shows the Intel Stratix FPGA family architecture [71].

The reconfigurable logic core is composed of Logic Array Blocks (LAB) with basic logic blocks known as Adaptive Logic Modules (ALMs). The ALMs contain Look Up Tables-bases (LUT) resources that can be divided between two combinational adaptive lookup tables (ALUTs) and four registers, which can be reconfigured to build logic functions, arithmetic functions, and register functions [70]. Several LABs can work cooperatively through the interconnection network to implement large logic functions.

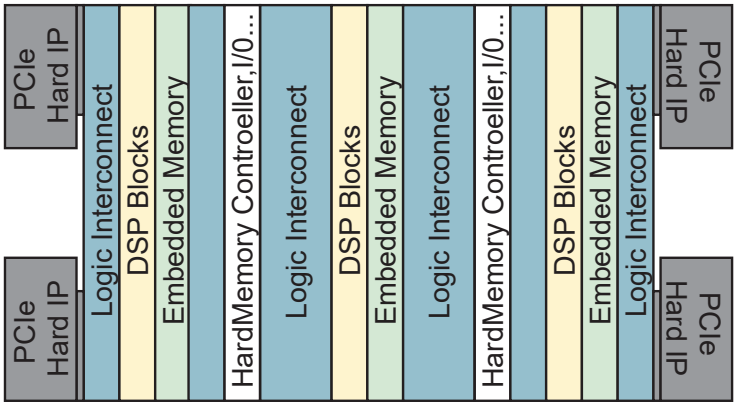


Figure 2.1: Intel Stratix 10 internal block diagram architecture.

Table 2.2: Intel FPGAs used in board accelerators.

Device	Launch Date	Transistor size	Logic Elements	DSP blocks	Embedded memory	External memory
Stratix IV 530	2008	40nm	531200	1024	27,3 Mb	DDR2
Stratix V 5SGXA7	2010	28nm	622000	256	57.2 Mb	DDR3
Arria GX 1150	2013	28nm	427200	1518	67.2 Mb	DDR4
Stratix 10 GX 2800	2013	14nm	2753000	5760	244 Mb	DDR4
Stratix 10 MX 2100	2017	14nm	1073000	3960	239.5 Mb	HBM2
Agilex F014	2019	10nm	1437240	4510	190 Mb	DDR4

The performance and energy efficiency capabilities of FPGAs accompanied by programmability changed around the year 2004 with the improvements in High-Level Synthesis had attracted the attention of the programmer's community. This impulsed FPGA adoption from embedded fields using the FPGA like a programmable interconnection technology, to high performance or power-efficient applications, making it part of the system as a hardware accelerator. As an accelerator, the FPGA can be embedded in the SoC tightly coupled to the CPU, or can be a discrete device with PCIe communication and external memory for more data storage.

Discrete FPGAs offer higher compute performance per chip area than embedded ones, fitting better in the HPC domain applications. The evolution of FPGAs acceleration cards started with the Stratix IV from Altera manufacturer. This board is part of Stratix series for high-end performance at low cost. In this devices, one of the first implementations using OpenCL was reported [31, 6, 109], and comparing the performance per watt rations with CPU and GPU, the Stratix IV was around 5× better [15]. The Stratix V FPGA emerged with transistor size reduction and partial reconfiguration features in 2010 with a 28nm technology [72]. Until 2012, Stratix V was the largest device platform with OpenCL support and evidenced the difference of kernel optimization with other devices, centering the efforts in models and optimization methodologies [50, 162, 53] and acceleration of specific applications [163, 164, 183, 113]. A low end FPGA from Arria family was introduced for discrete boards in 2011 with 20nm technology, it was the first FPGA with hardwired floating point capabilities [12]. Arria family was oriented to data structures applications [86], image processing[151], and artificial intelligence [8, 177]. As in other platforms, the main worries in the application development was the methodology to achieve

a good performance with OpenCL in specific applications [183, 185, 75, 160, 184, 143, 48]

A modern FPGA from the Intel Stratix series is the Stratix 10 with 14nm transistors. Its main novelty was the introduction of the hyper register architecture for high-speed designs [69], achieving a maximum frequency of 500MHz for HLS designs. Also, Stratix 10 MX model was the first FPGA with the modern High Memory Bandwidth (HBM2). The HBM2 is tightly coupled to the die and their interface is composed of multiple banks as Figure 2.2 shows, each HBM bank is completely independent with a 128-bit data bus, operating at DDR data rates. The inclusion of HBM2 memory suppose an increasing of bandwidth and memory I/O interfaces with more power efficiency compared with DDR3/DDR4 memory [61, 107]. Theoretically, in terms of performance Stratix 10 can offer up to 9.2 TFLOPs being this technology near to GPU devices in deep neural applications, compared with the Titan X Pascal with a peak of 11 TFLOPSs, meanwhile Arria 10 offers 1.36 TFLOPSs [123]. The main application focus, as in other areas, is neural networks and stencil computations [23, 41, 185].

As mentioned in the previous section, one of the greatest limitations of accelerators is the communication with the host. As a solution, accelerators should increase the bandwidth among devices. One alternative in discrete FPGAs is to increase the PCIe bandwidth as in Intel Agilex (2019), supporting 16× PCIe 4.0, Agilex FPGAs are designed to provide efficiency in AI applications supporting Bfloat operations in DSPs [63]. On another side, the integrated FPGAs into heterogeneous SoCs can solve

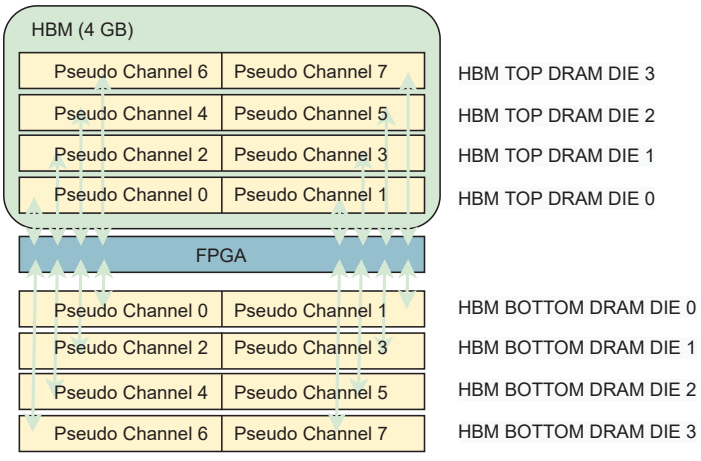


Figure 2.2: FPGA with an HBM memory as Intel Stratix 10 MX with 32 pseudo-channels and 8 GB of memory across top and bottom interfaces.

bandwidth issues [20]. Such FPGAs have a processor (ARM Cortex or Intel Xeon) tightly coupled to the FPGA in families such as Stratix 10 and Agilex. A resume of the main FPGA models used in acceleration boards is listed in Table 2.1.

The other main FPGA vendor, Xilinx, uses the same concept of partial reconfiguration for kernel and a static region to implement the communication structure with the host for OpenCL. It offers threes device families with the SDAccel tool to use OpenCL: Virtex-7, Kintex-7, and Kintex-Ultrascale [145]. Virtex architecture. Similarly to Intel, the Kintex is a low end FPGA with a focus on price/performance, instead the Virtex family is HPC oriented, it was the industry's first with 18×18 hard multiplier blocks [12]. Furthermore, both vendors share the need of performance models [21, 94] and specific optimizations per application field [83, 137].

2.3.1 FPGA Program Design

As the architecture features, the programmability in FPGAs is crucial and is considered one of their main drawbacks. FPGA design flow is large and complex compared with another processor because of the architecture flexibility.

The main steps of FPGA design comprise [90]:

1. Functional design, traditionally with programming models centered on register-transfer level (RTL) or more recently with High level languages.
2. Synthesis, the design is translated into circuit elements as a netlist.
3. Place and route, this is the implementation performed by the FPGA vendors where the design is mapped on the available resources following the clock and pin assignment conditions.
4. Implementation, the generated bitstream is programmed on the device.

At the end of each step of the design flow, verification and simulation should be done.

From a programmer's perspective, the design creation of the microarchitecture is the most time demanding. Programming with hardware description languages (HDL) such as Verilog and VHDL provides programmers the flexibility to make low-level design decisions that can produce high-quality results, however, they are extremely time consuming [26]. In the programming flow, the functional verification needs

a software model and a hardware test bench to simulate and compare with the software model.

The increasing complexity and hardware resources on FPGA have encouraged the evolution of design abstractions to improve the programmers productivity [99, 95]. The introduction of High-Level Synthesis for FPGAs in commercial tools started in early 2000's. It can often reduce the design effort with more friendly high level languages, from C-extended languages to more high level abstractions to support heterogeneous systems with frameworks as OpenCL, and more recently, over OpenCL, the SyCL framework. The main source of gain using the HLS design flow is the capacity to make faster functional verification compared with the traditional RTL design methodology. With High-Level Synthesis the same program for FPGA is the software test bench, then the verification time could be reduced as the tool design flow for Intel FPGAs in the bottom of Figure 2.3 shows. The compilation details for HLS are part of the section 2.3.2.

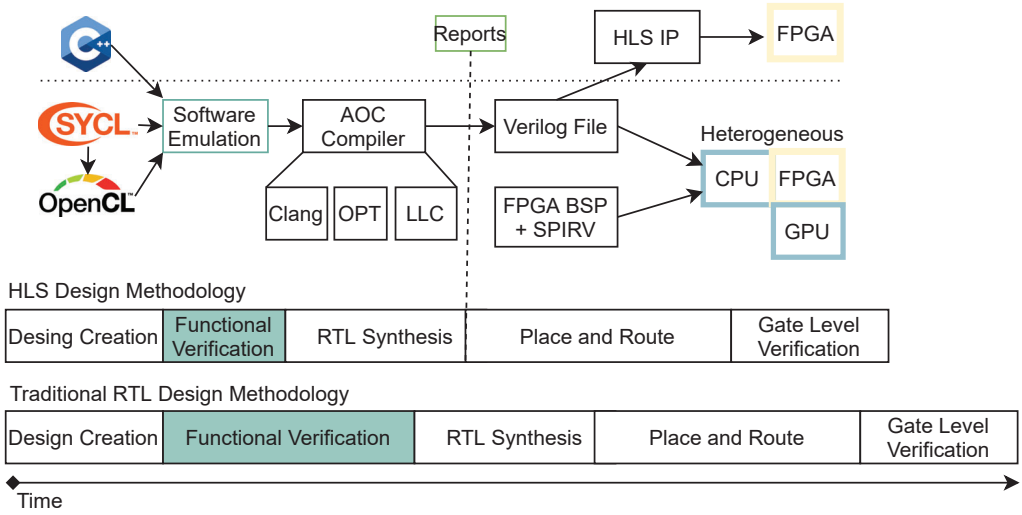


Figure 2.3: HLS synthesis tool flow for Intel FPGAs. The lower part compares the timing effort between HLS and traditional RTL design methodologies.

With High-Level Synthesis tools, the programmers can save development time around an order of magnitude because the implementations are more compact, less error prone, and require less hardware expertise. For hardware engineers, High-Level Synthesis tools allow to quickly explore the design space, crucial in the design of complex systems, reducing the time to market [114]. For this reason High-Level Synthesis has been embraced in the programmers community of High

Performance Computing (HPC) to develop applications with high performance and power efficiency with FPGAs devices as accelerators.

2.3.2 FPGAs as Accelerator on heterogeneous system

Discrete FPGAs offer high compute performance and match the HPC compute necessities evolving to accelerators. Three parts are identified to couple the FPGA to a host CPU:

1. A layer to support and control the communication with host, and external data storage to increase memory capacity typically required in HPC applications.
2. The application algorithm.
3. The host drivers to support the FPGA transactions.

The partial reconfiguration in FPGAs is an important feature that contributed to the communication and driver support of the FPGA and CPU. Different vendors such as Intel, Bittware, Terasic, and others, provide to programmers a Board Support Package(BSP) which is partially reconfigured on the device in logicLock region in the floorplan for host support with heterogeneous languages as OpenCL and SYCL, including drivers and protocols for communication and programming. The logic region reserved on the chip for the BSP, as for example, in an Stratix 10 FPGA in the Figure 2.1 covers the left side of the FPGA, in gray, for PCIe2 x8 support, and part of the memory controllers, in white. In FPGAs with HBM support, the memory controllers are allocated at the top and bottom of the chip. The free area of the chip is free for programming kernel applications and BSP interconnection.

With the lower hardware layer supporting High-Level Synthesis, the application design requires compilation tools as the developed by the Intel FPGA SDK. The AOC compiler, as in other High-Level Synthesis frameworks, rely on LLVM. It uses CLANG front-end to parse the high level code extensions to produce un-optimized LLVM IR, the middle-end performs optimizations with compiler passes. Finally, on the back-end, the compiler instantiates Verilog IPs [166]. In this step, the designer can analyze the hardware blocks to be generated during placement and routing stages. After Verilog transformation, two ways are possible, as Figure 2.3 shows. The first is to generate hardware IP Blocks from C/C++ code that can then be interfaced to the BSP later, as in the OpenVX proposal in Chapter 5. The second path is to generate

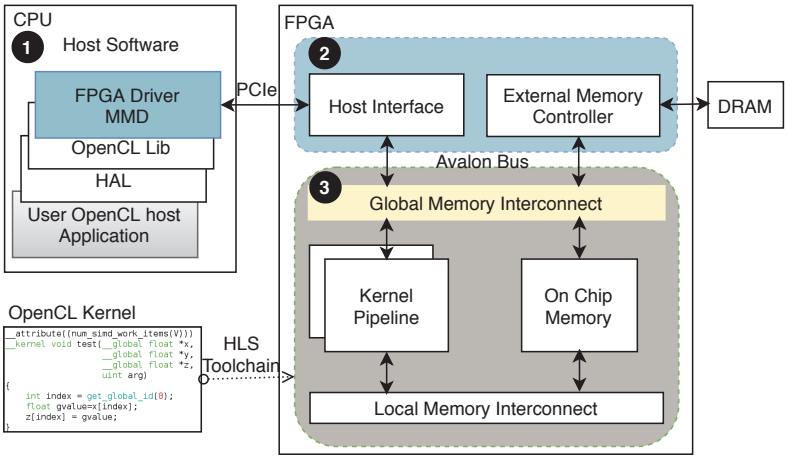


Figure 2.4: Main elements of OpenCL BSP for FPGAs. ❶ and ❷ represent the BSP, and ❸ the kernel logic.

kernels directly coupled to the BSP to use the FPGA as an accelerator, such as the kernels synthesized with OpenCL and SyCL.

The BSP and the kernel pipeline composed an FPGA accelerator, and internally they have their own architecture and components described in detail in the next subsection 2.3.2.

Kernel pipeline and Board Support Package

The kernel pipeline is part of the total architecture of the FPGA as accelerator, since OpenCL was the first heterogeneous language used for programming FPGA and it is the most extensively used. In this section, as in the Figure 2.4, we present the main components of an OpenCL application using the Intel FPGA SDK for OpenCL as a reference. Without loss of generality, this flow also represents other toolchains (e.g., SYCL). On the host side, ❶, the application communicates with the FPGA device through the board support package [1] (BSP, in blue on the figure). A BSP implements the lower layers of the application stack such as the memory-mapped device (MMD) library, performing the basic I/O with the board, and the PCI express (PCIe) communications. On the FPGA side, the BSP, ❷, provides support to communicate back with the host, and with the device memory, DRAM and external devices.

[1] Manufacturers often provide BSP, but advanced users can tune and re-implement them.

From a programmer's perspective, the most important component in a BSP is the kernel logic, ❸, which corresponds mainly to the compiled OpenCL kernel. In fact, programmers seldom need to generate a new BSP[2]. The compilation process consists of two main steps (without lost of generality, OpenCL is used as example). First, a translator generates HDL code from the OpenCL, and second, a synthesis tool generates the bitstream. The translator creates four blocks from the code: local memory interconnect (LMI), on-chip memory, the kernel pipeline, and global memory interconnect (GMI). The last two blocks are the ones that most critically affect performance, and therefore, they are described as part of the study of this dissertation in Chapter 4.

The major vendors of FPGAs, Xilinx and Intel, have developed their HLS compilers to generate pipeline architectures. The main advantage of this strategy is the amount of parallel operations that are executed per clock cycle, splitting the operating with registers, avoiding computation stalls. Splitting up the processing into small pipeline stages also helps to reach higher frequencies, improving kernel and memory performance [138].

In terms of parallelism, the deeper the pipeline is, the greater the number of items that can simultaneously advance. As a result, the FPGA performance mainly depends on two pipeline characteristics: initiation interval and frequency. Initiation interval is the number of clock cycles between two consecutive kernel work-items start, and frequency is the cycle time of the longest pipeline stage.

Although with HLS the programmers uses common languages for CPU and GPU devices the optimization and programming patterns [162, 106] are different. FPGA kernels require detailed annotations to guide the hardware generation, and since the synthesis process is time expensive, the optimization should be done in an intermediate stage, after Verilog generation, called "intermediate compilation". In this phase the designer can analyze the hardware blocks to be generated during place and route stage. Depending of the compiler heuristic and together with the fact that the vast majority of software is not designed for FPGA synthesis [95], makes HLS difficult to adopt. For this reason, the "intermediate compilation" based on reports, is one of the noticeable advances in HLS helping to improve productivity allowing to analyze the generated IP blocks.

[2]Also note that HLS tools would always require a BSP to compile an OpenCL kernel for supporting the aforementioned low-level tasks.

3 Experimental Framework

This thesis comprises several studies that were carried out with multiple heterogeneous systems using several programming models, such as OpenCL or OpenVX, running on a host CPU with accelerators, mainly FPGAs and, in some cases, in combination with a GPU.

This chapter presents a general overview of the software and hardware frameworks together with the general methodology used for modeling, running the experiments, and computing relevant metrics such as throughput, portability, and load balancing. Since the results are extracted from different compilation stages in the FPGA compilation flow and high level frameworks, specific details change and are detailed in each Chapter's methodology section.

3.1 Evaluation Hardware

The heterogeneous systems used in this work consist of a host CPU connected to one FPGA, or one FPGA plus a GPU device as Figure 3.1 shows. In these systems, the devices are connected to the CPU through an external PCI-express (PCIe) bus, and they are common in HPC computing systems.

Four heterogeneous systems have been set up to carry on all the experiments. Table 3.1 shows their configurations based on three host CPUs, listed in Table 3.2,

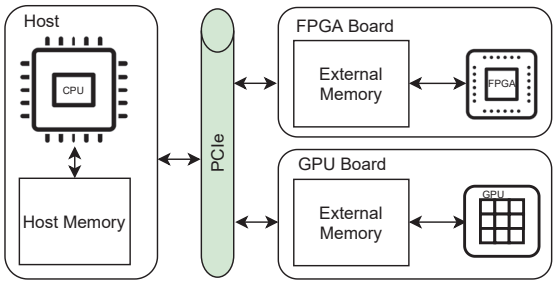


Figure 3.1: Heterogeneous system with an FPGA board and a CPU.

Table 3.1: Device combinations used as heterogeneous system in this thesis. The host, FPGA and GPU capabilities are described in Tables 3.2, 3.4, and 3.3.

	FPGA Board	GPU Board	In Chapter
Host 1	DE5-net	TITAN X	6
Host 3	Stratix 10 GX	-	4 and 5
Host 2	Stratix 10 MX	-	4
Host 3	Stratix 10 MX	-	5

combined with three FPGA boards in Table 3.4, and one GPU in Table 3.3. The CPU only acts as a host in Chapters 4 and 5, while in Chapter 6, it is considered as another device.

Communication is a key factor in heterogeneous systems' design, and it is the main reason for the multiple combinations used in this work. The PCIe bus allows to connect diverse device architectures through a high-speed serial bus commonly used in motherboard interfaces. Although CPUs and motherboards offer many PCIe port expansions, not all are available to match the maximum bandwidth capabilities of each device. In fact, the first combination used for load balancing with three devices (Host 1, Table 3.1), connects both the FPGA and GPU with 8 lanes, even if the GPU supports 16 lanes, because of the CPU-chipset-motherboard ensemble limitations.

Upgrading the FPGA or the host may not always be enough to overcome these communication issues, and two factors can still prevent to reach the maximum bandwidth. First, as FPGA board increase dimensions, they require a larger space in the motherboard, and depending on the board layout, they can block required PCIe ports to connect other accelerators. Second, the BSP and OpenCL drivers support is limited in some devices. For example, in order to improve device communication performance, the Host 2 was configured with an AMD processor, but this processor had incompatibilities with Intel FPGA PCI drivers and the PCIe OS support should be emulated, reducing the PCIe throughput. Also AMD processors are not yet officially supported by Intel OpenCL SDK.

Host 3 aims for full compatibility with Intel FPGA devices using a CPU manufactured by Intel. This system has more PCIe capabilities with 4 PCIe ports with 16 lanes, but since motherboard implements switch/multiplexation of CPU PCIe lanes, the

Table 3.2: Main characteristics of the three host systems.

Feature		Host 1	Host 2	Host 3
CPU	Manufacturer	Intel	AMD	Intel
	Model	i7-6700k	Ryzen 1920X	Xeon Bronze 3204
	Cores	8	12	6
	Launch date	2015	2017	2019
	Frequency	4 GHz	3.4 GHz	1.9 MHz
Mother-board	Manufacturer	ASUS	Micro-Star	Supermicro
	Model	Z170-PRO	X399 SLI PLUS	X11SPA-TF
External Memory	Type	DDR4	DDR4	DDR4
	Bw. per bank	17.1 GB/s	17.1 GB/s	17.1 GB/s
	Capacity	64 GB	96 GB	96 GB
	Banks	4	6	3
	Connector	DIMM	DIMM	DIMM
OpenCL	Version	2.0	-	2.0
OS	Distro	CentOS 7	CentoOS 7	CentoOS 7
	Kernel	3.1	3.1	3.1

OpenCL FPGA driver does not work [1]. The driver needs direct interconnection to the CPU processor and only one slot position is directly connected, the rest has a switching chip, leaving available one slot for FPGA, limiting the addition of more heterogeneity with more FPGA devices.

On the device side of the heterogeneous system, we used a TITAN X GPU from NVIDIA. This GPU has a Maxwell architecture manufactured with 28 nm technology, and Table 3.3 shows its main features. Compared with the rest of used devices used, this GPU is the most powerful in terms of raw floating-point compute power with 6.6 TFLOPS in single precision given by this equation $Frequency \times Cores \times 2$ where 2 are the multiply-add operations effectively support for floating point unit (FPU) in each CUDA core. The reported bandwidth is limited by host PCIe lanes and was measured with CUDA bandwidth test. The driver support is highly reliable since its a high end product that has been in the market for a while. It provides CUDA and OpenCL support with runtime compilation of kernel programs.

[1] A personal communication with Intel engineers confirms the PCIe driver limitations

Table 3.3: Characteristic of NVIDIA GeForce GTX TITAN X GPU board.

Feature		GeForce Titan X
GPU	Manufacturer	NVIDIA
	Cores	3072
	Frequency	1.1 GHz
	Peak Floating Point	6.6 TFLOPS
External Memory	Type	GDDR5X
	Bandwidth	480 GB/s
	Capacity	12 GB
PCIe	Bandwidth	6500 MB/s (8 lanes)
	Lanes	16
	Gen.	3
Board	Dimension	2-slots
OpenCL	OpenCL Version	3.0

The other device coupled to the host has been several different FPGAs. Namely, in this work, we have worked with three FPGA board models that were upgraded during the development of this thesis. All of them belong to the Intel Stratix family: Stratix V, Stratix 10 GX and MX. The details of each board are shown in the Table 3.4. The PCIe bandwidth was measured with the board test of each BSP board. The FPGA models evidence the evolution of the FPGA boards, increasing PCIe bandwidth 4 $\times$, the amount of DSP resources 22 $\times$, and memory bandwidth 16 $\times$, comparing the best and worst FPGA features. The FPGA peak FLOPS in single precision are calculated as $DSP\_Frequency \times DSP\_blocks \times 2$ where 2 is the amount of FLOPS per clock cycle in each DSP block at a rated speed of 450 MHz [101].

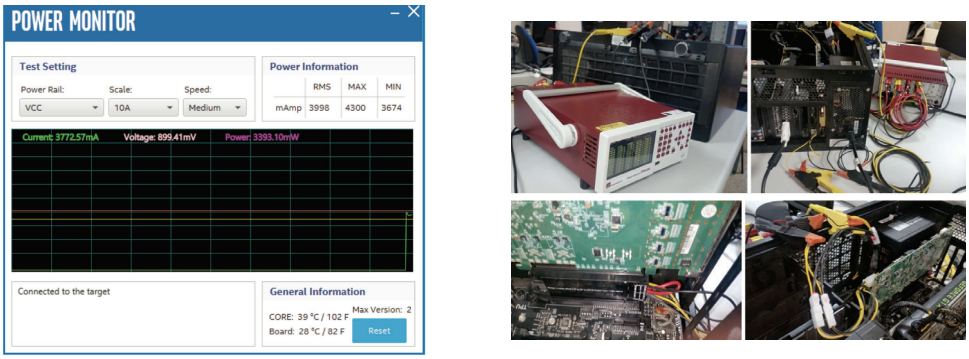
FPGA boards and BSPs to support FPGA OpenCL SDK are development kits designed for evaluation proposes provided by Terasic for the Stratix V board and directly from Intel for the two Stratix 10. For most experiments the BSP wasn't recompiled, but an in-house BSP was compiled for the Stratix 10 GX to boost the DDR4 memory data rate from 1866 to 2666 MT/s, the maximum supported by the hardware. This change required to update timing settings of the memory controller in the static part of the BSP based on the memory speed-bin table in data-sheet [102]. Timing adjustments in BSP required a two-step compilation for custom board design [64]. First, a

Table 3.4: Characteristic of Terasic DE5-net, Intel Stratix 10 GX, and Intel Stratix 10 MX (early version) board development kits.

Feature		DE5-net	Stratix 10 GX	Stratix 10 MX
FPGA	Family	Stratix V	Stratix 10 GX	Stratix 10 MX
	Model	5SGXA7	2800	2100
	Logic Elements	622000	2753000	1073000
	Embeded Memory	57.2 Mb	244 Mb	239.5 Mb
	DSP	256	5760	3960
	Peak Floating Point	0.2 TFLOPS	5.1 TFLOPS	3.6 TFLOPS
	Launch date	2010	2013	2017
External Memory	Type	DDR3	DDR4	HBM2
	Bandwidth	22.3 GB/s	14.9 GB/s	350 GB/s
	Capacity	4 GB	2 GB	8 GB
	Banks	2	1	32
	Connector	SO DIMM	HiLo	Embedded
PCIe	Bandwidth	2880 MB/s	6112 MB/s	6191 MB/s
	Lanes	8	8	8
	Gen.	2	3	3
Board	Dimension	1-slot	2-slot	2-slot
OpenCL Support	OpenCL Version	1.0	1.0	1.0
	Quartus Version	17.1	18.1/19.4	19.3

flat compilation is performed of the entire design with the `aocl -bsp-flow=flat` command to get a timing clean revision with a frequency near to 450 MHz and full-filing PCIe and memory timing constrains. The second step is performed a base compilation of the adjusted static BSP with the LogicLock restrictions using the `aocl -bsp-flow=base` command to get the base.qar file, which is the static region. Finally, the resulting file is imported in future kernel compilations within the standard compilation flow. The new BSP with a DDR4-2666 support was used to evaluate and compare the model in Section 4.7.3. Besides, the new BSP reaches a maximum frequency of 441 MHz, enough for the memory controller and PCIe communication. Comparing the original Intel BSP and our modified version with the board_test kernel provided by Intel for BSP tunning shows an speed-up of 48 % in memory transactions.

To finish, we describe the employed methods to measure power in CPU, GPU and FPGA devices. For CPU we used Intel RALP counter [74] to measure CPU package and RAM power with a default sample rate of 1000 samples per second. Nvidia provides the *nvidia-smi* tool that supports power measurement with a sampling period of 1 second. Finally, in FPGA, two power measurement techniques were used in this thesis: First, internal chip measurements using the power monitor provided by Intel for each FPGA board as Figure 3.2a shows. This tool measures the chip current and voltage at the power supply, including for the FPGA logic core rail (with power supply of 0.9 V) and the others ports (with power supply of 3.3 V) and I/O interconnection. The sample rate is of 1 second, and we ensure at least 5 samples with multiple kernel executions. This is the method used in Chapter 5. The second measurement method, used in Chapter 6, is invasive in the two power sources. First one is the voltage and current supply of the FPGA board from PCIe port, with a PCI-raiser board. And second one is the external power supply, through the 6-molex pin connector. Both are connected to a Newtons4th PPA520 power analyzer as Figure 3.2b shows. The sample rate with this method is 10^6 samples per second.



(a) FPGA power monitor.

(b) FPGA invasive power measurement.

Figure 3.2: FPGA power measurement methods.

3.2 Software Framework

OpenCL is the base programming model that can glue the GPU, FPGA, and CPU in a heterogeneous system and it is the underlying driver and runtime for recent frameworks as Intel DPCPP(SyCL) and the frameworks developed in this thesis. For the sake of simplicity, we only discuss the hardware and kernel programs in the devices which correspond to the *Kernel programming model* of the OpenCL

specification. This model allows the management of the physical resources available, expressing the parallelism to map programs efficiently in a variety of devices such as out-of-order superscalar pipeline, a massive parallel or re-configurable architectures represented by a CPU, GPU, and FPGA respectively [57]. The OpenCL kernels are syntactically similar to C syntax and can be compiled at runtime in the CPU and GPU cases or offline as is the FPGA case.

FPGA compilation is performed with the Intel OpenCL SDK which creates the FPGA programming file (.aocx) and Intel high level synthesis SDK to create IP/RTL files. OpenCL is the default driver to couple the FPGA with the host, and the programs are linked as binary objects using offline compilation. Figure 3.3 describes the general multi-step compilation for the FPGA [65]. The FPGA report generated in intermediate compilation stages provides the metrics used in this work to evaluate kernel performance in early stages. Html reports show kernel performance parameters such as initiation interval, expected maximum frequency, and latency in each loop of the kernel. The report also contains the system viewer analysis which includes a view of all the kernel program as a graph, organizing the pipeline into the blocks identified by the compiler and also shows memory type (Register, RAM, DRAM) and memory load/store units interconnection with Avalon ports. These reports are the main source of data for the proposed model in Chapter 4 and the OpenVX high-level framework in Chapter 5. Also, the kernel profiling allows tuning and verification of the proposed analytical model.

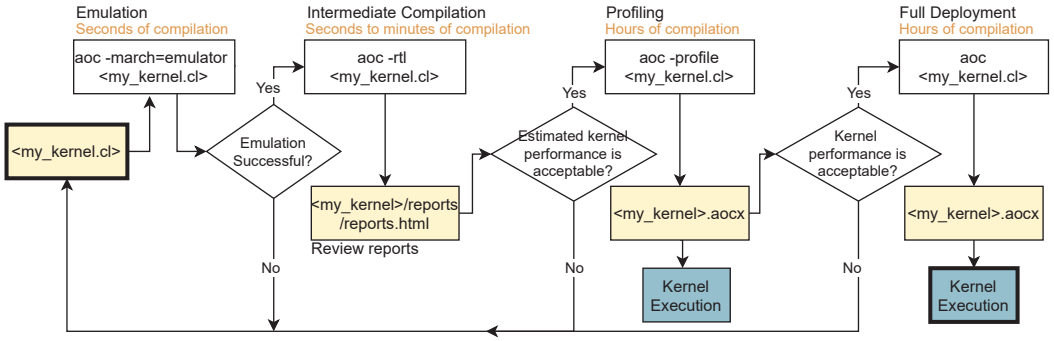


Figure 3.3: OpenCL compilation flow for Intel FPGA board. The box in yellow correspond to generated files, in blue are FPGA execution stages and the rest are the aoc compilation commands in each design stage: Emulation, Intermediate compilation and Full deployment.

OpenCL kernels for CPU and GPU can be compiled at runtime using just-in-time compilation. Kernels execution and device management require a host coordination

for interacting with multiple driver vendors. This task is defined in the *platform model* and in the *execution model* of the standard. The first model is important for program portability, and the second one defines the communication mechanism through command-queues associated with each device, and the events of the execution.

The last OpenCL model is the *memory model* which abstracts the memory hierarchy providing memory ordering guarantees in the parallel execution [78]. In our system, GPU and FPGA devices have separated memories from host devices, and consistency is enforced by synchronization events (barriers, in order queues, and events). The OpenCL execution in this work is always implemented in-order command queue and synchronization events are with blocking commands, this means, the device command queue waits until the kernel is finished. This synchronization behavior is imposed by manufacturer driver limitations.

OpenCL requires the management of many details in each model, but provides enough support for single device execution as done in the proposed model. However, the complexity increases with the addition of more devices in multi-device heterogeneous systems as in the load balancing example in Chapter 6. Increasing the level of abstraction with higher level frameworks helps to alleviate the complexities for programmers facilitating the adoption of less common devices as FPGAs. Due to the implementation of the Intel tools, all the high level frameworks used in this thesis rely on OpenCL to communicate the host with the FPGA.

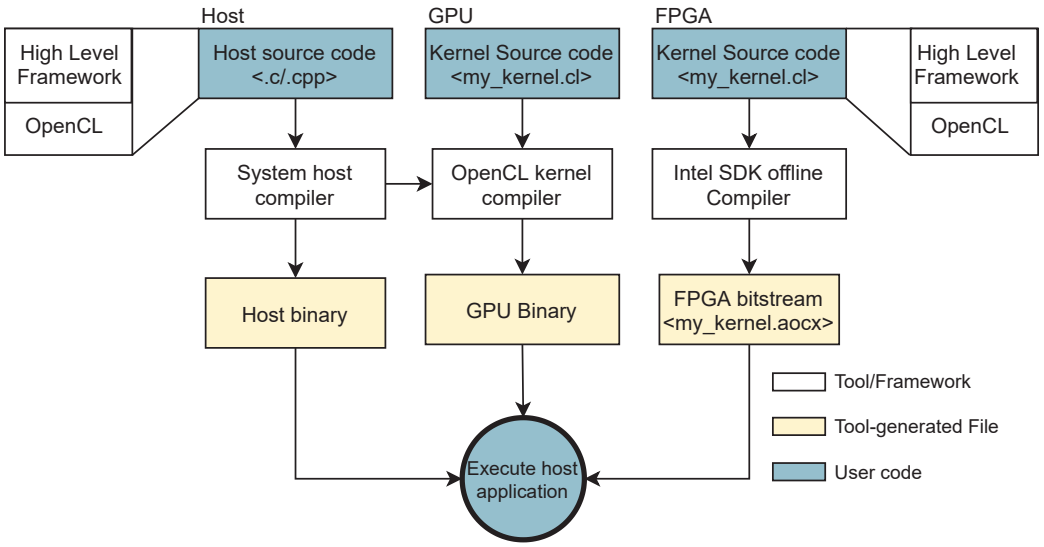


Figure 3.4: OpenCL Flow for a heterogeneous system with a host CPU, GPU and FPGA devices. The host program in this thesis uses two host programming frameworks: OpenCL and EngineCL with less code lines.

4 Analytical Time Estimation of Memory-bound Applications for FPGA Using High-level Synthesis

The memory hierarchy organization is one of the physical elements that can limit the performance of an application, and, for many, the memory bandwidth represents an upper bound on achievable performance. In FPGA applications, the performance is limited by the main memory unless the number of available FPGA resources is exceeded [170, 162]. In this chapter, we analyze and propose an analytical model of the memory access behavior based on memory access patterns HLS-generated by applications because they act as a proxy for application performance conditioning memory behavior from kernel memory request to external memory.

Since placement and routing is the slowest stage generating a bitstream, our model improves programmer's productivity giving a deeper understanding of the memory units and access patterns at early compilation stages that allows to optimize applications. For the sake of generality, we validate the model on DDR4 and HBM memory technologies.

4.1 Introduction

Besides using parallelism, many HPC applications exceed the available memory bandwidth. Although external memory on FPGAs is evolving from external DRAM to 3D-stacked high bandwidth memory (HBM), which has increased bandwidth from 25 to 409 GB/s, the performance of each individual memory bank has only grown up at a 7% annual rate. Comparing this with FPGA resources that have grown at 48% per year [154], the memory wall problem in FPGA applications is evident.

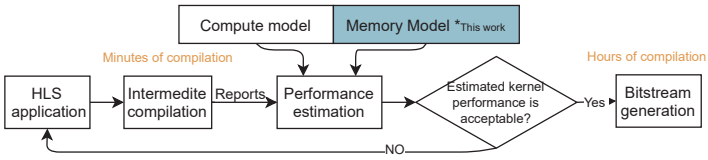


Figure 4.1: Memory model proposal(in blue) in the optimization process of FPGA kernels using HLS.

Exploiting the potential benefits of FPGA technology is a challenge for programmers, even with the development of high level design with languages such as C, C++, or OpenCL [144, 44, 94]. Generating highly tuned code is time consuming, and CPU or GPU optimization techniques are not always suitable for FPGAs. Programmers have two options for easy coding. Either they write well-known code patterns from previous explorations [157, 65], or they rely on pre-synthesis analytical models for estimating performance [183, 162, 21, 94]. These models analyze the RTL code from High-level synthesis (HLS) compiler tools, the high-level code, or both.

Model-based optimization seeking to better exploit FPGA resources and simplify hardware generation focuses mainly on the compute part, or kernel pipeline. Previous studies have oversimplified the organization of global memory interconnect (GMI), which manages memory request between the kernel-pipeline and the off-chip DRAM memory. The lack of detail in the models results in the error seen in two state-of-the-art analytical models [162, 21]. The error is multiplied by 3 when the DRAM specification changes and can be larger than 50% for accesses with data dependencies since those models ignore the differences in memory access and DRAM technologies. Such errors could become more common in future systems because of technological advances to high-memory bandwidth devices.

In this chapter, we analyze the GMI and their interaction with the external memory compiled with HLS tools, which affect the effective memory bandwidth and can significantly impact execution time. Combining information from the analysis of the GMI an their main components, such as load-store units (LSUs), plus the DRAM organizations, enables us to build an analytical model to accurately estimate the execution time. The model mainly requires static information from pre-synthesis reports, Verilog hardware instances, and DRAM memory timing parameters. The Figure 4.1 shows in blue our model and how it contributes in an intermediate compilation stage of the HLS generation. In this stage, the model can potentially reduce the application development time.

4.2 Related Work

Performance modeling of FPGAs using HLS has attracted the attention of many researchers to ease kernel optimizations. The standard tools from the two main FPGA vendors, Xilinx and Intel, help to address optimizations with analytical reports. In Intel case, tools focus only on three performance metrics: Initiation interval, latency, and frequency without execution time estimations.

Existing performance models target one of two different domains: embedded FPGAs [181, 178, 179], usually using C/C++ as the high-level language, or HPC discrete FPGAs with external components such as DRAM memories and PCIe ports. In the latter case, the models are mainly oriented to OpenCL codes [162, 94, 75] and C/C++ [21]. While embedded and HPC models have similarities in the pipeline model, the key difference is the memory system, where the throughput of an internal memory may be 380 $\times$ better than an external one.

In HPC, the memory wall is one of the main limitations of FPGAs for applications. The memory requires a controller to reorder requests to minimize row conflicts, and as a consequence the throughput depends on memory controller implementation [184, 30]. The behaviour of memory controllers is often overlooked [113, 32] or simplified as in the performance model proposed by Wang et al. [162] for Intel OpenCL SDK. It uses a coarse grain model which shows inaccuracies in the memory estimation and requires the extraction of LLVM-IR information that is not provided by the vendor's compiler. In a similar way, the Boyi framework [75] limits the memory estimation to sequential or random accesses with fixed weights, and although this framework is mainly based on memory access optimization, it only evaluates how the OpenCL execution model changes accesses.

Coarse-grained memory models reduce the optimization capabilities on HLS for FPGAs, this problem being detected by the FlexCL framework [94] for Xilinx FPGAs. FlexCL improves models covering memory access patterns with a short CPU/GPU execution, but it continues being the main source of error of the model. As some comparisons show, the memory controller makes differences in the access pattern and hence performance [184, 176, 27, 112]; moreover, CPU/GPU devices have a more sophisticated memory hierarchy that can hide DRAM latency. As well as the memory controller, the memory standard or technology changes the interaction with the FPGA pipeline. For this reason, the inclusion of memory parameters to cover

these technology differences is necessary, DRAM technology being the most widely used. Approaches such as FlexCL obtain latency parameters from modeling the memory latency [21] as HLSCope+ does [19]. Other performance estimators for Xilinx FPGAs include physical DRAM specifications, but these limit access patterns to sequential and random, as a result of their platform experiments; also, HLScope+ includes a correction factor given the lack of knowledge about the Xilinx DRAM controller.

The most feasible source of memory controller behavior is the analysis of Verilog units used by the compiler to generate the hardware controller, as in this study, but often this approach is rejected because of its tediousness [162]. A more user-friendly source is the use of RTL reports, which shows the type of LSUs to assemble a command request to DRAM [65].

The knowledge of LSUs plus DRAM specifications is combined in this proposal to achieve an accurate memory model that can adjust to changes in memory technology from DDR4 to HBM.

4.3 FPGA External Memory and BSP

As it was mentioned Section 2.3.2, host communicates with the FPGA through the board support package performing the basic I/O with the board, and the PCI express (PCIe) communications. On the FPGA side, the BSP, provides support to communicate back with the host, and with the device memory, DRAM and external devices.

The BSP on the FPGA side differs for each FPGA model and each type of external memory, requiring specific intellectual property (IP) controllers and interfaces. For example, for a DDR4 memory with multiple banks, in Figure 4.2a, the BSP uses the Avalon-MM interface and it has a memory bank divider which can support the interleaving of memory banks for one variable, or uses each bank separately. For HBM memory, Figure 4.2b, the interface with the BSP is the Advanced extensible Interface (AXI), and the 32 HBM pseudo-channels have a separate GMI and controller because each pseudo-channel works as independent memory using the "heterogeneous memory" feature of the OpenCL compiler although the technology is the same as DRAM [58].

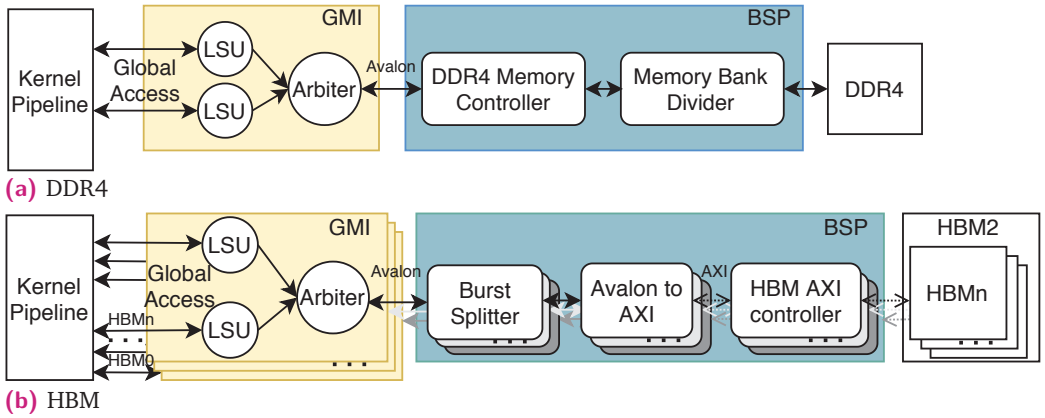


Figure 4.2: FPGA block units for Intel OpenCL SDK with a) DDR4 and b) HBM memory.

The Global Memory Interconnect, from here GMI, is between the kernel pipeline request and the BSP, before to the memory controller. GMI is identified as a critically kernel performance block. Therefore it is described in detail.

4.3.1 Global Memory Interconnect

The GMI manages the kernel-pipeline request to the external memory. In any OpenCL program, each access to a variable in the external memory constitutes a global access(GA). Since global accesses are the main source of kernel stalls, the GMI implements several strategies to maximize external memory throughput and kernel pipeline flow. Architecturally, like other hardware memory interfaces from Intel [59], the GMI has two main components: LSUs, which track in-flight memory operations, and arbiters, which decide on the order of access. Specifically, there are two independent round-robin arbiters one for read and one for write accesses. ¡

Intel FPGA SDK [65, 67] has defined three LSU types for the GMI: burst-coalesced LSU, prefetching, and atomic-pipelined. To understand the access pattern of each LSU, Listing 4.1 and Table 4.1 show the code that generates the third column.

```
1 #define N 1024
2 int random_vector[N]={5,1023, 450, 100, ...}
3 __kernel void test_patterns(  __global int *restrict x, __global
      int *restrict z, const int *cn )
4 {    int i    = get_global_id(0);
5      int out = 0;
6      local int lmem[1024];
7      //Code Snippet form Table I
```

Table 4.1: LSU types and their modifiers in global memory interconnect. The code snippets are from Intel FPGA SDK [65].

LSU Type	Description	Code Snippets[a]
Burst-Coalesced[b]	Requests are grouped into a set of DRAM bursts	
Aligned	Index is contiguous and aligned to page size	`out = x[i];`
Non_Aligned	Index has a modifier not aligned to page size	`out = x[3*i+1];`
Write_ACK	Index to access has dependencies	`out = x[k]; // k is random`
Cache	Index has repetitive dependencies	`for (uint j=0; j<N; j++) z[N*i+j] = x[j];`
Prefetching	Compiled as Aligned Burst-Coalesced	`out = x[i];`
Atomic-Pipelined	Unique LSU for atomic operations	`atomic_add(&x[0], 1);`

[a] Each code snippet corresponds to a define in listing 4.1.
[b] The burst-coalesced type has four modifiers affecting its organization.

```
8       #if defined(Bust_Coalesced_Aligned) || defined(Prefetching)
9           out = x[i];
10      #elif Bust_Coalesced_Non_Aligned
11          out = x[3*i +1];
12      #elif Bust_Coalesced_WriteACK
13          int k   = random_vector[i];
14          out = x[k];
15      #elif Bust_Coalesced_Cache
16          for (uint j=0; j<N; j++){
17              z[N*i+j] = x[j];}
18      #elif Bust_Coalesced_Non_Aligned
19      #elif Atomic
20          atomic_add(&x[0], 1);
21      #endif
22      z[0]    = out;
23  }
```

Listing 4.1: OpenCL Code for access patterns in Table 4.1

The LSUs with burst characteristics groups request before being sent to external memory and requires greater hardware complexity. In Atomic type, contrary to burst, serializes the operation and guarantees atomicity. Note that Each global access in the source code may translate to one or several LSUs, as Section 4 describes.

Each LSU type provides a different maximum bandwidth, the burst-coalesced LSU with an aligned modifier being the most efficient type on DRAM technology because it maximizes effective bandwidth utilization. Figure 4.3 shows a read operation generated by a burst-coalesced LSU. Each LSU has a coalescer unit that tries to group continuous memory addresses into a single burst DRAM operation. Next, the read

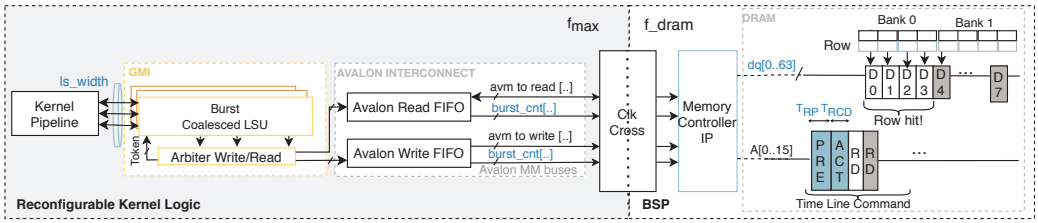


Figure 4.3: Simplified model of a read operation in a single DRAM bank with an Burst-Coalesced Aligned LSU. The parameter names in blue are used in the model in Table 4.2.

arbiter dispatches this operation to the Avalon Interconnect FIFO in order to issue a DRAM access to the Memory Controller IP through the Avalon Bus. The benefits of bursting come from the DRAM organization [180] because during a read operation at least three commands are required: precharge (PRE), activate (ACT), and read out (RD). PRE opens a row in every bank; ACT then opens a row in a particular bank; and RD reads the burst out back to the controller.

When an LSU receives a requested address, it attempts to group consecutive addresses into a burst, the *burst_cnt* bus size defining the maximum number of burst requests at compilation time, because contiguous access to memory enables the overhead of PRE/ACT commands to be hidden.

In a burst-coalesced LSU, three counters trigger a request to the DRAM: 1) the Burst_cnt bus, that usually corresponds to memory page size, 2) the maximum number of threads allowed to be coalesced, and 3) the time out to minimize stalls in the kernel pipeline when consecutive requests cannot be coalesced. The compiler can modify this LSU depending on the memory access pattern and other attributes [65]; e.g., in the case of data dependencies, the compiler infers a write-acknowledge LSU (ACK) with a work-item level coalescer.

In a *Prefetching* LSU, the behavior is similar to that of a burst-coalesced LSU since it has a continuous access to external memory, but loading data to a register or RAM anticipating a large amount of data. For write operations, it uses a burst-coalesced non-aligned LSU. In high-end FPGAs, such as Stratix 10, the prefetching LSU is not available; then, the compiler generates a burst-coalesced LSU even with exactly the same code as that the Intel SDK provides for the *Prefetching* LSU [1].

The last type of LSU is the *Atomic-pipeline*; Intel provides limited support for 32-bit integers and it does not fully conform with the OpenCL specification version

[1] Our assumption is that this behaviour likely depends on the OpenCL SDK version.

1.0. *Atomic-pipeline* is considered one of the most expensive functions which might reduce kernel performance and increase the amount of hardware resources, but its usage can simplify a kernel design [76]. In FPGAs with “heterogeneous memories", this LSU is not available.

4.4 Performance Estimation for FPGAs

The kernel pipeline and the external memory accesses directly impact application performance. Kernel pipelines have already been modeled to predict the execution time aiming at the automatization of the compilation process [94, 162, 21]. For pipelines, one key challenge is the selection of the right execution model, choosing between task and ND-Range, because an incorrect choice may increase the execution time by as much as two orders of magnitude [75, 183].

Existing models have simplified the memory component, especially the GMI, losing details that might provide good opportunities for optimization of kernel implementation. Substantial simplification may be valid for old FPGA devices with simple memory organization but does not apply for current models because kernel resources have grown faster than external memory resources; e.g., an Intel Stratix 10 delivers 9 TFLOPS and the newer Intel Agilex delivers 20 TFLOPS, while DRAM has only improved from DDR4 @ 1333 MHz / 2666 Mbps to DDR4 @ 1600 MHz / 3200 Mbps or DDR5 @ 2100 MHz / 4400 Mbps. In terms of performance, these traditional memory technologies are growing slowly compared with FPGA compute resources, which double every generation [71, 68].

Although external memory technologies are evolving, compared with on-chip memory, the throughput of external DRAM banks is still 380× worse than on-chip, and it is 80× larger in size [71]. Hence, the prediction of FPGA kernel execution time focuses on kernel pipeline and external memory, ignoring the local memory because, in most situations, its impact is negligible.

A novel memory such as HBM, composed of multi-channel DRAM memory, increases the memory bandwidth and concurrency to maintain sufficient parallelism to support kernel requests. FPGA models such as Stratix 10 MX can reach 450 Gbps with an HBM2 composed of 32 pseudo-channels. The main challenge with HBM for FPGA programmers is application design because the Stratix 10 MX was not designed with

a hardware interconnect to enable communication with HBM. That flexibility implies the HLD programmers have to decide how to manage parallel requests in each HBM pseudo-channel [93, 84].

In Intel FPGAs with HLS, as shown in Figure 2.4, the external memory controller has independent units separate from the kernel logic, where the LSUs have the same behavior on all DRAM models, this making it possible to analyze different memories with the same model.

4.5 Memory Model for FPGA accelerators

Based on the fact that the GMI and its interaction with the DRAM have a large impact in kernel execution time, therefore, the modeling of both can provide an accurate analytical model. As the timing of external DRAM memories has been described in the past [21], the proposed model integrates that DRAM analysis to guarantee the accuracy of the estimations.

For programs limited by memory, especially bandwidth, the execution time can be estimated accurately by modeling two key components: the GMI, which is the interface between the kernel pipeline and the DRAM memory; and the DRAM memory timing models themselves. The latter has already been modeled by Cho [19], while the modeling of the former, GMI, can be broken down into models of the different LSUs.

Fortunately, the information available after the translation phase, including datasheet and user input, provides enough detail to estimate both GMI and DRAM delays, without the long delays of the full compilation process. During the translation from OpenCL to Verilog, each global access from the kernel source code generates one or several LSUs in the GMI. For each global access, the HLS compiler determines the proper type of LSU according to a static analysis, as described in Section 4.3.

Table 4.2 summarizes the model input parameters with their corresponding sources, which are described below:

1. Report: html file which shows the kernel's basic blocks and the LSU types for each global access. It is generated in an intermediate compilation stages using *aocl -rtl*.

Table 4.2: Description of model parameters. The *param* label for the Verilog source refers to a variable name in a Verilog instance.

Source	Variable	Definition
Report	$\#lsu$	Number of load-store units
		units per bank
	ls_width^i	Memory width of i LSU [bytes]
	f_{max}	Estimated kernel frequency [Hz]
Verilog	$burst\_cnt^i$	Size of Avalon $burst\_count$ port
		param:BURSTCOUNT_WIDTH
	max_th^i	Maximum threads in a burst
		param:MAX_THREADS
User	δ	Address stride of memory access
	ls_acc^i	Number of access of i LSU
	ls_bytes^i	Bytes of a single ls_acc
	v	Kernel vectorization
Datasheet	$\#banks$	Number of banks in parallel
	dq	Memory data width [bytes]
	bl	Memory burst length
	f_dram	Memory frequency [Hz]
	T_{RCD}	Row activation time [s]
	T_{RP}	Precharge row miss time [s]
	T_{WR}	Time to recovery from Write [s]

2. Verilog: These files contain the description of the LSU IPs, including key thresholds such as max_th^i. They are generated with *aocl -rtl* command.

3. User: Users need to provide the iteration limit for dynamic loops to estimate the number of memory accesses, since it is not usually available at compile time. Note that a compiler pass could automatically infer the user information.

4. Datasheets: The DRAM datasheets provide the timing and the organization of DRAM memory chips.

To begin with, let T_{est} be the estimated execution time of memory intensive applications. With multiple DRAM banks accessed in parallel, the slowest bank time access T_{bank_n} determines the total execution time, such that:

$$T_{est} = \max_{n=1,...,\#banks} T_{bank_n} \tag{4.1}$$

where T_{bank_n} represents the total delay of the *n-th* DRAM bank estimated as the sum of the minimum time, T^i_{ideal}, plus the overhead time, T^i_{ovh}, from every transaction from every LSU, as shown in (4.2). While T^i_{ideal} only depends on maximum memory data transfer capacity and hence is the same for all LSU types, T^i_{ovh} varies with the type of LSU, as the next subsections describe.

$$T_{bank_n} = \sum_{i=1}^{\#lsu} \delta^i \cdot (T^i_{ideal} + T^i_{ovh}) \tag{4.2}$$

where the δ^i factor represents the stride of an access. Regardless of the stride, LSUs always request to DRAM a whole burst of consecutive data, and upon reception, the LSUs discard part of the data burst, increasing the number of memory transactions; e.g., a stride of two discards half of each data burst and doubles the number of accesses.

Assuming a minimum time for fetching all data for the i LSU, T^i_{ideal}, this time can be estimated as the size in bytes , ls_bytes^i multiplied by the number of accesses, ls_acc^i, divided by the kernel memory bandwidth, bw_mem^i, as shown in (4.3).

$$T^i_{ideal} = \frac{ls\_bytes^i \cdot ls\_acc^i}{bw\_mem^i} \tag{4.3}$$

All these equations are valid for memory-intensive applications that can saturate the available memory bandwidth. At this point, the addition of more compute resources does not provide any benefit because execution time is already dominated by the DRAM bank access delay. When the kernel-pipeline clock frequency, f_{max}, is higher than the required minimum frequency for each LSU, f^i_{min}, then the memory

bandwidth is saturated. In that case, the bw_mem^i reaches the maximum bandwidth. Otherwise, the memory bandwidth is non-saturated bw^i_{nsat} as (4.4) shows.

$$bw\_mem^i = \begin{cases} bw\_dram & f_{max} \geq f^i_{min} \\ bw^i_{nsat} & otherwise \end{cases} \tag{4.4}$$

To satisfy the memory bandwidth saturation condition, the kernel needs a minimum DRAM memory data request size of ls_width^i, for each i LSU, noting that ls_width^i cannot be greater than DRAM burst $dq \cdot bl$. The memory bandwidth saturation for double data rate DRAM is $bw\_dram = dq \cdot 2 \cdot f\_dram$, where f_dram is the DRAM frequency. The ratio of bw_dram to ls_width^i describes the relation between kernel-pipeline requests and external memory capacity, defined as f^i_{min}, in (4.5).

$$f^i_{min} = \frac{bw\_dram}{ls\_width^i} \cdot \delta^i \tag{4.5}$$

The modifier δ^i increases the memory burst requests, and therefore, the kernel-pipeline requirements for memory bandwidth.

Although f_{max} is estimated in the intermediate compilation, it could be inaccurate as the wire delay is not considered [144]. The increase in kernel-pipeline resource usage and algorithm complexity could reduce the reported f_{max} after synthesis.

When f^i_{min} is less than f_{max}, the memory bandwidth is non-saturated, and two cases are possible: first, the number of LSUs, $\#lsu$, per memory bank is equal to one, and second, $\#lsu$ is greater than one, in this case, the kernel fully exploiting the double rate memory frequency, multiplying the f_max by two. Finally, bw^i_{nsat} is a portion of the relation between f_{max} and f^i_{min} defined in (4.6).

$$bw\_nsat^i = \begin{cases} bw\_dram \cdot \frac{f_{max}}{f^i_{min}} & \#lsu = 1 \\ bw\_dram \cdot 2\frac{f_{max}}{f^i_{min}} & \#lsu > 1 \end{cases} \tag{4.6}$$

The relation between f_{max} and f^i_{min} shows the “clock crosser" influence on two different clock frequency domains between kernel-pipeline and memory controller, as shown in gray and white boxes on Figure 4.3. This is evidence that the effective

DRAM memory bandwidth in a FPGA could be modified after the compilation process if the f^i_{min} condition is not satisfied.

While this work is focused on bandwidth saturated programs, the non-saturated memory bandwidth includes compute cycles, and these have already been covered [162, 21]. Given bw_mem, the model can predict whether this new model should be used to estimate the execution time or previous compute-oriented models would be preferable, as set out in (4.7).

$$\text{Kernel Bound} \Rightarrow \begin{cases} \text{Memory saturated} & bw\_mem^i = bw\_dram \\ \text{Memory non-saturated} & \text{otherwise} \end{cases} \tag{4.7}$$

Finally, once a kernel is defined as memory saturated, T_{est} can be calculated with (4.1).

4.5.1 Burst-Coalesced LSU

The burst-coalesced LSU is one of the main types of GMI, as listed in Table 4.1 in Section 4.3. In this LSU type, in order to saturate memory bandwidth, the Avalon FIFO needs to be filled with requests. When the kernel pipeline does not make enough requests to fill the memory burst before time out, the memory bandwidth is non-saturated.

It is possible to achieve T_{ideal} for contiguous memory accesses, this type of access hiding PRE/ACT latencies, as was shown in Figure 4.3. Furthermore, bank-interleaving memory controllers can completely hide the opening of new memory banks [60] while the $\#lsu$ remains below two. When the $\#lsu$ increases, this forces the DRAM to open a new row, adding T_{ovh}.

The T^i_{ovh} is proportional to DRAM latency of opening a new page, given by row miss commands (T_{row}). These can be calculated based on the number of times that an i LSU has to open a new row, which depends of the number of burst transactions, with a given $burst\_size$, required to request the total number of bytes ($ls\_acc \cdot ls\_bytes$), formulated as in Equation (4.8). It should be noted that LSU latency and the amount

of data in the Avalon FIFO would hide the kernel latency, and for this reason, only the DRAM latency is considered.

$$T_{ovh}^{i} = \begin{cases} 0 & \#lsu \leq 2 \\ \frac{ls\_acc^{i} \cdot ls\_bytes^{i}}{burst\_size^{i}} \cdot T_{row} & \text{otherwise} \end{cases} \tag{4.8}$$

The estimation of $burst\_size$ and T_{row} for each LSU modifier are analyzed in Subsections 4.5.1 to 4.5.1.

Burst-Coalesced Aligned LSU

This modifier is generated when all the kernel requests are memory addresses aligned to page size, buffering contiguous memory requests until the largest possible burst, or DRAM page size, can be made [67]. Where multiple load/store requests are consecutive words to memory, the burst-coalesced aligned LSU burst-coalesced aligned LSU maximizes the memory throughput. The complete architecture of this LSU for a load and store request is shown in Figure 4.3

Here, to estimate T_{row}, the DRAM $burst\_size$ is defined as the size of burst transaction, which can overlap DRAM commands. DRAM sets the minimum burst transaction size to $dq \cdot bl$, but it can transfer multiple consecutive burst for the same open row yielding (4.9), where $burst\_cnt^{i}$ represents the bus size of the transaction counter, as shown in Figure 4.3.

$$burst\_size^{i} = 2^{burst\_cnt^{i}} \cdot dq \cdot bl \tag{4.9}$$

The estimation of T_{row} is not trivial because the controller can overlap commands due to reordering strategies and the page policy [161]. This model takes into account the inter-command delay for row buffer misses [21] using ACT/PRE latencies, as (4.10) shows. The command sequence PRE and ACT, for read and write, is considered with the same minimum timing as the FPGA profile shows a minimal bandwidth difference between operations.

$$T_{row} = T_{RCD} + T_{RP} \tag{4.10}$$

In a kernel, each global access variable reduces the memory bandwidth with increases in T_{ovh}. The overhead is only zero if the accesses to DRAM banks are consecutive. But if global accesses are to different addresses, as in the case of a multiple global access pointer, the accesses are not consecutive and therefore T_{ovh} appears. Based on this model, we can make a first observation:

Observation 1

Each variable in the same DRAM bank adds an overhead of T_{row}. This time is null using one bank per global access; for example, with multiple DDR4 banks, manually distributing the data buffers and disabling the interleaving with the compilation flag *-no-interleaving*; and with HBM using one variable per pseudo-channel.

Burst-Coalesced Non-Aligned LSU

Both aligned and non-aligned LSUs try to coalesce requests from multiple threads in a single burst command; however, the δ stride of non-aligned access adds a new trigger for a memory request, the number of threads, max_th, that have been launched and coalesced in one memory request.

Equation (4.11) calculates this constraint, called max_reqs, representing the maximum size of a DRAM request. When a coalescer assembles a request, either the request occurs when the amount of data requested is equal to a DRAM page or when the number of coalesced requests have reached max_th, defined as a constant in the LSU Verilog source code. This limit is affected by δ, it reducing the effective burst request. In the other case, the δ fraction of ls_width is the effective burst size, as (4.12) shows. Note that ls_width should be bounded by DRAM page size.

$$max\_reqs^i = \frac{max\_th \cdot ls\_width^i}{\delta + 1} \tag{4.11}$$

$$burst\_size^i = \begin{cases} \frac{max\_reqs^i}{\delta} & max\_reqs^i \leq 2^{burst\_cnt^i} \cdot dq \cdot bl \\ \frac{ls\_width^i}{\delta} & \text{otherwise} \end{cases} \tag{4.12}$$

Based on this model, we can make a second observation:

Observation 2

The stride value δ multiplies the number of memory accesses required, it being able to saturate DRAM bandwidth with discarded data.

Burst-Coalesced Write-Acknowledge LSU

When the global access includes data dependencies in its indexation, the compiler generates a write acknowledgement signal to guarantee the correct ordering of accesses [65]. Therefore, the burst size equals the aligned case from Equation (4.9), and most important, each burst only consumes ls_bytes increasing the total time by $\frac{dq \cdot bl}{ls\_bytes}$. The write-ack signal adds a write command to the DRAM access, increasing the T_{row} delay as (4.13) shows. Based on the write-acknowledge LSU model, we can make a third observation:

$$T_{row} = T_{RCD} + T_{RP} + T_{WR} \tag{4.13}$$

Observation 3

Although the compiler detects access dependencies, the logic tries to generate a burst request to DRAM, because each FPGA cycle at a minimum frequency of $\frac{2 \times F_{mem}}{bq}$ is equivalent to 1 burst request to DRAM.

4.5.2 Atomic-pipelined LSU

The atomic-pipelined LSU executes a read and a write DRAM command. It only supports integer data types without bursting (therefore, in (4.2), $\delta = 1$). For example, `atomic_add` from Listing 4.2 atomically sums `val` to `p`, which is atomically read and written. When `val` is constant within a loop or for multiple work items, then the compiler performs v operations atomically.

Atomic operations cannot be used with multiple memory interfaces as is the case of HBM2 because BSP does not provide support for them.

```
1 int atomic_add(volatile __global int *p, int val);
```

Listing 4.2: Atomic-pipelined add prototype function

Equation (4.14) shows the resulting T^i_{row}, including the two accesses, and T^i_{ovh}, depending on the vectorization factor v. Note that memory saturation in atomic should include the LSU as a unique operation (sum of ls_width) of two LSUs. Based on atomic, we can make a fourth observation:

$$T^i_{row} = 2 \cdot (T_{RCD} + T_{RP}) + T_{WR}$$
$$T^i_{ovh} = \begin{cases} \frac{T^i_{row}}{v} & \text{val is constant} \\ T^i_{row} & \text{otherwise.} \end{cases} \tag{4.14}$$

Observation 4

The atomic LSU is the most time expensive LSU because one FPGA cycle performs only one atomic operation. Atomics are limited to *int* data types.

4.6 Methodology

The experiments have been run on two FPGAs with different memory technologies: an Intel Stratix 10 GX Development Kit with 2 GB of DDR4 DRAM HiLo running at 1866 MHz [102] and an Intel Stratix 10 MX Development kit with a HBM2 memory with 32 pseudo-channels, each one with 256 MB of capacity running at 800 MHz [93, 107]. For details of FPGA models, please reefer to Table 2.2. The Table 4.3 shows the parameters required for the model on each FPGA. The other parameters come from the intermediate compilation of the Intel FPGA SDK for OpenCL 18.1 for Stratix10 GX and 19.3 for Stratix 10 MX. The OpenCL versions are different because the manufacturers designed the BSPs with different Quartus IP versions.

To validate the model, two types of benchmarks are analyzed: first, a set of microbenchmarks, targeting each LSU type from Table 4.1 inside Listing 4.3, where user parameters such as v and the number of global access ($\#ga$) vary. For DDR4 memory on Stratix 10 GX, only one bank is available, and in the Stratix 10 MX with HBM memory, the “heterogeneous memory" feature is used, this assigning each global access to an HBM bank. For burst-coalesced aligned and non-aligned LSUs, δ variations are validated scaling the array accesses by δ. In the non-aligned case, an offset argument is added to the scaled index forcing the compiler to this LSU.

Table 4.3: Fixed variable value to evaluate the LSU model on Stratix 10 GX and Stratix 10 MX with a DDR4 1866 and HBM2 memory respectively. All variables are defined in Table 4.2.

Memory	Variable	Value	Variable	Value	Variable	Value
DDR4-1866 [102]	f_dram	933.3 MHz	dq	8 B	bl	8
	T_{RCD}	13.5 ns	T_{RP}	13.5 ns	T_{WR}	15.0 ns
HBM2 [93, 107]	f_dram	800.0 MHz	dq	8 B	bl	4
	T_{RCD}	14.0 ns	T_{RP}	14.0 ns	T_{WR}	15.0 ns

```
#ifdef HBM // for HBM with multiple banks
  #define g_bank(global_mem_label) \
  __attribute((buffer_location(global_mem_label)))
#else // for DDR4 memory
  #define g_bank(global_mem_label)
#endif
__attribute((num_simd_work_items(SIMD)))
__kernel void test_coalesced(
  __global g_bank(HBM1) const int *restrict x0,
  ..
  __global g_bank(HBMn) const int *restrict xn,
  __global g_bank(HBM0) const int *restrict z)
{
    int id = get_global_id(0);
#ifdef Burst_Coalesced_Aligned
    z[id] = x1[id] + ... + xn[id];
#elif Burst_Coalesced_Non-Aligned
    z[3*id+1] = x1[3*id+1] + ... + xn[3*id+1];
#elif Burst_Coalesced_Write-Acknowledge
    int idr = rand[i]; //work item index
    z[idr] = x1[idr] + ... + xn[idr];
#elif Atomic
    atomic_add(&z[0], x[id]);
    ...
    atomic_add(&z[n], xn[id]);
#endif
}
```

Listing 4.3: OpenCL template microbenchmark to vary global access number.

A second validation is performed with 18 different HPC benchmarks, all memory bound, covering mainly three scientific fields: 1) Physics, 2) Dense linear algebra, and 3)Computer vision. The benchmark are selected from the following sources:

Intel FPGA SDK, Xilinx SDAccel, NVIDIA OpenCL, Rodinia FPGA [183], Chai [46], and FBLAS [100], in which input channels were modified to fit the DRAM inputs.

The execution time is measured with *aocl -report* enabled with profiler compilation, this setting up the hardware counter in the LSU. The atomic cases are measured with OpenCL events since this type of LSU does not have dynamic counters implemented.

4.7 Results

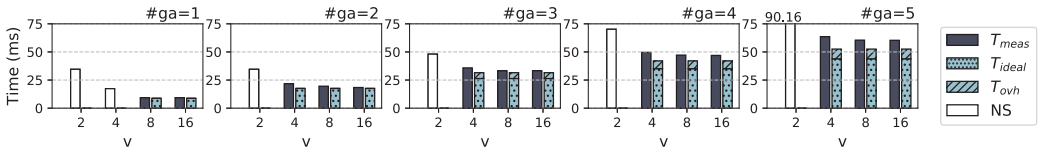
The model validation comprises two sets of experiments. The first, microbenchmarks, includes small programs with multiple configurations of kernel v, δ, and $\#lsu$, enabling us to understand how each parameters affects performance in isolation. The second set is made of complete benchmarks to test the model with well-known applications. A third set of experiments are conducted to compare our proposals with previous ones [162, 21].

The model assumes that in memory saturated applications, the execution time depends more on memory delay than on kernel frequency; this is valid provided that the kernel frequency is high enough for the memory controller to fully exploit bandwidth, namely, f_{min}. The programmers are able to estimate how well the memory bandwidth is exploited based on ls_width and f_{max} from reports since the results show the dependency on these parameters .

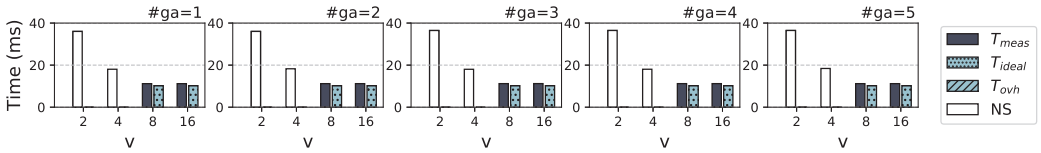
4.7.1 Microbenchmarks

For the sake of completeness, each LSU modifier is evaluated separately. The evaluation comprises the microbenchmark from Listing 4.3 with their body tuned to the LSU type and modifier. Every loop body is based on vector addition to easily change $\#ga$.

Note that in HBM2 memory each global access has a single pseudo-channel to parallelize bank access, while in DDR4, multiple global accesses must be arbitrated by a controller.



(a) DDR4 1866



(b) HBM2, one global access per pseudo-channel

Figure 4.4: Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced aligned LSU varying the vectorization factor (v) and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. The bars with dots and stripes represent T_{ideal} and T_{ovh}, respectively. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.

Burst-Coalesced Aligned LSU

Investigating each LSU type in more detail, Figure 4.4 compares the measured, T_{meas}, and analytically estimated, T_{est}, execution times for a burst-coalesced aligned LSU. For T_{est}, each bar corresponds to the sum of T_{ideal} (dotted) and T_{ovh} (striped). For HBM2, the slowest bank from Equation (4.1) is shown, while DDR4-1866 has only one bank. With this LSU type, each global access generates one LSU ($\#ga$ is equal to $\#lsu$).

For all cases, errors remain below 15%, the simplification of the DRAM commands in the model and the refresh time being among the main sources of error, which can reduce memory efficiency, e.g., the DDR4 IP controller reduces efficiency by around 3.5% [60]. The experiment also evidences that the higher the $\#lsu$, the higher the T_{ovh}; e.g., DDR4 bandwidth reduces by 26%, from 14.2 to 10.5 GB/s with five LSUs. Hence, in this case, *Struct of Array* is a good option for reducing $\#lsu$. In the case of HBM2 memory, the time remains the same with the increase in $\#lsu$ because they run in parallel with one LSU per pseudo-channel.

Figure 4.5 shows the times, normalized to T_{meas} with $\delta = 1$, for multiple stride values. Execution time shows a linear dependency on δ because of the data discarded in each DRAM burst.

Notice that burst-coalesced aligned LSU cannot be generated with all δ values because the compiler does not detect DRAM page alignment. With HBM2 memory,

(a) DDR4 1866 **(b)** HBM2

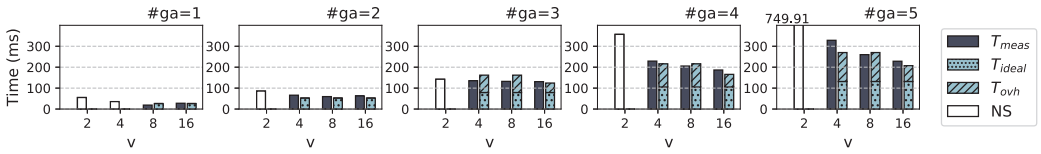
Figure 4.5: Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{meas} for $\delta = 1$. The experiment varies δ with $\#lsu = 3$ and $v = 16$ for burst-coalesced aligned LSUs in two types of external memory: a) DDR4 1866 and b) HBM2, adjusting for special cases.

strided write operations need a correction factor of 4 because they do not detect coalescing, and the burst splitter divides the request into $bl = 4$ words inside a burst taking bl cycles to transfer it, while a read request only needs one clock cycle for bl words. Comparing HBM2 stride request with DDR4, the performance of HBM2 is lower, by 2 $\times$ in the worst case, starting from $\delta = 2$ due to bursts splitting in store for HBM, in spite of parallels between the three LSUs used in this test.

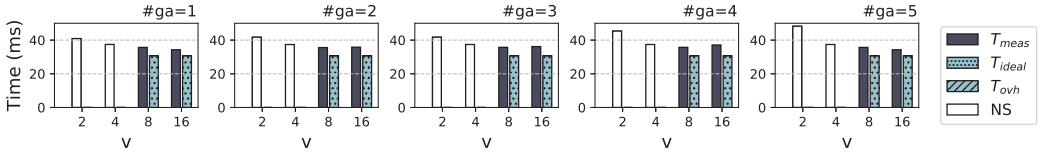
Burst-Coalesced Non-Aligned LSU

The burst-coalesced non-aligned LSU is depicted in line 17 of Listing 4.3 for a $\delta = 3$. Similar to the aligned modifier, in this case, the global access is also supported by just one LSU. burst-coalesced non-aligned LSU, in Figure 4.6, shows a 22% larger error than burst-coalesced aligned LSU, this being attributable to the latency of the coalescer having a large variance; e.g., the number of required address comparisons depends on the coalescer state. The largest errors, as with burst-coalesced aligned LSU, are related to small vectorization factors; in the case of DDR4 with v=4, the calculated f_{min}=349 MHz compared with f_{max} after compilation which is in the range of 301 to 418 MHz placing the kernel near to a non-saturated memory state and increasing the minimum error by 13%. Also note that neither v nor $\#ga$ correlates with the error.

Further, for v and $\#ga$ larger than 4 and 3, respectively, the number of threads in a burst, max_th of Equation (4.12), significantly impacts execution time, which increases linearly and not exponentially like v. This "max_th effect" can also be seen varying δ as Figure 4.7 shows for $v = 16$ and $\#lsu = 3$, with times normalized to $\delta = 1$. For $\delta = 7$, the max_th restriction appears optimizing the access that increases with strides. Compared to an aligned LSU, the performance is 60% lower on average

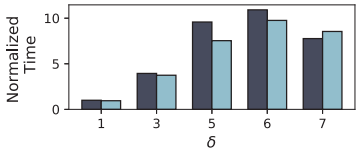


(a) DDR4 1866

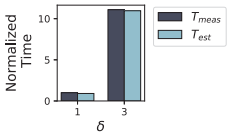


(b) HBM2, one global access per pseudo-channel

Figure 4.6: Measured (T_{Meas}) and Estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced non-aligned LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a)DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.



(a) DDR4 1866



(b) HBM2

Figure 4.7: Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{Meas} in $\delta = 1$. The experiment varies δ with fixed values of $\#lsu = 3$ and $v = 16$ for burst-coalesced non-aligned LSU in two types of external memory: a) DDR4 1866 and b) HBM2.

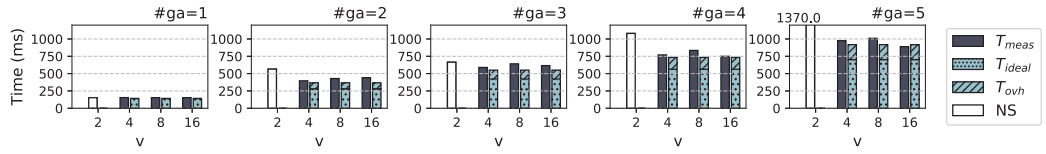
due to address comparison increases and the burst window being reduced to avoid long kernel stalls.

Unlike in DDR4, in HBM2, the execution time does not have T_{ovh}, as in the burst-coalesced aligned LSU case, due to the use of just one LSU per pseudo-channel. It should be noted that $\#ga$ does not vary the estimation results, showing independence between HBM channels.

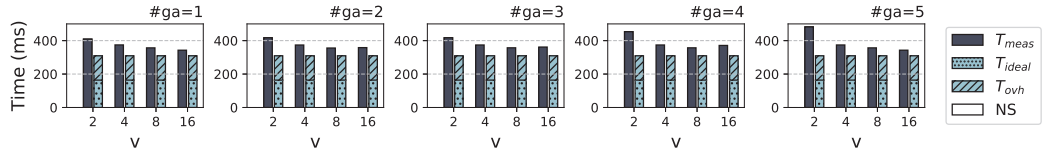
Burst-Coalesced Write-Acknowledge LSU

The evaluation of this LSU type uses the microbenchmark in Listing 4.3, with the code snippet from lines 20 to 21 of.

An array of constant values is generated by software with random values between 0 and 2048, reducing the probabilities of coalescing (2048 over 64 and 32 floats coalesced in DDR4 and HBM2 respectively).



(a) DDR4 1866



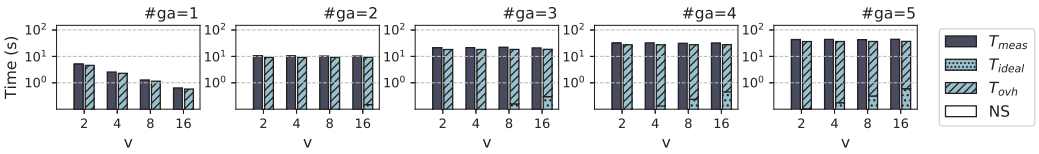
(b) HBM2, one global access per pseudo-channel

Figure 4.8: Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for burst-coalesced write-acknowledge LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated.

In the previously analyzed LSU types (burst-coalesced aligned LSU and burst-coalesced non-aligned LSU), v affected the lsu_width; by contrast, with write-acknowledge LSU, the lsu_width remains constant. To increase the vectorization, the compiler generates as many LSUs as the desired v for each global access. The assumption is that every thread is accessing a different memory location, controlling the memory consistency with the ACK signal. Figure 4.8 shows the comparison between the measured and estimated execution times.

Among all burst-coalesced LSU modifiers, write-acknowledge LSU is the one that penalizes performance the most, growing $24\ \times$ more than with burst-coalesced aligned LSU. The read operations show a stall on read until 98% with two LSUs. To optimize these cases, the programmer should evaluate a balance between the data dependency with writes vs. the use of on-chip memory with a tiling strategy.

For HBM2, the assumption of this LSU type is replaced by a burst-coalesced aligned LSU tree, but as write-acknowledge LSU uses a signal to control pipeline flow. Here, the estimation has a maximum error of 12% for all $v = 2$ values, as with burst-coalesced non-aligned LSU, kernel-pipeline is near to memory saturation with a $f_{min} =$ 400 MHz and a minimum kernel frequency after compilation of 367 MHz, compared with expected value of 450 MHz.



(a) DDR4 1866

Figure 4.9: Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for Atomic-pipelined LSU varying the vectorization factor (v) and global access ($\#ga$) in a DDR4 1866 memory. Non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. The time axis is in seconds and logarithmic.

Atomic-pipelined LSU

The evaluation of this LSU type uses the microbenchmark in line 23 of Listing 4.3. In this code, to generate a single global access ($\#ga = 1$), the global access $xn[id]$ is replaced by a local variable id. Otherwise, each atomic operation generates one global access per v to avoid coalescing. Only DRAM results are shown, because the atomic LSU is not supported with HBM2 memory.

In general, the atomic-pipelined LSU does not change the lsu_width, unlike the burst-coalesced LSU, making T_{ovh} the most significant component in the case of this LSU. Figure 4.9 shows that execution time increases linearly with $\#ga$, the maximum error of 16% corresponds to unaccounted 5 ns per atomic operation. The hypothesis is that this delay is close to the time between the beginning of the internal write transaction and that of the following read command in the same group and same bank (T_{WTR}).

Overall, analyzing read stalls quantifies the impact of the LSU on kernel performance. For burst-coalesced aligned and non-aligned LSUs, the read stall percentages are under 20% because the coalescer partially hides the δ-induced delay. Meanwhile, write-acknowledge LSU has a stall percentage of over 50% as the extra signalling serializes the requests. The atomic-pipelined modifier cannot be measured because profiling is unsupported, but it is safe to assume that stalls will be high due to atomicity requirements.

4.7.2 Applications

To cover a large set of possible scenarios, this section evaluates the model with 18 bandwidth bound applications, mixing single task and NDRange kernels with and

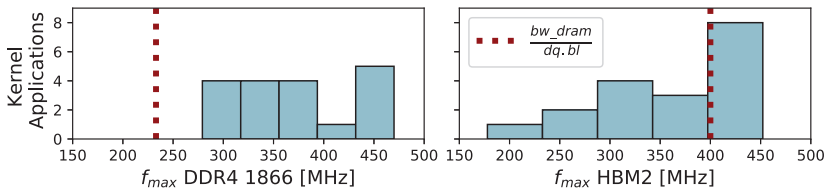


Figure 4.10: Frequencies Histogram for 18 kernel applications; the red dotted line shows the required minimum frequency for maximizing bw_dram.

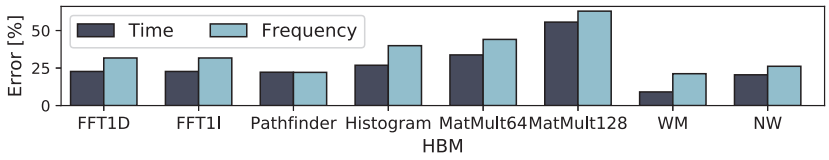


Figure 4.11: Estimation error of the execution time and frequency error from pre-synthesis report and after place-and-route in kernels that are limited by a frequency under $400MHz$ after synthesis in HBM cases.

without channels. Table 4.4 reports the measured and estimated times with the corresponding errors for all of them.

For all the applications with a DDR4-1866, the relative error remains below 9.2% with an average value of 7.6%. With HBM2 memory, the error is higher, with a maximum of 55%.

The main source of error in HBM is the frequency requirements from the controller, which needs $f_{min} = 400$ MHz to maximize bandwidth. Such an f_{min} is difficult to achieve with high resource usage that increases the pressure on the place-and-route compilation phase and reduces the achievable target frequency f_{max} [138]. For example, in MatrixMult with $v = 128$, the kernel requires the highest (53%) DSP resource allocation among benchmarks, the compilation time is around 9 h, and the kernel only achieves a $f_{max} = 177$ MHz, 55% lower than the expected f_{min}. Further, MatrixMult with $v = 64$ uses 26% of DSP blocks, takes 5 h to compile, and yields an $f_{max} = 268$ MHz, 33% lower than f_{min}. If future HLS tools improved place-and-route capabilities, errors would certainly decrease.

To illustrate the frequency differences in DDR4 and HBM2 between applications, the histogram in Figure 4.10 shows the applications distribution in terms of frequency and marks the minimum frequency required to maximize memory bandwidth. On HBM2, 8 applications are critically bounded by the frequency after place-and-route because they do not reach the pre-synthesis reported f_{max}. Figure 4.11 analyzes

Table 4.4: Kernel applications and estimated time in two memories: DDR4 1866 and HBM2.GMI- global memory interconnect BCA- burst-coalesced aligned LSU. BCNA- burst-coalesced non-aligned LSU. ACK- burst-coalesced write-acknowledge LSU. M- Measured. E- Estimated.

			DDR4 1866				HBM2			
Kernel	GMI	$\#lsu$	BW [GB/s]	M.Time [ms]	E.Time [ms]	Error [%].	BW [GB/s]	M.Time [ms]	E.Time [ms]	Error [%]
axpy[100]	BCA	3	11.8	31.9	31.5	1.2	34.5	11.2	10.2	8.6
Dot[124]	BCA	3	13.3	29.4	31.5	7.3	23.4	11.2	10.2	8.5
FFT-1D Direct [65]	BCA	2	13.8	9.5	8.8	7.3	19.8	6.6	5.1	22.4
FFT-1D Inverse[65]	BCA	2	13.8	9.5	8.8	7.4	19.6	6.6	5.1	23.4
iamax[100]	BCA	2	14.3	9.2	8.8	4.6	11.7	11.2	10.2	8.6
nn[183]	BCA	2	13.9	11.0	10.3	6.5	17.6	8.7	8.0	8.8
PrefixSum [75]	BCA	2	12.7	10.0	9.0	10.1	23.1	5.2	5.8	9.6
ROT[100]	BCA	4	11.6	35.7	39.5	10.6	47.9	11.5	10.5	8.7
Sobel Filter HD [38]	BCA	3	13.2	1.9	2.0	6.2	14.1	1.7	CB	-
VectorAdd [65]	BCA	3	12.1	33.3	33.2	5.1	35.9	11.2	10.2	8.6
VectorAdd$\delta = 2$	BCA	3	5.9	67.9	63.0	6.5	4.7	82.6	81.9	0.8
Histogram[46]	BCA	2	14.3	8.9	8.4	5.8	9.2	13.4	9.8	26.6
Hotspot[183]	BCNA	3	7.5	9.7	8.8	8.7	16.9	12.8	CB	-
MatrixMult (v=64) [65]	BCNA	3	8.9	31.2	27.9	10.3	17.6	15.7	10.4	33.3
MatrixMult (v=128) [65]	BCNA	3	9.1	121.2	107.8	11.0	11.6	94.4	41.9	55.6
Pathfinder[183]	BCNA	3	7.6	27.6	25.4	7.9	11.6	13.8	16.5	20.2
WM [171]	BCNA	2	13.9	59.8	55.8	6.6	12.6	0.2	0.2	6.2
NW[183]	BCNA/ACK	4	0.3	1.4	1.4	4.0	0.3	0.2	0.2	25.7

these 8 applications and shows the post-synthesis time and frequency error compared to the estimated pre-synthesis values. There is a strong correlation between time and frequency error, suggesting that compiler accuracy estimating the frequency can limits the model's accuracy. As a especial case, the Stratix 10 MX with HBM2 memory requires higher frequency to saturate memory. In this device, the frequency estimation worsens compared to that of the GX because the MX BSP uses 32 separate global memory interfaces connecting to the physical pseudo-channels with 256-bits buses. In fact, the worst estimation time and the worst frequency estimation from the tool comes from MatrixMult in where the routing tool reports routing congestion warning.

4.7.3 Comparison With Other Models

This subsection compares the proposed model with two state-of-the-art models: Wang and HLScope+ [162, 21], reproducing the mathematical models for the microbenchmarks, with $f = 16$, and for the vectorAdd application. Unfortunately, comparison with other applications is unfeasible because the dynamic profiling tools feeding Wang and HLScope+ are not available. The tests are run with two BSPs for Stratix 10 GX with different DRAM frequencies, 1866 and 2666 MHz.

In all but one case, μb burst-coalesced aligned LSU, the error found in this study is lower than that of Wang and HLScope+ as Table 4.5 shows. Comparing the maximum error of each model, this proposal is up to 400 and 5 $\times$ more accurate than Wang and HLScope+, respectively.

In Wang's case, the errors come from an incomplete support of all LSU modifiers and not fully including the memory features (bandwidth, frequency, row misses, ...), unlike in this study.

On the other hand, the HLScope+ model for Xilinx devices considers memory bound applications where the estimation is primary affected by DRAM bandwidth. HLScope+ requires a board characterization to compute the controller overhead (Tco) [125]; this parameter is different for each benchmark because Tco varies with access type; this study uses Tco =2.5 ns for $\#lsu > 3$, and Tco =0 ns in other cases.

Table 4.5: Execution time estimated error; μb, BCA, BCNA, and ACK refer to microbenchmark, burst-coalesced aligned, burst-coalesced non-aligned, and burst-coalesced write-acknowledge LSUs, respectively.

Memory	Benchmark	$\#lsu$	Wang[%]	HLScope+[%]	This work[%]
DDR4-1866	μb BCA	1	17.3	12.7	5.6
	μb BCA	4	0.3	10.6	4.4
	μb BCNA	3	-	71.1	4.0
	μb ACK	32	8049.9	63.2	27.9
	VectorAdd	3	19.3	21.0	5.1
DDR4-2666	μb BCA	1	69.6	57.8	4.7
	μb BCA	4	37.8	19.6	5.8
	μb BCNA	3	-	137.9	8.7
	μb ACK	32	11 279.4	47.6	8.8
	VectorAdd	3	67.9	63.3	1.0
HBM2	μb BCA	1	145.5	83.7	8.4
	μb BCA	4	151.2	83.7	8.6
	μb BCNA	3	-	118.8	13.9
	μb ACK	32	4910.8	78.1	14.7
	VectorAdd	3	9.8	83.7	8.7

The two state-of-art models compared only support aligned and random access, but as this study shows, the memory strategies go one step further using HLS tools combining and modeling GMI and DRAM behavior.

In addition, note that Wang and HLScope+ do not adapt well to memory changes and only cover DRAM, unlike the proposal in this study that supports both.

4.8 Conclusions

As in other HPC processors, memory in FPGAs is one of the most critical aspects of system performance. This work proposes an analytical model that identifies the main parameters that control the total execution time when the kernel-pipeline saturates memory bandwidth, a common situation for HPC applications. Specifically, the model determines the memory saturation through the relationship with memory occupation and kernel frequency and accurately estimates the kernel execution time without a time-consuming synthesis process, helping programmers and HLS tools to

design and anticipate performance without extensive exploration processes, as used in other studies.

The model stems from a detailed study of the generated RTL code, instantiated IPs, and FPGA architecture without loss in flexibility that is demonstrated with two DRAM technologies: DDR4-1866 and 3D-stacked HBM2.

The results show the model has an average error of 11.4% for DDR4 and 10.4% for HBM2. Errors above average are directly associated with kernel frequency limitations in the compilation process. Compared with two state-of-the-art models, mainly focused on computing, the proposed model at least halves the error and shows adaptability to two technologies and memory frequency variations, unlike other proposals. Our future work aims to integrate this type of model into scheduling policies of heterogeneous systems, where predicting performance before launching a kernel can make a difference, helping to achieve higher performance and energy efficiency.

4.9 Contributions

- We provided a detailed description of the global memory interconnect for HLS in FPGAs.
- We proposed, to the best of our knowledge, the first analytical model that estimates the execution time of HLS-compiled memory intensive applications. This work has been published in *28 th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020*.
- We introduced a novel classification of the FPGA kernel-pipeline state based on memory bandwidth and a set of hints derived from observations of the analytical model to identify bottlenecks and guide programmers for optimizing kernels in two memory technologies: DDR4 and HBM. This work has been published in *IEEE Transactions on Computers, 2021*. The repository with the memory model and collected data is open in the repository IEEE DataPort

5 FPGA Frameworks to Improve Design Productivity

In this chapter, we take program portability as one of the pillars to improve programmability in FPGAs, exploring the implementation of a high level framework for computer vision applications as OpenVX. Previous proposals for FPGAs are vendor restricted to either Xilinx or Intel FPGA, and in this chapter, we ported a Xilinx proposal of OpenVX to Intel FPGA devices. The proposed tool is templated-based, enabling programmers to keep a platform-independent code, hides design decisions such as the type of communication among function nodes in a graph (streams), the concurrency programming model(system of task), and how to integrate the design in an FPGA accelerator coupled to a host(OpenCL libraries). All the new features in the library are evaluated with complete graph applications.

5.1 Introduction

Developing applications for FPGAs has the main drawback the low programmability since the freedom in the design space of FPGAs needs to handle a large amount of details. Although HLS helps in the design processes and verification, the design needs to be FPGA-friendly being this tricky, furthermore, since the programming tools are vendor specific, the code portability is another important problem in FPGAs.

The development of libraries using open programming standards can increase the programmers productivity, even more if the libraries promises portability among vendors. With this aim, in this chapter is explored an open programming standard as OpenVX for computer vision applications, since these applications tend to offer a high level of parallelism with many data-independent operations.

OpenVX presents an open, royalty-free standard for cross-platform acceleration [45] where applications are expressed as graphs to maximize optimization potential

because all dependencies are known before the graph is processed. On FPGAs, the acceleration of OpenVX applications remains a challenge because their efficient implementation requires per-device specific optimizations on primitives and communication. Some High-Level Synthesis libraries address these requirements; e.g., HiFlipVX, an optimized library of OpenVX functions that exploits streaming capabilities and parametrization for Xilinx FPGAs [83].

However, HiFlipVX highly-tuned implementation is neither portable nor efficient on other FPGA platforms such as Intel. Implementing a portable OpenVX API for FPGA requires to maintain a user-facing API as close as possible to the OpenVX standard, in this cases the support is extended for Intel FPGA devices with different external memories as DDR4 and HBM.

5.2 Related Work

Computer vision and image processing algorithms require high performance and energy efficiency that can be achieved with FPGAs [48]. Unluckily, the big effort required for programming FPGA is a huge drawback that makes its adoption difficult.

Image processing on a FPGA can be implemented with Domain Specific Languages (DSL). For example, the newly HeteroHalide [92] extends the Halide DSL, formerly used on CPU and GPU, to support Intel and Xilinx devices. Hipacc [137] developed another DSL to support multiple back-ends from different vendors and devices such as FPGA, GPU, and CPU. Also, a Hipacc extension provides support for the OpenVX API [127], but they do not include results for whole application graphs, as this work does. PoliMage [22] and Pu's [132] are two proposals that support Xilinx FPGAs. Despite DSLs are facilitating FPGAs adoption, the steep learning process and the difficulties to enlarge their functionalities remain a challenge.

A more suitable option to ease FPGA implementation is the adoption of a library approach or standard based library. For example, implementing functions from the OpenCV library, Xilinx provides the xfOpenCV library [169]. Other libraries target specific FPGAs vendors, e.g., HiFlipVX and AFFIX are OpenVX libraries for Xilinx and Intel, respectively [83, 151, 150].

Standard libraries together with the adoption of vision standards such as OpenVX could ensure adequate cross-platform portability and performance. Moreover, applications require intensive tuning for each FPGA vendor, even with HDLs. In the case of computer vision applications, as other ones, each type of FPGA requires specific coding style to achieve optimal performance [48, 159]. Among the available options, HiFlipVX and AFFIX are the ones that could offer a more general computer vision library.

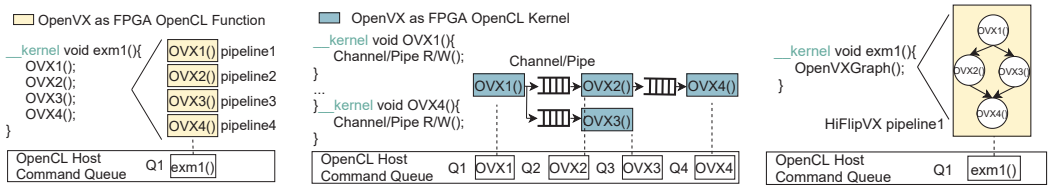
Nevertheless, AFFIX is based on OpenCL, which limits the OpenVX functions and graphs implementation. On the other hand, HiFlipVX, through the use of standard C++ language simplifies the graph's implementation, and it focused on portability including explicit data type management to generate optimized hardware. Moreover, HiFlipVX was validated out in numerous embedded applications for Xilinx [142, 131, 2]. This previous analysis recommends to extend HiFlipVX as an standard OpenVX based library to be compatible also with Intel FPGAs.

5.3 OpenVX Programming Flow Alternatives on FPGA

One of the most successful approaches in heterogeneous systems is OpenCL, because it unifies the programming language across devices such as CPU and GPU. As an High-Level Synthesis language for FPGA, OpenCL still suffers from a limitation: optimization strategies differ from those from other devices and require choosing the appropriate OpenCL execution model [75, 185]. Furthermore, code written with only the OpenCL standard does not perform well on FPGAs as it requires manufacturer defined extensions.

Programming the OpenVX standard using OpenCL for FPGAs is a challenge since OpenVX applications use a graph-based programming model where nodes, instances of kernels, contain the function code; and edges represent the data movements [45]. This data flow programming model has two main design alternatives in OpenCL:

- *Standard OpenCL*: each OpenVX node is an OpenCL kernel, as shown in Figure 5.1a. This alternative is portable between manufacturers; but the main disadvantage is the lack of guarantees to generate a deep pipeline connecting the function nodes, because each kernel requires control and communication



(a) Standard OpenCL **(b)** OpenCL Channels **(c)** HiFlipVX proposed flow

Figure 5.1: Programming flow alternatives for OpenVX using HLS for FPGA devices. The yellow boxes show OpenVX functions implemented as OpenCL functions and the green ones the OpenVX functions implemented as kernels. The bottom boxes show host command queues, Qn, that manage the kernels.

with the host. Outside of the standard, Xilinx defined their own pragmas and streaming interfaces to generate deep pipelines.

- *OpenCL channels*: each node is an OpenCL kernel, and channels/pipes connect them all. This option allows deep pipelines by the use of streaming communication among kernels, as shown in Figure 5.1b. In this case, multiple command queues are required to launch every kernel from host to get a concurrent execution of the graph. This approach is implemented and named differently by each FPGA vendor; e.g., Intel and Xilinx adopt channels and pipes, respectively.

These two approaches evidence the portability problem between manufacturers and the limitations of standard OpenCL API, whereby each FPGA manufacturer extensions help to optimize and guide the compilers through bitstream generation. Even, sometimes, these extensions are different per FPGA device family limiting portability [159].

In terms of performance, the use of the aforementioned channel approach allows higher throughput and lower latency, but due to restrictions of the OpenCL standard, generating portable and easy to use libraries is a challenge. For example, AFFIX implements OpenVX graphs with single-input single-output host pipes [151] curtailing the OpenVX specification, which defines multiple-input multiple-output edges.

Besides OpenCL, a more flexible HLS language is C/ C++. Although C/C++ suffers the portability restrictions between manufacturers, the programming details can be hidden to the programmer under wrapper layers.

For Xilinx devices, HiFlipVX implements OpenVX using C/C++, enabling a highly parameterizable library. However, to complete an efficient and portable OpenVX

Table 5.1: Programming flow alternatives to implement the OpenVX standard.

Programming flow	Manufacturer portable	Deep pipeline	Host dependency
Standard OpenCL	✓	✗	LOW
OpenCL Channels	✗	✓	HIGH
HiFlipVX	✗	✓	-
This Work	✓	✓	LOW

specification, it is necessary to port the library to Intel devices. The differences between C/C++ standards and compiler, such as OpenCL, are not trivial, showing differences between manufacturers. Also, FPGA families present a wide variety of designs, from simple embedded devices to high-performance ones with external memory and ports, specially oriented to HPC applications.

This work overcomes those limitations. Specifically, HiFlipVX achieves both portability, supporting two of the main FPGAs manufacturers, and performance, by coalescing OpenVX nodes in a single OpenCL/RTL element maximizing pipeline deep for Intel FPGAs as shown in Figure 5.1c. With this strategy, OpenVX applications overcome the pipeline depth limitations in Standard OpenCL (Figure 5.1a) and reduces the host dependency on OpenCL Channels implementation (Figure 5.1a). This property is specially crucial for Intel FPGA devices as Table 5.1 shows.

5.4 HiFlipVX

HiFlipVX is an open source HLS FPGA library for image processing applications [83, 82, 81]. HiFlipVX is a C++ based library containing 53 functions, which are highly optimized and parametrizable using templates. Most of its functions, or object kernels, are based on the OpenVX standard. They are implemented to be streaming capable with stream data objects, on edges, to link kernel instances as nodes in a graph. It extends the OpenVX based functions by additional parameters, such as vectorization, or more options, such as additional data types.

The functions in HiFlipVX can be categorized in pixelwise, filter, analysis, and conversion functions as Figure 5.2 shows. Pixelwise functions process the input images pixel by pixel, like adding two images together. Filter functions work in a window on the input image, like in a Gaussian filter. The conversion functions

change the image by scaling it or changing the image format. The analysis functions usually have to perform a complete analysis of the input image, such as creating a histogram.

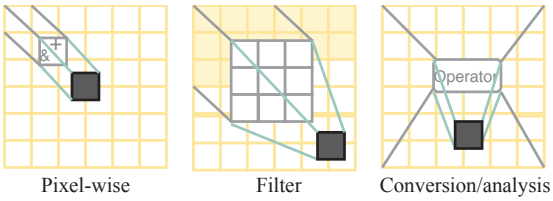


Figure 5.2: Image functions categories implemented in HiFlipVX.

The library outperforms a vendor-specific library, xfOpenCV, for Xilinx FPGAs in terms of resources and execution time [83]. The functions of HiFlipVX were used for various applications [142] [131], and even shows that the use of vectorization increases not only performance but energy efficiency as well [2].

5.5 Methodology

All experiments have been run on two high-end FPGAs: an Intel Stratix 10 GX Development Kit and an Intel Stratix 10 MX. Both boards use the PCIe Gen3 x8 to connect with the host CPU. For more details about FPGAs resources please refer to Table 2.2.

This work evaluates the performance portability of HiFlipVX with parameters such as latency, initiation interval (II), and resource estimation from RTL compilation using i++ HLS compiler V. 19.4. The FPGA core power measurements use the Board Test System application provided by Intel, with a 1 second sampling rate. To ensure power accuracy, kernels run at least 1 minute to obtain measurements. For most experiments, the Stratix 10 GX was selected as the reference board, since the only difference with the MX is the memory technology: DRAM vs. HBM banks.

Our benchmark suite comprises four representative OpenVX graphs, including all the categories of Figure 5.2, from the Intel OpenVX and Khronos samples:

- Canny edge detector: Popular multi-stage algorithm for edge detection and suppressing noise.

- Auto-contrast: Algorithm to improve contrast in images, adjusting the image intensity.
- Census transform: A common algorithm for correspondence problem used in stereo image processing for disparity calculations [174].
- Skin tone detection: Algorithm to detect human white skin tone.

Finally, these benchmarks are also used to compare with existing state-of-the-art approaches running them on the same FPGA, except the skin tone which is not implemented by other works.

5.6 Tuning HiFlipVX for Intel FPGAs

The OpenVX specification provides a high level abstraction to easily implement computer vision applications on multiple devices. The OpenVX objects are designed for dynamic applications, so the runtime provides support to manage objects during execution. However, since bitstream generation takes a long time, on FPGAs, the verification and optimization of OpenVX graphs has to be statically performed at compile time.

The OpenVX standard leaves the optimization process to vendors. In the case of HiFlipVX, the new implementation supports programmer's optimizations through specialized versions of its template-based API for each vendor. So, programmers can tune the OpenVX applications according to the FPGA platform with minimal changes in the user-facing code. Such portability from Xilinx to Intel implementation has required changes in the implementation of three OpenVX components: execution model, kernels, and edges. Kernel nodes are the compute part of the graphs, while edges have the memory management with virtual and image objects which potentially improves speed up.

5.6.1 Execution model

For Intel FPGAs, HiFlipVX synthesizes every graph as a single kernel (Figure 5.1c). The system of task, a proprietary Intel API, enables task-level pipelining, allowing

asynchronous nodes to create a graph for Intel FPGAs. On the contrary, the Xilinx specialization uses the `HLS dataflow` pragma for function or loop level parallelism.

5.6.2 Kernels

HiFlipVX kernel nodes are implemented with C++ functions, so the first step to maintain kernel performance and properly guide the compilation process is to add specific *translations* of Xilinx's pragmas to their Intel counterparts. Specifically, the next two pragmas and component attributes are used:

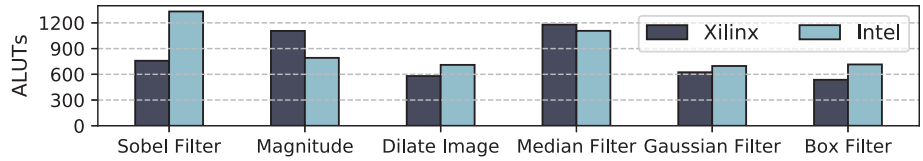
- `HLS array_partition/hls_register`: forces the compiler to generate variables as registers.
- Loop pragmas: the difference is the location in the code. These pragmas are inserted after and before the loop, for Xilinx and Intel, respectively.

The resource utilization comparison of the HiFlipVX for Xilinx [83] and Intel devices shows that the ALUTs resource usage is similar, less than 15% variation for 6 representative OpenVX functions (all running at the same 100MHz frequency), except Sobel Filter, 27% difference, as depicted in Figure 5.3a.

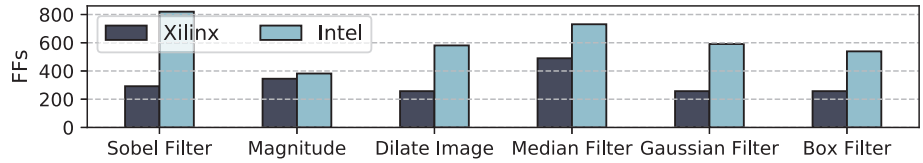
Since the core programmable unit in Intel Stratix architecture packs four-input LUTs and registers (FFs), the FFs usage in Figure 5.3b shows a similar tendency as ALUTs. The RAM usage in Figure 5.3c, shows the same number of blocks although the RAM sizes are different, 18K in Xilinx and 20K in Intel; the similarities are attributed to SIMD vectorization of 1 which synthesizes arrays and variables as registers. These results evidence the differences between architectures and HLS tools; e.g., Xilinx LUTs are capable of self-split to implement two separated logic functions, unlike Intel that has dedicated ALUTs to improve routing time in complex designs [49].

5.6.3 Edges

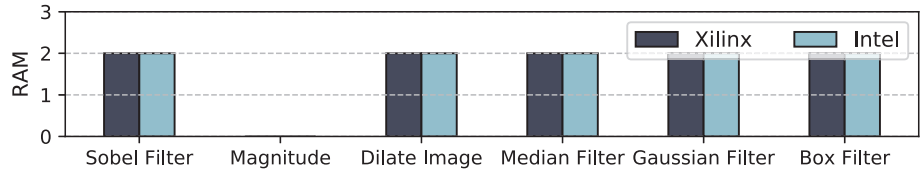
For Xilinx, HiFlipVX implements optimized communications through streaming with *HLS STREAM* pragma. The pragma creates FIFOs or double buffers to transfer data between functions or loops in a data flow area and it uses pass-by-pointers for kernel



(a) ALUTs



(b) FFs



(c) RAM

Figure 5.3: Resource comparison between Intel and Xilinx [83] FPGA at 100MHz and vectorization equal to 1, for 6 sample OpenVX functions.

node parameters. In general, these choices guarantee an Initiation Interval, II, equal to 1 cycle and low latency for filter-type kernels [83].

The lack of equivalent pragmas for streaming communications in Intel API and the pass-by-pointer as parameters can result in kernels with poor performance, constraining the II up to 114 cycles, because the HLS tool generates a single Avalon Memory-Mapped (MM) Master interface with a single arbiter for all variables [38].

When function parameters are passed-by-reference, which are more suitable for Intel [73], the II reduces to 1 cycle, substantially improving the pipeline performance. The first two groups of bars in Figure 5.4 show a 114× cycle difference between the pass-by-pointer and pass-by-reference for a 3x3 filter.

Reference parameters do not support concurrency requirements of nodes in OpenVX graphs. To support them, the Intel system of task with *stream* as function parameters allows nodes to run asynchronously. Streams reach an II of 1 cycle and, in practice, resulting in Avalon streaming interfaces which provides high-bandwidth and low latency communication.

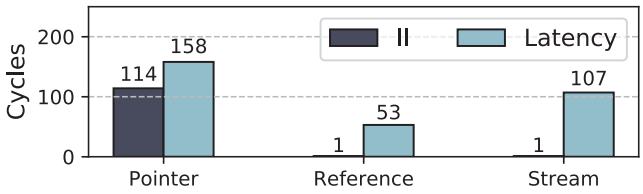


Figure 5.4: Latency and Initiation Interval for interface optimizations on edges in a 3x3 filter function (lower is better).

Comparing the *streams* with *reference*, streams latency is up to 2 × higher because system of task adds control logic in kernel pipeline to communicate among graph nodes. Figure 5.4 shows the impact on both II and latency of all the interface changes: passing arguments by reference and stream communication among kernels.

In terms of code implementation, Intel stream interface has 3 specific data types: *stream_in* for inputs, *stream_out*, for outputs, and *stream* for general interconnect between kernel nodes. To achieve the portability, the Listing 5.1 shows the data type redefinition of the vx_image based on templates which allows to adapt the hardware with the vectorization factor (V) and the capacity of stream buffers (buff_cap). This implementation hides the hardware interface details to programmers.

```
1
2  template<class T, const size_t V,
3      int stream_type, uint buff_cap=256>
4
5  using vx_image=
6      typename conditional<stream_type == vx_streamIn_e,
7      ihc::stream_in<vx_image_t<T, V>>,
8      typename conditional<stream_type==vx_streamOut_e,
9      ihc::stream_out<vx_image_t<T,V>>,
10     typename conditional<stream_type==vx_stream_e,
11     ihc::stream<vx_image_t<T,V>, ihc::buffer<buff_cap>>, vx_image_t
      <T,V> > ::type>::type>::type;
```

Listing 5.1: vx_image for virtual image implementation with Intel streams support

Virtual image objects implemented with *streams* are limited to access by reference, and the use of arrays of *streams* are not allowed. Also, multiple reads from a stream by different nodes require to duplicate the number of edges in the FPGA. For this reason, a custom internal kernel `vxSplit` is needed to concurrently feed multiple kernels with a single copy of the data-stream references.

Contrary to virtual image objects, images references allow direct user access, which creates an opaque reference to an image buffer [45]. In Intel FPGA, the user can access data through external ports using the Avalon MM buses.

In embedded FPGAs, *stream* interfaces with input and output qualifiers are enough to control I/O ports. However, discrete devices with an external memory, as DRAM, require memory IP controllers. To manage external DRAM memories, HiFlipVX takes advantages on existing host drivers for OpenCL/SYCL to perform the required transactions. Also, those transfers are transparently instantiated with two custom kernels.

The DRAM interfaces are created with the *ihc::mm_master* to specify the external Avalon MM data bus interconnection to the OpenCL/SYCL drivers. Listing 5.2 shows the vx_image to create images for FPGAs with DRAM support. The data bus size (WIDTH_MEM) is parametrized with the specification of DRAM memory controller from the BSP (Board Support Package), and PORT enumerates the bus interface. The last template parameter, emb_x, advises the compiler whether the interface is embedded or not, for Xilinx devices it is always true.

```
1 template<class T, const uint WIDTH_MEM, uint V=1, uint PORT=1, uint
      emb_x=1>
2
3 using vxCreateImage =
4     typename conditional < emb_x == 0,
5     ihc::mm_master<vx_image<T, V>,
6     ihc::aspace<PORT>, ihc::awidth<WIDTH_MEM>,
7     ihc::dwidth<32>, ihc::latency<0>,::maxburst<16>,
8     ihc::align<64>, ihc::waitrequest<true>>,
9     vx_image<T, V>>::type;
```

Listing 5.2: vx_image for image implementation with Intel DRAM support

Image objects for DRAM generates load/store units for continuous and aligned memory accesses, user defined parameters as burst size with the coalescence parameter are available for user optimizations. Furthermore, the load/store controller allows to adjust technology differences between FPGA boards and maximizes DRAM bandwidth. Figure 5.5 shows the interfaces and load/store units required to interconnect a DRAM memory to HiflipVX graph.

To evaluate Load/Store units, Figure 5.6 plots the performance of Canny edge detector as a representative graph. It shows that high coalescence factors with very

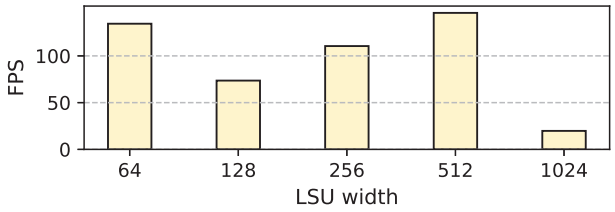


Figure 5.5: DRAM memory interconnection to a HiFlipVX graph.

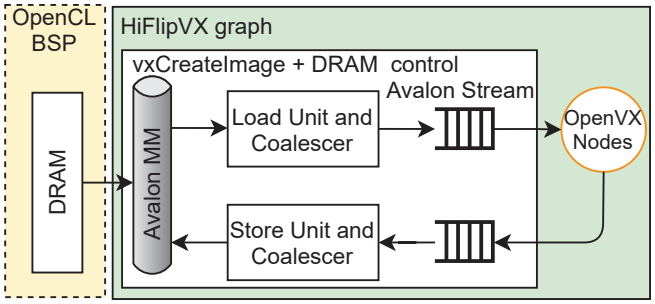


Figure 5.6: Efficiency (Frames Per Second) for canny edge detector with a HD image varying coalescing to read DRAM memory (LSU width), higher is better.

wide LSUs, $>$ 512 bits, can reduce performance up to 7 $\times$, because the compiler heuristic generates a non-aligned controller access. In load/store units with a bus width smaller than 512 bits, the maximum DRAM burst is underused, except in 64 bits which is the same bus width as DRAM (dq).

Once an OpenVX graph has been programmed in C/C++ with HiFlipVX, the compilation flow depends on the target FPGA. For an embedded FPGA, the bitstream can be generated after the RTL generation, and for an Intel discrete FPGA, it must be coupled to a BSP, which is part of the OpenCL and SyCL drivers, to enable communication with a host CPU.

OpenCL and SyCL Library feature allows including RTL modules into function kernels packaged into an library object (.lib). However, the system of tasks used in HiFlipVX is not supported yet. To overcome this problem, the compilation flow has an additional step, supported with a tool extension in HiFlipVX that takes two inputs: 1) an XML file with the BSP memory port descriptions, and the RTL from HiFlipVX, both of them compatible with the target FPGA.

The tool extension output enables the library generation with Intel standard *aoc* tools. As result, the HiFlipVX libraries objects are ready to be used in OpenCL/SyCL kernels. Since OpenCL backend implementation is more mature than SyCL, it has

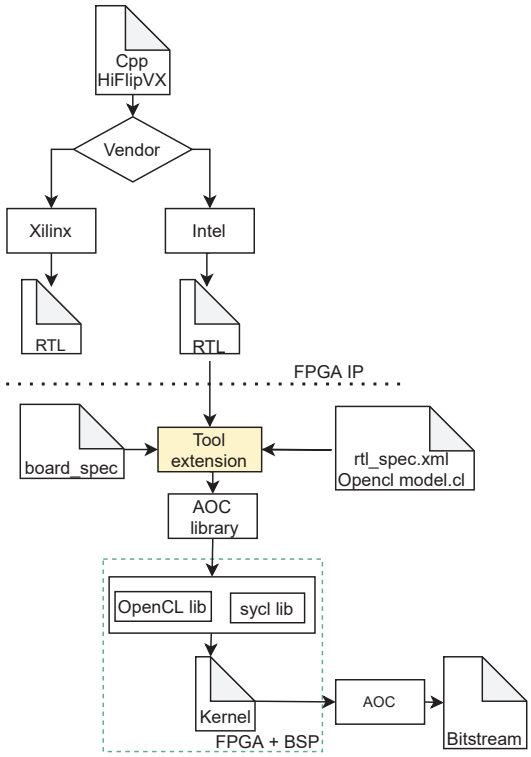


Figure 5.7: HiFlipVX programming and compilation flow for Xilinx and Intel FPGAs.

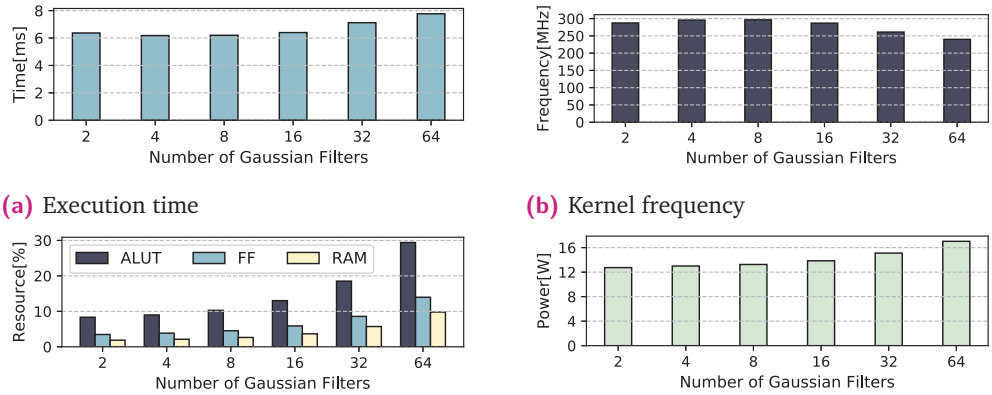
been chosen to be evaluated in this work. Figure 5.7 shows the compilation flow to couple the Intel HiFlipVX graph to a heterogeneous system (right path) and how the Xilinx flow is unaffected (left path).

5.7 Results

This section starts analyzing how the new HiFlipVX implementation behaves when the graph complexity changes. Then, it evaluates HiFlipVX running 4 representative OpenVX graphs and, finally, compares this work with two state-of-the-art proposal.

5.7.1 HiFlipVX Scalability Analysis

To assert how system of tasks and deep pipelines impact on graph scalability, the first stage of the SIFT feature detector is used as a synthetic graph benchmark. This multi-Gaussian graph applies multiple times a Gaussian filter to an image stream [97]. The



(a) Execution time

(b) Kernel frequency

(c) Resource usage

(d) Kernel Power

Figure 5.8: Impact of node scalability on a) execution time; b) frequency; c) resource utilization; and d) power consumption for the Multi-Gaussian synthetic benchmark using the Stratix 10 GX.

benchmarks allow us to tune the depth of the resulting kernel pipeline by adding Gaussian filtering steps, one after the other.

Figure 5.8a plots the impact of the number of filter nodes for the multi-Gaussian graph (kernel pipeline depth) on execution time and FPGA frequency. From 2 to 16 filters, memory latency hides computation which flattens the execution time. After that point, 16, execution time increases almost linearly with the number of filters, showing good scalability. Please note the slight frequency reduction for large number of filters also contributes to the larger execution time.

Resource usage is shown in Figure 5.8c, which increases linearly, with a growing rate of 0.33, 0.17, and 0.13 for ALUTs, FFs, and RAMs resources, respectively. As a consequence, FPGA power raises with a growing rate of 74mW per additional Gaussian filter stage as is shown in Figure 5.8d. In summary, HiFlipVX with the system of task scales well without adding any extra overhead increasing the graph complexity.

5.7.2 OpenVX Application Resource Utilization

This section analyzes resource usage (per-kernel) of four representative applications: Canny edge, Autocontrast, Census transform, and Skin tone detection. Figure 5.9 shows the graph diagram for all of them. For the sake of clarity, the custom internals

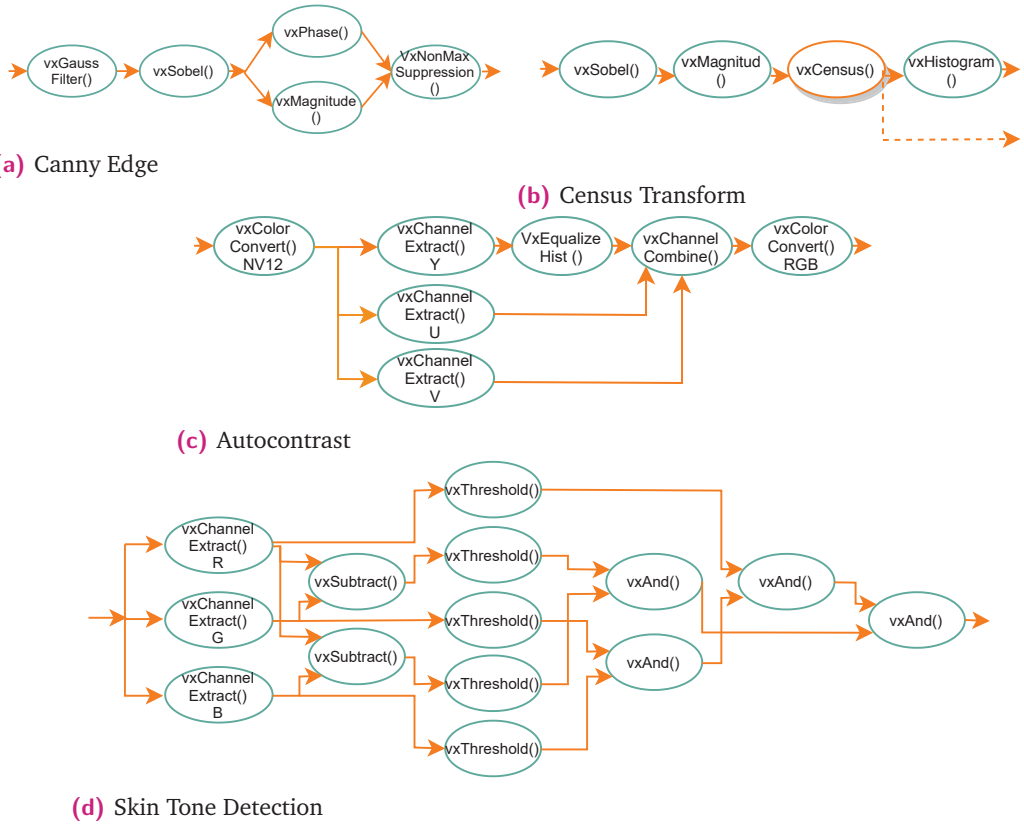


Figure 5.9: OpenVX application graph diagrams. a) Canny edge detector, b) Census transform, c)Autocontrast image, d) Skin tone detection.

kernels, enabling DRAM and splitting data streams (vxSplit) described in Sec. 5.6, are not depicted in the graphs.

Canny Edge

The Canny edge detector, Figure 5.9a, is a multi-node graph algorithm that extracts the edge information from images. In HiFlipVX, its implementation consists of 5 nodes. Table 5.2 shows the estimated resource usage from the Intel HLS compiler report for all Canny edge nodes[1].

The image objects are the most resource demanding function since it track multiple external memory request at a time, trying to group access before being send to the memory controller.

[1]Since all FPGAs used in this work are from the same family, Stratix 10, the resource estimation on HiFlipVX graph are equal, and from here on, all results corresponds to the Stratix 10 GX FPGA, except when noted.

Table 5.2: Estimated resource usage for each OpenVX function in Canny edge graph using HiFlipVX with a 4K image and vectorization factor of 8 on a Stratix 10 GX.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	10553	39508	17	0
vxGauss	3627	5620	18	0
vxSobel	5907	8854	19	0
vxSplit	221	117	2	0
vxMagnitud	5907	8966	4	8
vxPhase[2]	165	133	1	0
vxNonMaxSuppression	4605	5842	18	0
Store Image Object	4177	11949	18	0

Table 5.3: Estimated resource usage for each OpenVX function in Autocontrast graph using HiFlipVX, with a HD image and vectorization factor of 1.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	934	3025	16	0
vxColorConvert(NV12)	1273	1744	0	1
vxSplit	130	103	0	0
vxChannelExtract	143	119	0	0
vxEqualizeHist	2584	3874	1029	0
vxChannelCombine	173	143	0	0
vxColorConvert(RGB)	1037	1268	0	0
Store Image Object	1102	3605	18	0

Autocontrast

Autocontrast, requiring to extend HiFlipVX to support the graph from Figure 5.9c with two new kernels for color conversions: NV12 to RGB and RGB to NV12, and EqualizeHist.

Autocontrast requires more RAM resources than other graphs because the intensity channel (Y) is stored in RAM memory until histogram is calculated. This strategy avoids stalls in streams at expense of higher resource usage that mainly depends on the input image size; e.g., if the image size changes from HD to 4K, RAM usage increases by 4 $\times$. HiFlipVX enables the user to provide FPGA tuning parameters. For example, in this case, the code includes a hint to implement the DRAM access coalescence with a LSU width of 64 bits to save resources and compensate the extra

Table 5.4: Estimated resource usage for each function in Census transform using HiFlipVX, with a 4K Image and vectorization factor of 8.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	10569	39520	17	0
vxCensus	179	208	0	0
vxHistogram	960	16924	2	0
Store Image Object	2095	5501	18	0

Table 5.5: Estimated resource usage for each OpenVX function in Skin tone graph using HiFlipVX, with an HD image and vectorization factor of 1.

Function	ALUTs	FFs	RAMs	DSP
Load Image Object	3675	17257	16	0
vxAndNode	152	119	0	0
vxSubtract	230	153	0	0
vxThresholdNode	154	120	0	0
Store Image Object	2380	6936	18	0

DRAM usage. The Table 5.3 shows the resource for each function in the Autocontrast graph.

Census Transform

Census transform is not part of the OpenVX standard, so we added it to the HiFlipVX library. The implementation concatenates several filters as Canny does. Table 5.4 shows the estimated resource usage for Census transform functions, while Table 5.2 shows the usage for shared functions between Census transform and, above explained, Canny.

Skin Tone Detection

The last evaluated graph is Skin tone detection, which requires threshold objects to produce output Boolean images. The graph is composed by 14 nodes of four different OpenVX kernels that process 8 bit data. Table 5.5 shows the resources for each function in the Skin tone graph.

5.7.3 OpenVX Application Analysis

The new HiFlipVX implementation simplifies the adoption of different FPGAs. For example, this section evaluates execution time, frequency, power, and energy on two FPGA devices: Stratix 10 GX (S10GX) and Stratix 10 MX (S10MX). The main difference between the boards is the global memory. While the S10GX has one DRAM bank with a data port width of 512 bits, the S10MX has a HBM multi-banked memory composed by 32 DRAM banks and a data port width of 256 bits per bank. Running on both boards only required to change the *vxCreateImage* port declaration.

Table 5.6 shows the execution time, frequency, power and energy of the four graphs. For all of them, the S10GX has higher frequencies and lower execution times, with time gains between 1.4 and 6,8%. Since all graphs are compute bound; e.g., in Canny edge and Census transform the maximum memory bandwidth used is 2.4GBs for S10GX and S10MX, the HBM memory does not provide any advantage in spite of using one memory bank per variable. Most probably, the same Stratix 10 architecture explains the close results. For power and energy, in all but Autocontrast, the S10GX consumes more energy and with the lower execution time increases average power, up to 18%. The higher energy consumption in Autocontrast by the S10MX may be due to the BSP differences and the required extra RAM that increases routing complexity.

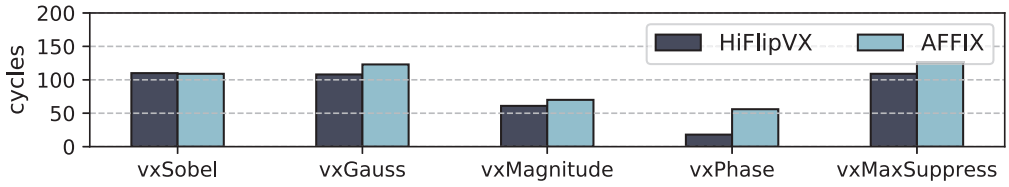


Figure 5.10: Latency of canny edge for HiFlipVX and AFFIX using an Stratix 10 GX FPGA.

5.7.4 Comparison with Existing Approaches

AFFIX [151] is a previous proposal that implements OpenVX graphs. It relies on OpenCL channels to offer an implementation based on OpenVX standard, as shown in Figure 5.1b. The use of OpenCL limits the programmability of AFFIX. Comparing the graph codes from Figure 5.11, AFFIX, lower left, relies on OpenCL macros that are error-prone, difficult to maintain, and moves away from the clarity of OpenVX,

Table 5.6: HiFlipVX results on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image.

OpenVX Application	Stratix 10 GX				Stratix 10 MX			
	Time [ms]	Frq [MHz]	Power [W]	Energy [mJ]	Time [ms]	Frq [MHz]	Power [W]	Energy [mJ]
Canny edge	6.8	310	13.2	89.8	7.3	293	11.3	76.5
Census	6.8	331	12.9	87.7	6.9	326	10.9	74.6
Autocontrast	23.1	301	13.1	302.6	23.9	294	13.8	318.1
Skin tone	33.2	343	12.8	424.9	35.7	315	10.9	361.5

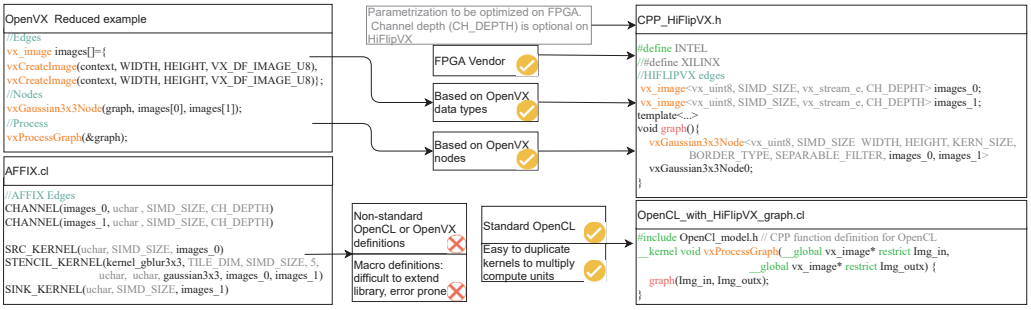


Figure 5.11: Code comparison between a reduced version of OpenVX, AFFIX, and HiFlipVX. OpenVX definitions and FPGA optimization parameters are marked in orange and grey, respectively.

upper left. In contrast, our work allows to use a well-formed C++ code, the same language as OpenVX API, to program graphs using OpenVX standard with templates to optimize hardware generation. In any case, in order to integrate HiFlipVX graphs to a host CPU, HiFlipVX can have a simple OpenCL interface called from a single queue command to execute the graph.

In order to comparatively analyse performance against our proposal, we modified AFFIX to communicate host and FPGA kernels through the on-board DRAM instead of using the Intel host pipe extension to directly communicate between the host and FPGA kernels. Although host pipes reduce latency overhead, they have two limitations: they are only supported on a few Arria 10GX development kits [66], and also, each pipe can only have one input and one output port. This second fact limits graph implementations; e.g., the Census transform was reduced to one output as shown in Figure 5.9b where the AFFIX implementation follows the dotted line and ignores the solid one. To compare with this work, it is mandatory to replace the pipes with equivalent DRAM input/output to run benchmarks on Stratix10 GX and Stratix 10 MX boards.

Table 5.7: Comparison between HiFlipVX and AFFIX on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image.

OpenVX Application	Stratix 10 GX		Stratix 10 MX	
	$\frac{Time\_HiFlipVX}{Time\_AFFIX}$	$\frac{Energy\_AFFIX}{Energy\_HiFlipVX}$	$\frac{Time\_HiFlipVX}{Time\_AFFIX}$	$\frac{Energy\_AFFIX}{Energy\_HiFlipVX}$
Canny edge	3.2	2.4	3.6	1.9
Census	3.6	1.7	3.4	1.8
Autocontrast	0.8	0.9	0.8	0.7

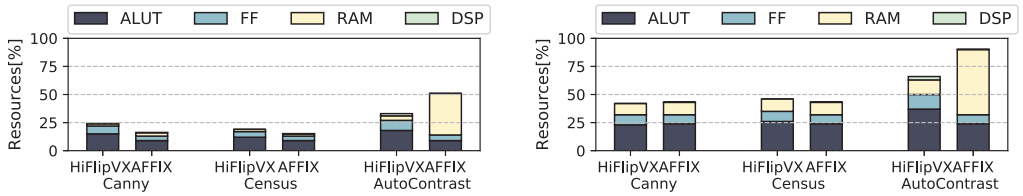
In Table 5.7, it can be observed that HiFlipVX reaches a speed-up of 3.4$\times$ and 3.6 $\times$ for Canny Edge and Census. In case of Autocontrast, it was not possible to use the same implementation for both AFFIX and ours, so that, they are hardly comparable. Our approach is behaving a 20% worse since synthesized frequency is lower in comparison to AFFIX (Figure 5.12). This penalization on frequency is due to a higher consumption of resources (RAM) in the HiFlipVX implementation.

Comparing the energy of the proposals, HiFlipVX dissipates 23% less power than AFFIX in Census transform case, our assumption is that, in HiFlipVX, the dispatch circuits to connect nodes with host are minimized[3], also, in Canny edge and Census transform the HiFlipVX frequency is lower than AFFIX, at least 50% in Canny. In Autoconstrast, with the worst performance, it is only 10% less energy efficient.

Comparing the latency between both implementations in Figure 5.10, HiFlipVX shows a 10% of improvement on average. One of the pipeline speed-up sources comes from the hyper-optimized loop structure which is enabled by default in the HLS compiler. The use of hyper-registers on an application has demonstrated a performance gain of 1.4 $\times$ on Stratix10 devices compared with previous FPGAs generation [168]. Although the compiler tries to apply this technique in both AFFIX and HiFlipVX, in case of AFFIX, the use of OpenCL channels is inhibiting this optimization.

Figure 5.12 shows the resource consumption: ALUT, RAM, FF, and DSP; of AFFIX and HiFlipVX. In case of Canny edge, Census transform, and Autocontrast, AFFIX has a higher utilization of ALUTs and FFs resources than HiFlipVX. In opposition to HiFlipVX, in AFFIX, OpenCL generates a "kernel dispatch logic" for each OpenVX kernel to communicate with the host, which is responsible for an increase of 1463 ALUTs and 1467 FFs per kernel node. In the case of HiFlipVX, kernel nodes are

[3]Compilation reports state this difference in the dispatch logic between AFFIX and HiFlipVX



(a) Stratix 10 GX **(b)** Stratix 10 MX

Figure 5.12: Resource usage per logic unit relative the total units on Stratix 10 GX and Stratix 10 MX for AFFIX and HiFlipVX implementations.

collapsed in a single kernel with a single dispatch logic with saves from 4 to 24% of resources per kernel in the evaluated graphs on the Stratix 10 GX. On the Stratix 10 MX, the difference is less than 5% between implementations.

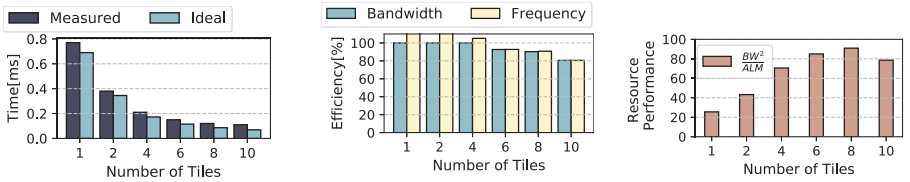
In Autocontrast, the amount of RAM resources in HiFlipVX is 4 $\times$ bigger as it is sensitive to image size. In contrast, AFFIX implementation prefers to split the pipeline and read twice from external memory instead of using RAM resources. Concerning to DSPs resources, in HiFlipVX the color conversion is implemented with a 8-bit approximation [103] that does not require DSPs for float operations in contrast to AFFIX.

At last, we compare our proposal against the traditional OpenCL model, depicted in Figure 5.1a, used by the Chai benchmark [53] for Canny edge. Our approach, HiFlipVX reaches a speedup of 9 $\times$ in comparison to Chai's. There are two limiting factors that justify this results in Chai's: communication between nodes through external memory is slower and shallow kernels (short pipelines) do not fully exploit FPGA paralelism.

5.7.5 Tiling HiFlipVX for HBM memory

The previous results of S10MX with HBM2 memory shows it has low speed up than S10GX since each memory bank has less bandwidth with the same vectorization factor. The performance can increase because each variable can be assigned to a different memory bank in S10MX, removing memory bank overhead.

In the case of memory-bound applications, as the 3x3 filters, maximizing the memory bandwidth with HBM2 technology requires increasing the vectorization two times until 128. Bit width is limited to 64 bits by data type supported in C/C++. For this reason, the kernel calculations and stream buffers are duplicated to execute all the



(a) Execution time

(b) Bandwidth and Frequency efficiency

(c) Resource performance

Figure 5.13: HiflipVX tiling with HBM2 memory bandwidth. A) Measured time and estimated ideal time. B) Memory bandwidth efficiency is the relation between measured and maximum expected memory bandwidth; and frequency efficiency is the relation between kernel frequency and minimum required frequency (400MHz). C)Resource performance is the relation between total memory bandwidth vs. logic resources usage (ALM).

pixels in one cycle; this increases the memory bandwidth from 2.4GB/s to 11.8GB/s per pseudo-channel.

The next optimization step is to increase resource usage, where the limit is the 32 pseudo-channels that can be exploited with a tiling strategy, dividing the image into rows to increase the total memory bandwidth. For example, in Sobel filter with 1 image input and 2 outputs, the hardware is multiplied 10× in steps of 2 to show the kernel performance in Figure 5.13. With the increase of the memory bank usage, the routing congestion appears and produces a frequency degradation of up to 24%. If the locations of memory bank ports are far from the PCIe port, the frequency is degraded 15% as maximum. Despite this, more than 4 tiles reduced frequency under 400MHz and with it the effective memory bandwidth as Figure 5.13b shows. With more than 8 tiles the performance benefits measured with the relation of memory bandwidth over resource usage stars to decrease as Figure 5.13c shows. The speed-up improvement from 8 to 10 tiles is only 9%, and the maximum speed-up with 10 tiles is 7×, 30% less than expected, instead of with 8 tiles, the speed-up is 6.4×, 20% less than expected. These results show bank distribution's performance dependency and the necessity of metrics for choosing the best trade-off among resource usage, floorplan distribution, and performance.

5.8 Conclusions

One of the main features of OpenVX is the portability among devices. However, on FPGA devices, providing cross-platform support remains a challenge. This chapter

presents a cross-platform OpenVX library for FPGAs based on HiFlipVX library, which originally only targeted Xilinx devices. This new version efficiently supports Intel FPGAs exploiting the novel Intel's system of tasks to coalesce OpenVX nodes into accelerated graphs on Intel FPGAs.

The new implementation introduces a novel compilation flow that integrates the expressiveness of OpenVX graphs in C/C++ with the performance of OpenCL kernels. Also, applications can interoperate with OpenCL and SyCL code. With these 3 aspects, the library gains flexibility to support multiple FPGA architectures and devices with conventional and High Bandwidth Memories.

In terms of resource utilization, on Intel devices, the enabled optimizations save around 1.5% of ALUTs usage per node in graphs versus the standard OpenCL approach with one kernel per node, since the host hardware control communication is only generated for the complete HiFlipVX graph application. Compared with the state-of-art, HiFlipVX performs up to 3.6 and 9.6 $\times$ faster than AFFIX and Chai, respectively. Energy results also reflects the successful implementations with savings up to 2.4 $\times$.

5.9 Contributions

- We porposed a new portable implementation of OpenVX for FPGAs using HiFlipVX, originally designed for Xilinx devices, providing compatibility with Intel devices. This work has been developed in collaboration with the University of Dresden, and it has been published in *29^{th} Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2019*. The repository is available in GitHub
- We extended the interoperation of HiFlipVX applications as OpenCL/ SyCL libraries to support discrete FPGA devices with either DRAM or HBM memories. This work has been submitted and is under minor review in *Journal of Systems Architecture (JSA), 2021*
- We analysed the performance and energy efficiency of different graph applications. Compared with previous OpenVX for Intel FPGA implementations, this work improves the performance $2.6\times$ and saves 1.6 $\times$ of energy on average.

6 FPGAs on Heterogeneous System for Energy Efficiency

One of the main objectives in a heterogeneous system is how to use the available devices efficiently. In the previous chapter, the focus was the FPGA as a power-efficient processor itself. In this chapter, the FPGA interacts with other devices as CPU and GPU, sharing the workload with the help of EngineCL runtime and our extended support. We present the main changes in EngineCL runtime to support the FPGA to make the cooperative execution transparent. Finally, we explore three load balancing algorithms using the CPU, GPU, and FPGA devices, evaluating performance and energy efficiency as metrics.

6.1 Introduction

The flourishing of heterogeneous systems promises better performance and energy efficiency [51]. Such heterogeneous systems are often comprised of a CPU and an accelerator, with GPU being the most widely used accelerator. GPUs have delivered an excellent performance for multiple application domains, combined with a rich software ecosystem, enables programmers to adopt them without suffering a high entry barrier. However, GPUs require substantial power dissipation that can be unaffordable in many environments. As a result, other devices such as FPGAs have emerged as complementary accelerators.

In comparison to other accelerators, FPGAs can provide a better performance to power ratio. The downside, however, is that application development on FPGAs requires knowledge of digital design, which often is the main obstacle preventing their broad adoption by programmers. To mitigate this problem, high-level synthesis frameworks with languages like C, C++, or OpenCL have emerged to improve programmer's productivity [111].

OpenCL enables parallel execution between accelerator devices without masking the hardware architecture and allows portability across devices [43]. But it does not provide any support for load balancing between them. Load balancing is critical in order to minimize execution time in heterogeneous systems. This problem has been extensively studied, specifically in the context of two device systems; CPU coupled with either a GPU, a Xeon Phi, or an FPGA [128, 105, 91, 11, 98, 121].

EngineCL is a high-level framework that provides scheduling and data management primitives on top of OpenCL, easing the programmability of heterogeneous systems [120]. While EngineCL has been successful in systems coupling a CPU with either a GPU or Xeon Phi device, it requires extentions to support FPGAs. As FPGAs have already been deployed in HPC systems, and furthermore, future systems will probably integrate more and more accelerators on a single die [24], it is important for high level frameworks to provide efficient support for FPGAs.

This chapter shows the EngineCL extension to provide FPGA support, and load balance *parallel_for* constructs among both CPU, GPU, and FPGA, so that programmers can improve performance and energy efficiency without dealing the complexities of cooperative execution and device management. Such transparent cooperative execution has entailed a substantial number of modifications in the design and implementation of EngineCL. These include the communication mechanism between the host and the FPGA, how arguments are passed to the kernel, support for different kernels for each device, and the queuing system to overlap computation and communication.

The experimental results show that heterogeneous systems deliver significant performance over using the fastest available device for all the scenarios under consideration. This conclusion even holds when the heterogeneous system is comprised of unbalanced devices. This is crucial in presence of FPGAs for kernel performance can dramatically vary depending on the kernel implementation [162]. The average improvement using the best balancing algorithm is 53.5%. On the other hand, it should be noted that the improvements in performance are not always followed by a reduction in energy consumption, for energy efficiency strongly depends on both the devices and the benchmark.

6.2 Related Work

High-level synthesis has enabled to widen the programmers audience for FPGA and its inclusion in heterogeneous systems [88, 115].

Many different applications have benefited from heterogeneous execution in a plethora of systems; e.g., DNA/RNA alignment on a CPU+GPU system [16], graph analytics on a CPU+FPGA system [182]. Even a fully heterogeneous system, CPU+GPU +FPGA, has been proposed for accelerating a real-time location problem and a pipeline HPC application [3, 139].

Load balancing is a challenging aspect of heterogeneous computing that has been widely addressed. When benchmark behavior is defined and/or remains constant, static scheduling tuned for the application tends to provide the best results. Tsoi *et al.* divide the problem between devices with an analytic model [155]. However, static load balancers require an exploration phase, and they do not adapt to unexpected changes on application throughput, which can lead to load unbalance inefficiencies. Dynamic balancers face the imbalance problem, but at the cost of potential penalties due to load balancer activity. Pandit and Govindarajan presented FluidiCL where a CPU and GPU work on a shared iteration space, and each device starts from the beginning and end of the iteration space, respectively [128]. In order to avoid load balancer penalties, Qilin, HDSS, and Concord propose to calculate the computational speed of each accelerator at runtime and then assign a single chunk of work to each accelerator [98, 9, 80]. While Qilin relies on a trained-database that provides execution-time projection for all the programs it has ever executed, HDSS and Concord rely on a brief exploration phase that computes the relative computational speed of each device. The weakness of these proposals is that they cannot be adapted to irregular applications that are addressed by the adaptive schedulers like LogFit and H-guided [158, 122, 129, 121].

MKMD maps multiple kernel into multiple devices in a two-phase approach, the first phase assigns kernels to devices and the second enables work-group level partitioning to keep all devices busy [91]. Industry solutions include Intel TBB, supporting GPU offloading with OpenCL [85], or Qualcomm Heterogeneous Compute SDK, supporting GPU and DSP offloading [135]. For an ample overview of load balancing techniques, please refer to Mittal and Vetter [105].

The most recent advances in heterogeneous systems are focused in tightly couple processor with CPU+FPGA, in this cases the scheduling polices should be careful in the selection of the chunk size to align CPU and FPGA caches [140]. Other advances are focused on modern programming frameworks as OneApi which is an alternative for single-source programming paradigm [138, 172], it is used in co-execution using CPU+GPU SoC [28] and some proposals added load balancing algorithms extending the oneAPI runtime [119, 118]

In comparison to previous works, to the best of our knowledge, ours was the first to face the load balancing problem for the *parallel_for* paradigm on a heterogeneous platform composed by three accelerators: CPU+GPU+FPGA.

6.3 EngineCL Runtime

EngineCL is a runtime that relies on OpenCL C++ back-end to communicate and program accelerators, simplifying devices programming and squeeze their performance out [120]. EngineCL provides three load balancing algorithms. It divides a single task among devices in a heterogeneous system; Also, EngineCL hides the underlying hardware details by considering different devices as a single virtual device.

Features as portability, usability, and performance are the main design pillars of EngineCL. While code portability on different devices is enable with the use of OpenCL as back-end, the programmer is responsible for managing device architecture and host communication using concepts such as platforms, devices, contexts, buffers, queues, kernels and arguments, data transfers and error control sections. As the number of devices and operations increases, the code grows quickly making it difficult to maintain, decreasing programmers' productivity. EngineCL solves these issues by providing a runtime with a higher-level API that manages all the OpenCL resources of the underlying system independently.

The runtime follows Architectural Principles with well-known Design Patterns to strengthen its flexibility. EngineCL is layered in three tiers (see Fig. 6.1): Tier-1 and Tier-2 are accessible by the programmer. The lower the Tier, the more functionalities and advanced features can be manipulated. Most programs can be implemented in EngineCL with just the Tier-1. The Tier-2 should be accessed if the programmer

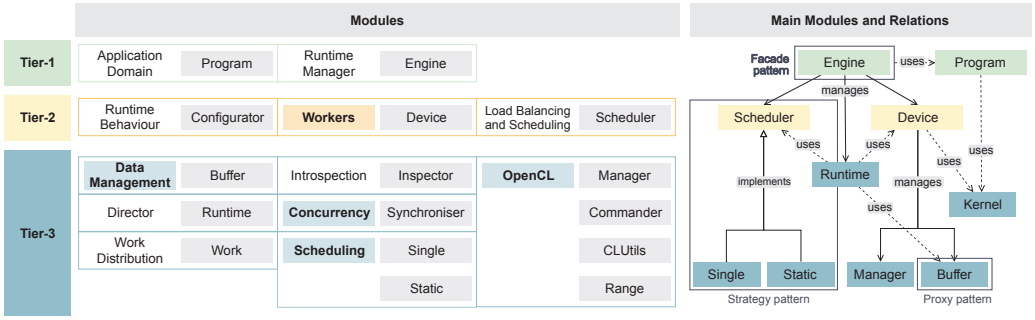


Figure 6.1: EngineCL architecture: tiers, modules and applied patterns. The highlighted modules are extended to support FPGAs.

wants to select a specific device and provide a specialized kernel or use more specific options. The Tier-3 consists of the hidden inner parts that allow a flexible system regarding memory management, pluggable schedulers, work distribution, high concurrency, and OpenCL encapsulation.

EngineCL provides high external usability and internal adaptability to support new runtime features, such as new schedulers, device types or communication-computation overlapping strategies. This is accomplished through a layered architecture and a set of core modules well profiled and encapsulated.

A set of well-known load balancing algorithms [25], described below, are provided [130]. The programmer should decide which one to use in each case, depending on the characteristics of the application and the architecture.

- **Static** This algorithm splits the data-set in as many packages as devices are in the system, proportionally to their computing capabilities. This division relies on knowing the percentage of workload assigned to each device in advance, and therefore the execution time between the devices is equalized. It minimizes the number of synchronization points, therefore, it performs well when facing regular loads. However, it is not adaptable, so its performance might not be as good with irregular loads.

- **Dynamic** It divides the data-set in packages of equal size, much more than the number of devices. A master thread in the host assigns packages to the different devices, including the CPU. This algorithm adapts to the irregular behavior of some applications. However, each package represents a synchronization point between the device and the host, where data are exchanged and a new package is launched.

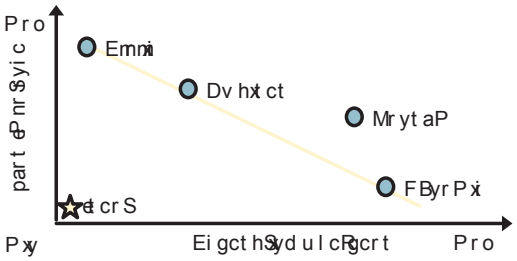


Figure 6.2: Three different alternatives of schedulers to alleviated the differences among applications and devices [25].

- **HGuided** The Heterogeneous Guided algorithm is an attempt to reduce the synchronization points of the Dynamic, while retaining its adaptiveness. It makes larger packages at the beginning and reduces the size of the subsequent ones as the execution progresses, until the minimum package size, given as a parameter, is reached. Furthermore, the size of the packet is weighted by the computing power of each device, defined as the amount of work that this device can complete in a time span. This adjusts the number of packets to achieve a more accurate load balancing than with all other algorithms.

  The size of the package for device i is calculated as follows:

$$packet\_size\_H = min\Big(Min\_package\_size, \lfloor \frac{G_r P_i}{k \sum_{j=1}^{n} P_j} \rfloor\Big) \tag{6.1}$$

  Where G_r is the number of pending work-groups in each launch, P_i is the computing power of the device i. Finally, k is a constant, between 2 and 3, and the smaller k, the faster decreases the packet size. This avoids too big package sizes when there are few devices.

These three alternative schedulers, tries to achieve as goal a minimum load imbalance among devices with the minimum overhead from the runtime. The Static, Dynamic and HGuided schudulers evidence the difficulties to achieve these goals at the same time as Figure 6.2 shows.

6.4 Coupling FPGA to EngineCL

Compared to other accelerators, FPGAs require a special work-flow for its integration into EngineCL. The main difference among FPGAs and the rest of accelerators (GPU,

Xeon Phi, ...) is the compilation process which is ahead-of-time (AOT) with an HLS tool, such as Xilinx SDSoC or Intel/Altera OpenCL SDK [146, 4]. AOT is used to reduce runtime overhead since the process takes between hours to days.

By using OpenCL, most of the boilerplate code (platform, device, context, buffers) can be shared among accelerators, which simplifies the FPGA integration into EngineCL.

EngineCL provides support for heterogeneous environments composed by CPU, GPU, and Xeon-Phi devices. The inclusion of FPGAs in EngineCL is not straightforward and requires modifications in several key points of the runtime: host device synchronization, kernel arguments management, and command queues management. Besides, to launch a kernel, EngineCL has to load the bitstream from a file to the FPGA.

Originally, EngineCL relied on asynchronous callbacks to notify kernel completion to the scheduler. Unfortunately, Intel's FPGA OpenCL runtime v17.1 requires a subsequent OpenCL function invocation to evaluate event status and call pending registered callbacks, which could not always be guaranteed in EngineCL. To operate with FPGA, callbacks are replaced with synchronous OpenCL commands managed by multiple host threads and command queues for each device.

To use the schedulers, we extend the EngineCL arguments in *workers* of Tier-2 and change data management of Tier-3, Fig. 6.1. In *workers* of Tier-2, we add iterations argument, which are the amount of work, because task-based kernels have statically defined just one work-item per execution. Originally, EngineCL copied every input data into memory of all devices, limiting the maximum problem size to the size of the smallest memory device, in our case the FPGA. To support any problem size, we have performed two modifications: 1) replacement of the offset argument with two item range (begin, end) arguments at each kernel invocation, 2) support for sending input data as required. Since each device has its own memory, the runtime keeps track of each size and sends chunks small enough to fit into the device memory.

Finally, to improve performance, we add a second command queue and two output device buffers (A and B in Fig. 6.3) to overlap memory read and computation commands as depicted in Fig. 6.3.

Each kernel invocation alternates output buffers, A and B, to avoid write-after-write hazard. Since the FPGA driver only allows one transaction over PCI-e at a time, there

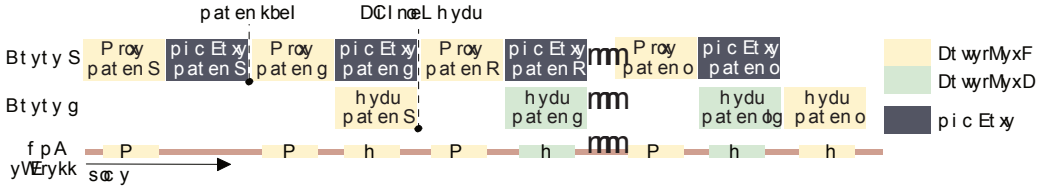
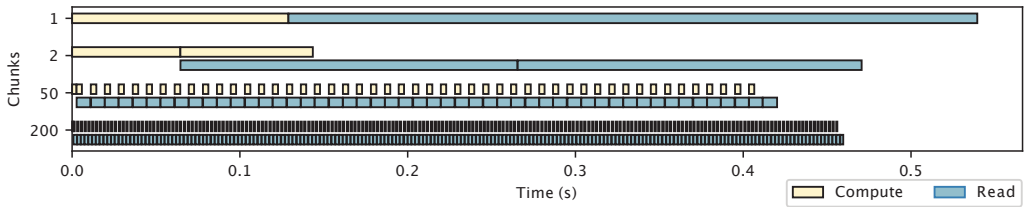
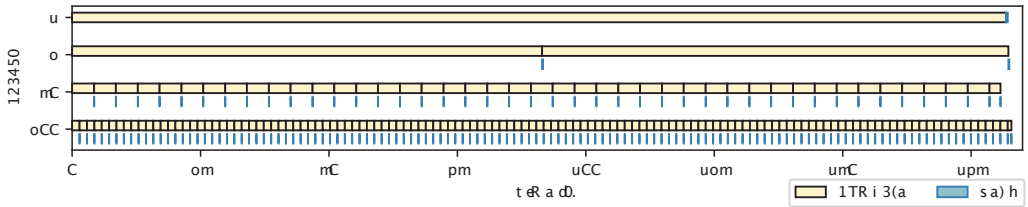


Figure 6.3: Overview of command queue overlapping with the two command queues and the two buffer sets (A and B).

is no opportunity to overlap read and write commands. To ensure overlapping with the single transaction requirement, we enqueue the write and compute of chunk i on queue 0 before enqueue the read command of chunk $i-1$ on queue 1, and then wait for both to complete with a *finish barrier* and a blocking read command, respectively. This change improves performance up to 30% (PCIe write speed is lower than the read one) for communication-bound problems and has little effect in performance-bound ones. Figure 6.4 shows the compute and communication-bound problems as references in a FPGA varying the number of chunks.



(a) Watermarking



(b) Matrix Multiplication

Figure 6.4: overlapping time performance varying the number of work load partitions(chunks) of the problem in the case of communication-bound and computation-bound problems in an FPGA.

Please note that all these changes do not entail any modification in the apps built on top of EngineCL because user-facing APIs remain the same.

6.5 Methodology

The experiments have been conducted in a heterogeneous system composed by an Intel core i7-6700k CPU (64 GB of RAM), a NVIDIA GeForce GTX TITAN X GPU (12GB of RAM), and an Altera DE5NET Stratix V GX FPGA (4GB of RAM); each device runs OpenCL version 2.0 (LINUX), 1.2 (CUDA 9.1.83), and 1.0 (Intel SDK v17.1) for the CPU, GPU, and FGPA, respectively.

Six benchmarks from different domains have been considered: Matrix Multiplication, Mersenne Twister, and Sobel Filter from the Intel Altera OpenCL repository, Watermarking and AES decrypt from the Xilinx SDAccel repository, and Nearest Neighbor from Rodinia optimized for FPGA [183]. Each benchmark has a different kernel implementation, tuned for each device. For example, FPGA devices work better with task-based kernels and very long shift register loops, while CPU and GPU perform better with NDRange kernels and smaller shift registers. But there are two exceptions, Matrix Multiplication and Nearest Neighbor that obtain the best results with an NDRange-based kernel on the FPGA. Table 6.1 shows the benchmark size (measured in work-items), main parameters, and FPGA resource utilization. In general, lower FPGA resource utilization translates into higher frequency.

Since the OpenCL FPGA runtime has a higher latency of initialization than that of the other two devices, a work-item kernel command is launched before starting the heterogeneous execution. Otherwise, the scheduler penalizes the FPGA for splitting the work.

Table 6.1: Work-items and FPGA characteristics: Clock frequency(CF), initialization interval (II), and FPGA kernel resources: adaptive logic module (AL), logic registers (LR), memory blocks (MB), and DSP.

Benchmark	Work items	CF (MHz)	II	AL %	LR %	MB %	DSP %
Matrix Multiplication	16×10^3	238.9	n/a	79	28	47	100
Mersenne Twister	22×10^7	274.4	~ 1	40	5	19	88
Watermarking	11×10^8	226.8	~ 1	16	10	15	3
Sobel filter	12×10^9	295.3	~ 1	13	8	18	0
AES	11×10^8	299.9	2	20	9	18	0
Nearest-Neighbor	40×10^8	210.8	n/a	54	19	31	94

In order to evaluate energy consumption, we rely on reading hardware counters with Intel RALP and NVIDIA system manager for the CPU and the GPU, respectively. Since the FPGA does not provide power counters, it is measured on board with the Newtons4th PPA520 power analyzer, sampling at 10^6 samples per second, and a PCIe riser card [55]. FPGA power is the sum of the power drained from the PCIe edge connector and the auxiliary 6-pin Molex connector.

6.6 Results

This section analyzes the inclusion of the FPGA inside EngineCL by exploring its 3 load balancing algorithms on a CPU+GPU+FPGA heterogeneous system. First of all, it explains how the parameters of each algorithm have been optimized. Then, the results obtained in terms of both performance and energy are analyzed.

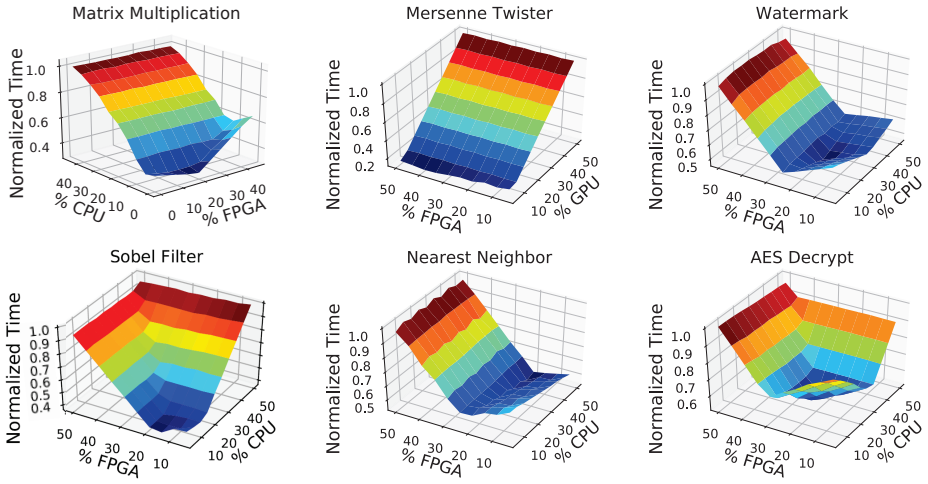


Figure 6.5: Normalized execution time to the worst for the Static scheduler. Work proportions go up to 50% for two devices, and the third device performs the remainder work.

6.6.1 Scheduler Tuning

Static

With static scheduling, the user has to choose the amount of work each device performs before starting execution. This distribution should be tuned for achieving

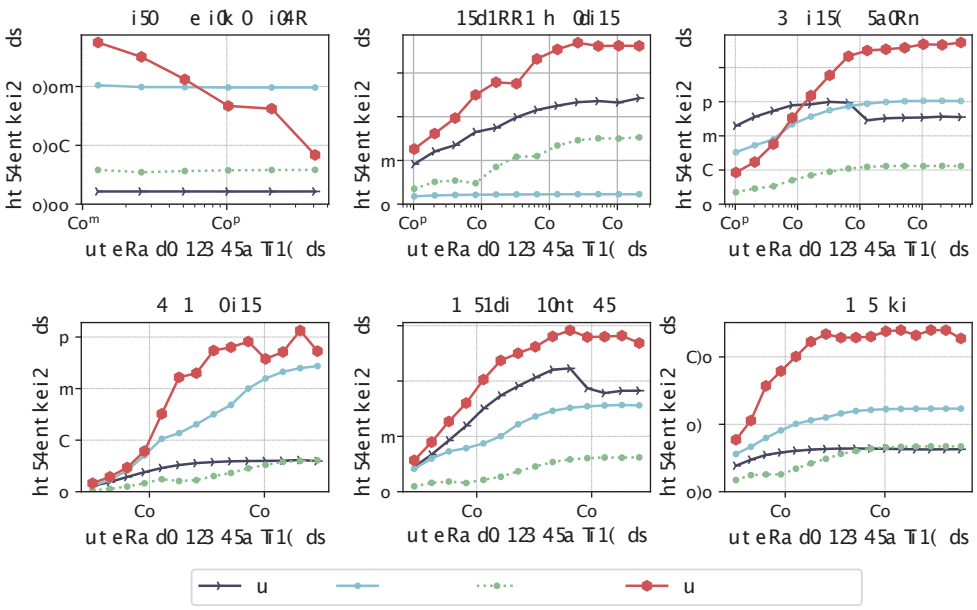


Figure 6.6: Dynamic scheduler throughput (GB/s) of CPU, GPU and FPGA with chunk size variation.

good load balancing and unfortunately, it is required an exhaustive exploration to find out the fine tuned distribution. To make matters worse, an static distribution is specific for each problem, input data and, computing device.

For instance, in this work, 100 executions per benchmark have been performed to achieve these values. Figure 6.5 shows the normalized execution time to the CPU with a workload percentage sweep in steps of 10% for two devices, while the third device takes the remainder work $(100 - (percentage_1 + percentage_2))$. Overall, setting the optimal percentages improves performance between 15 and 58%.

Dynamic

With this scheduler, each device fetches and executes chunks of work (equally-sized for all devices) until there is no work left. Figure 6.6 shows how chunk size has a significant impact on throughput, measured in gigabytes per second. In general, all benchmarks benefit from larger chunks except Matrix Multiplication. In Matrix Multiplication, the number of iterations to split is smaller than in other benchmarks, and the computational intensity is higher, so that the lower runtime overhead of smaller chunks does not pay off for the imbalance increment.

Table 6.2: Median performance improvement (PI) relative to the fastest single device execution, imbalance (IM), and average number of chunks (#C) for Static, Dynamic, and HGuided policies per benchmark.

Benchmark	Static			Dynamic			HGuided		
	PI %	IM %	#C	PI %	IM %	#C	PI %	IM %	#C
Matrix Multiplication	37.8	4.2	3	31.6	5.5	128	38.2	39.1	18.0
Mersenne Twister	43.1	2.0	20	52.5	6.4	60	45.1	8.8	25.0
Watermarking	31.3	1.4	28	70.3	0.8	280	66.6	0.0	36.4
Sobel Filter	14.9	1.1	23	13.9	0.0	11930	19.1	0.6	118.0
Nearest Neighbor	46.8	2.7	13	60.3	0.3	960	40.3	9.2	16.0
AES decrypt	58.6	4.1	23	92.7	0.0	280	96.1	0.0	94.6
Mean	38.7	2.6		53.5	2.2		50.9	9.6	

HGuided

Starts with large chunks that are automatically reduced. Two parameters tune the chunk size: computing power and minimum packet size. The computing power ratios were tuned in the static scheduler, but the sensitivity of HGuided to this parameter is small because when the minimum packet size is large enough, overall throughput remains high. This condition is relatively easy to fulfill because most benchmarks reach good performance with small chunks, as Figure 6.6 shows.

6.6.2 Scheduler comparison

Performance

From here on, all results use the best found parameters for the three schedulers. First, Table 6.2 compares them in terms of three metrics: (a) the percentage of improvement of the heterogeneous system with respect to the best single device (PI), (b) the imbalance percentage (IM), $\frac{T_{LD}-T_{FD}}{T_{LD}} \cdot 100$, where T_{FD} and T_{LD} are the execution times of the first and last devices to finish, respectively, and, (c) the total number of chunks (#C).

To begin with, heterogeneous execution achieves a substantial performance improvement in every benchmark over the best single-device execution. Comparing the three schedulers, Static obtains the worst results, even with the 100 exploratory

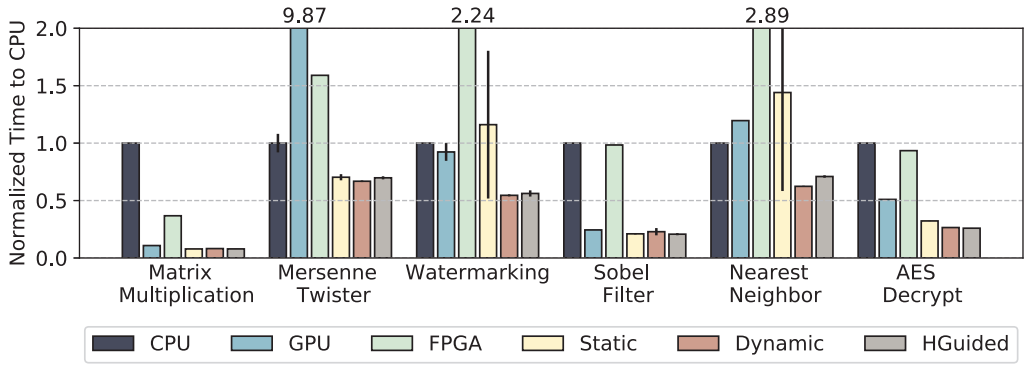


Figure 6.7: Overall median performance of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. Times are normalized to CPU in every benchmark.

executions per benchmarks, suggesting that dynamic scheduling policies should be preferred in systems with GPU and FPGA accelerators. In fact, Dynamic and HGuided reach an almost perfect balance (imbalance lower than 1%) in 4 and 3 benchmarks, respectively. Between Dynamic and HGuided, the former obtains an average better performance improvement, 53.5 vs. 50.9%, thanks to its gain in Nearest Neighbor, 60.3 vs. 40.3%. In this benchmark, the chunk size, larger than 10^7, selected by HGuided for the first chunks penalizes the overall throughput. Also, Dynamic executes many more chunks than HGuided without a big penalty in performance thanks in part to the compute and communication overlapping of the dual command queue implementation.

Finally, Figure 6.7 shows that EngineCL reduces execution time for all benchmarks, with improvements ranging between 13.9 and 96.1% for Dynamic-Sobel Filter and HGuided-AES, respectively. The reason behind those numbers is that EngineCL is able to assign the right amount of work to each device regardless its computing power for each benchmark.

Energy Consumption

Table 6.3 shows the average power for single devices running all benchmarks and analyzing four power metrics: Idle (I), which corresponds to the device sitting idle; Programmed (P), which corresponds to the power of the FPGA after it has been programmed, so P does not apply for CPU and GPU devices; and Device and Host Running (DR and HR), which corresponds to the power when a kernel is running

Table 6.3: Average power (W) for single device configurations. I, P, DR, HR represents Idle, Programmed, Device Running, and Host Running power, respectively.

	Average Power (W)								
	CPU		GPU			FPGA			
	I	DR+HR	I	DR	HR	I	P	DR	HR
Matrix Multiplication	13.6	77.7	15.8	132.1	29.2	14.3	25.1	28.8	27.7
Mersenne Twister		33.0		88.5	25.8		23.3	23.6	17.2
Watermarking		45.2		88.4	29.6		20.8	21.2	19.7
Sobel Filter		76.2		87.4	31.1		20.8	21.4	14.4
Nearest Neighbor		32.1		82.3	32.1		23.3	23.6	22.5
AES Decrypt		80.2		112.1	27.3		21.2	21.7	30.2

split between device and host, when possible (GPU and FPGA). Therefore, HR represents the power dissipated by EngineCL, the OpenCL runtime and driver.

Comparing the values, idle power keeps on the same range for the 3 devices, and both the CPU and GPU reach much higher DR values, 80.2 and 132.1W for the CPU and GPU, respectively. On the contrary, the FPGA has a lower DR of 28.8 W, but a high programming power (P), from 20.8 to 25.1W, suggesting that once programmed, running at least a small proportion of compute in FPGA could be beneficial compared to "waste" the device in programmed state. The only caveat is that HR impacts on FPGA energy efficiency because its value is on par with DR.

Figure 6.8 shows the normalized total energy compared to CPU-only energy. While in terms of performance, all three schedulers improve execution time compared to the best single-device, for energy they do not. Heterogeneous execution only improves energy consumption in Sobel Filter for HGuided.

Within the groups of benchmarks that degrade/improve energy, behavior is similar, so, for the sake of brevity, we only comment on a representative benchmark per group: Mersenne Twister and Sobel Filter. The former experiences an energy degradation around 2.5× for the heterogeneous configurations with regards to CPU because of the high GPU consumption; e.g., for the dynamic scheduler, GPU only processes 6% of the work-items and consumes 70W DR[1]. In terms of energy efficiency (work-items/joule), the GPU is around 23× worse than the CPU and the FPGA. The later, Sobel Filter, presents an opposite behavior compared with Mersenne

[1]This value corresponds to the cooperative execution and is lower than GPU-only DR, see Table 6.3, because there is less continuous work on the device.

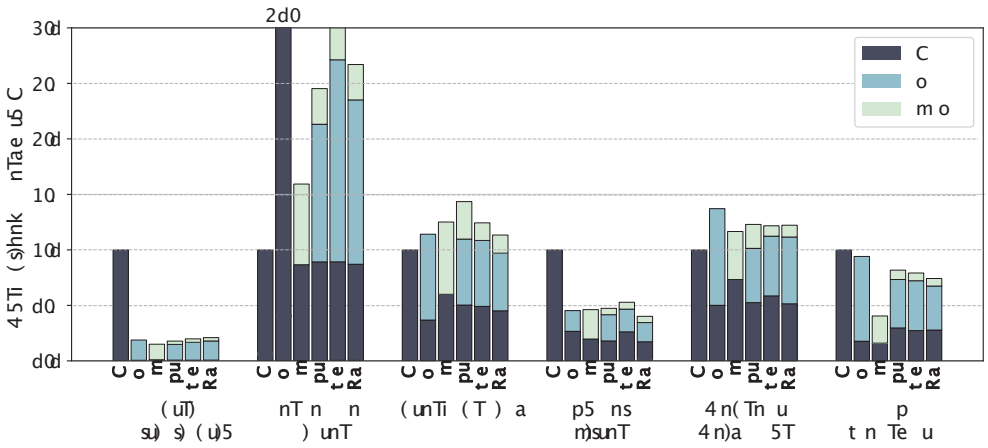


Figure 6.8: Overall energy of cooperative execution CPU(C)+GPU(G)+FPGA(F) with Static(St), Dynamic (Dy) and HGuided (HG) load balancers for each benchmark. Energy are normalized to the CPU device in every benchmark.

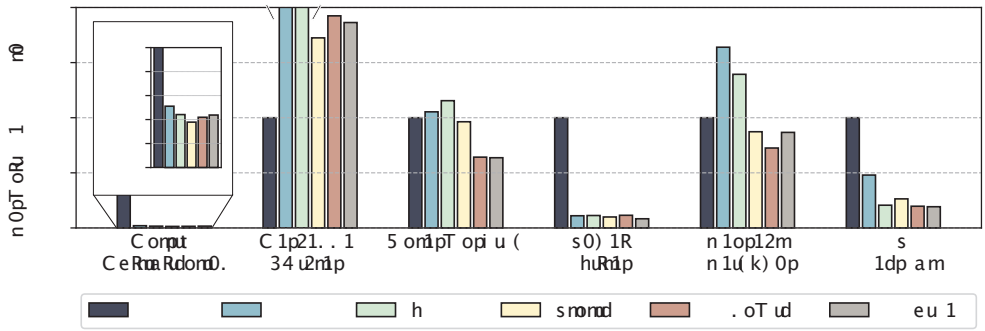


Figure 6.9: Overall normalized energy-delay product of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. The lower the better, and Matrix Multiplication results have been zoomed for clarity.

Twister. In HGuided, normalized energy improves by 60 and 11% compared with CPU and GPU only, respectively. In this case, the GPU and FPGA DR consume 80 and 21.4W and computes 65 and 17% of the work-items, respectively. For both devices, their energy-efficiency is 3× better than CPU's.

Another energy performance metric used in heterogeneous systems is the energy-delay product (EDP), as shown in Figure 6.9. The cooperative execution in Matrix Multiplication, Watermark, Sobel Filter, Nearest Neighbor and AES Decrypt improves EDP over the best single device, with a maximum improvement of 37% for HGuided in Watermarking compared to CPU. In those cases, the performance gains of multiple devices pays off for the extra energy consumption.

6.7 Conclusions

FPGAs can provide excellent performance with limited energy consumption, presenting an improvement opportunity for supercomputing systems. Nevertheless, FPGA programming with hardware description languages requires more expertise than programming other accelerators such as GPUs. Therefore, FPGA adoption requires high-level programming tools to facilitate this task.

EngineCL is an OpenCL-based framework allowing the automatic heterogeneous execution of parallel loops in multiple devices thanks to its load balancing algorithms. This article proposes an EngineCL extension to support FPGAs, so that users can cooperatively execute parallel loops in CPU+GPU+FGPA systems. To boost performance, the extension overlaps data transfer and compute operations by implementing multiple command queues and allows to execute per-device tuned kernels. And, to ease the adoption of the FPGAs, the extension does not change any user-facing APIs of EngineCL.

The results show that in CPU+GPU+FPGA systems, dynamic scheduling policies obtain better results than static ones. In fact, EngineCL provides performance improvements for all tested benchmarks, and gains range between 14.9 and 96.1% on a system with computationally unbalanced devices.

Load balancing policies do not manage to perform so well in the case of total energy consumed and energy efficiency. For energy consumption, the cooperative approach never beats the best single device except for HGuided in Sobel Filter. On the other hand, the cooperative approach is more energy efficient in 5 out of 6 benchmarks. These results indicate that it would be interesting study energy-aware load balancing policies.

Although current advances in FPGA devices could provide improved performance and energy efficiency in a heterogeneous system, driver support and FPGA-CPU compatibility limits the exploration of this proposal as Appendix 8 shows.

6.8 Contributions

- We extended EngineCL to add FPGA support without changing user-facing APIs, so that FPGAs can effectively cooperate with other hardware accelerators in the co-execution of a single massive data-parallel kernel.
- We improved the implementation in several aspects via addition of: (a) host-device synchronization, (b) command queues management, (c) mechanisms to better overlap computation with communication, and (d) runtime workers to allow task kernels. This work has been developed in collaboration with the University of Cantabria and it has been published in *Proceedings of the 18th International Conference on Computational and Mathematical Method in Science and Engineering, CMMSE2018* and *International Conference on High Performance Computing Simu-lation (HPCS), 2018*.
- We carried out an exhaustive evaluation of both co-execution and the load-balancing algorithms on a 3-device heterogeneous platform. In these experiments three different metrics have been evaluated: performance, energy consumption, and energy-delay product. This work has been published in *The Journal of Supercomputing, 2019*.

7 Conclusions and Future Work

This final chapter resumes the main findings of this dissertation and sums up the open future research.

7.1 Conclusions

The FPGAs as accelerator devices are becoming widely used in HPC due to their flexibility, low latency, high performance, and energy efficiency. But their fully adoption in the HPC domain still presents some challenges mainly in the programmability area.

Although in this thesis we presented our contributions from FPGA hardware to FPGA inclusion in the heterogeneous system, the chronological order of the research was the opposite. We started including the FPGA in a heterogeneous system as co-executor with CPU and GPU devices. This analysis showed that the FPGA inclusion is not straightforward since the FPGA programming flow is more complex than other well-known devices such as CPU and GPU. Even with high-level synthesis, which promises an increase in productivity, programming, and optimizing, kernels is complex, time expensive, and has limited driver support. After, this thesis focused on reducing the gap among programmers and FPGA devices with multiple contributions such as predictive models and high-level frameworks that ease the adoption of FPGAs by improving their productivity, necessary to face the post-Moore era.

For improving the FPGA programming flow, we analyzed multiple FPGA applications from a set of representative benchmark suites and existing analytical models. From the analysis, we observed that around 52 % of them are memory bound. Also, many tools overlook the performance effects of external memory and their interaction with HLS generated components as the Global Memory Interconnect. We performed a qualitative analysis of the FPGA architecture used for the Intel OpenCL SDK and extracted an analytical model from each type of memory access to accurately predict kernel performance. The advantage of the new model lies in the fact that it only

requires information of an early FPGA compilation stage before the time expensive place and route process, potentially increasing the kernel optimization process. Also, our analytical model extracts main hints to guide programmers during the optimization process, focusing their efforts to memory optimizations where the FPGA has enough resources.

Designing applications for FPGAs requires their seamless integration with standards that programmers already use for cross-platform acceleration. Higher abstraction levels provided by these standards help to improve the programmability, but, for FPGA, there is a gap between them and the bitstream generation process. High-level Synthesis provides a way to raise the abstraction and hide HDL nuances. Unfortunately, FPGA basic blocks and compiler translations from high-level languages to RTL are different among the main vendors such as Xilinx and Intel.

With the objective to standardize and achieve portability among FPGA devices, our proposal is based on the OpenVX standard for computer vision applications. Libraries developed for Xilinx devices are not efficient in Intel devices. For this reason, we extended the support to Intel FPGA devices based on the system of tasks to abstract the synchronization and hardware customization. Also, it is coupled to OpenCL, enabling a portable and flexible library. The proposal improves logic usage since it integrates the complete graphs with a single kernel call, and performs up to 3.6× better, and saves energy up to 2.4× compared with the state-of-art.

The final contribution of this dissertation raises the abstraction level further, including the FPGA support as a co-executor in a heterogeneous system with a CPU and GPU devices. In this part, the FPGA is coupled to a high-level runtime for load balancing, EngineCL. Also, the inclusion of double buffering strategies improves one of the main bottlenecks of accelerators, as is the host-device communication. As a result, we have presented one of the first heterogeneous systems with three devices: CPU, GPU, and FPGA.

Our proposals suppose a significant benefit in FPGA programmability promising a better adoption of the FPGA devices, also demonstrating by the analysis of energy and speed-up gains in a heterogeneous system. This makes FPGAs a promising options that still need research efforts to be a broadly included in the HPC computation.

7.2 Future Work

The research effort of this dissertation can be continued by the hand of technological advances and demands of nearby computing:

- The next step for the proposed analytical model for memory bound applications is the integration with the intermediate reports and the possibility to add the prediction to load balancing techniques to reduce unbalancing and runtime overhead.
- HiFlipVX library was integrated as a portable library for FPGAs and since computer vision graphs can be enabled to share the work load, two approaches can be followed: use the library for sharing data work load or task node graph compute distribution for balancing with another devices. This is especially interesting for a GPU finding the trade-off among energy and performance efficiency.
- With the inclusion of FPGA support in EngineCL and with the upgrades of FPGA with memory improvements such as HBM2 memory, logic resources and improvements in driver support, opens the door for future load balancing policies research not only looking for performance balance, but also to power, looking energy improvements in the heterogeneous system with co-execution.

The FPGAs are evolving, providing even more capabilities to enhance systems. However, features such as HBM memory or high-speed interfaces require more modular designs, which abstract the hardware complexities and ease the programming with designs oriented to be increasingly cooperative with other devices to fully exploit their capabilities.

From host-interconnection to on-chip FPGA-interconnection is identified as a potential feature to be improved in the FPGAs. For example, PCIe interconnection with the host limits the maximum performance of FPGA with HPC workloads. Although double buffering strategies help improve this, faster interconnections should be researched as well as novel CPU-FPGA coherent communications.

Besides, the advances in silicon technologies such as HBM2 memories force the rethought of the internal FPGA-interconnection. The current support demands programmers' knowledge of the physical HBM banks distribution in the FPGA die to optimize interconnections since they have strong routing and resource usage

dependencies, limiting the maximum bandwidth, and, then, kernel performance. Implementing strategies as fixed interconnection with memory controllers unifying memory space can improve memory performance and simplify the programming decisions about data buffer assignation.

While FPGA programmability progress has undoubtedly been done in the recent years, there is still a long way to go. The software improvements should be go hand-in-hand with hardware. FPGA programs require to advance in a way that the hardware adapts to the software, and not as it currently occurs, where the programmers have to adapt software to the FPGA architecture. The use of frameworks eliminates the barrier, but still requires specialized programming knowledge. This thesis contributes with an small step towards the adoption of FPGAs with multiple programability improvements.

Bibliography

[1] Ashwin M. Aji, Antonio J. Peña, Pavan Balaji, and Wu chun Feng. "MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL". In: *Parallel Computing* 58 (2016), pp. 37–55 (cit. on p. 6).

[2] G. Akgün, L. Kalms, and D. Göhringer. "Resource Efficient Dynamic Voltage and Frequency Scaling on Xilinx FPGAs". In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, Apr. 2020, pp. 178–192 (cit. on pp. 65, 68).

[3] Mohammad Alawieh, Maximilian Kasparek, Norbert Franke, and Jochen Hupfer. "A high performance FPGA-GPU-CPU platform for a real-time locating system". In: *2015 23rd European Signal Processing Conference (EUSIPCO)*. 2015, pp. 1576–1580 (cit. on p. 89).

[4] *Altera SDK for OpenCL Programming Guide*. 2017 (cit. on p. 93).

[5] Sam Amiri, Mohammad Hosseinabady, Andres Rodriguez, et al. "Workload Partitioning Strategy for Improved Parallelism on FPGA-CPU Heterogeneous Chips". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 376–3764 (cit. on p. 6).

[6] Andrey Andreev, Evgueni Doukhnitch, Vitaly Egunov, et al. "Evaluation of Hardware Implementations of CORDIC-Like Algorithms in FPGA Using OpenCL Kernels". In: *Knowledge-Based Software Engineering*. Ed. by Alla Kravets, Maxim Shcherbakov, Marina Kultsova, and Tadashi Iijima. Cham: Springer International Publishing, 2014, pp. 228–242 (cit. on p. 17).

[7] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. "Performance comparison of FPGA, GPU and CPU in image processing". In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 126–131 (cit. on p. 12).

[8] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. "An OpenCL™ Deep Learning Accelerator on Arria 10". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, 55–64 (cit. on p. 17).

[9] Belviranli, M. E. et al. "A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures". In: *ACM Transactions on Architecture and Code Optimization* 9.4 (Jan. 2013), 57:1–57:20 (cit. on p. 89).

[10] Meena Belwal, Madhura Purnaprajna, and Sudarshan TSB. "Enabling seamless execution on hybrid CPU/FPGA systems: Challenges amp; directions". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8 (cit. on p. 15).

[11] Alécio P. D. Binotto, Carlos E. Pereira, and Dieter W. Fellner. “Towards dynamic reconfigurable load-balancing for hybrid desktop platforms”. In: (2010), pp. 1–4 (cit. on p. 88).

[12] Andrew Boutros and Vaughn Betz. “FPGA Architecture: Principles and Progression”. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29 (cit. on pp. 16, 17, 19).

[13] I. Buck. “GPU computing with NVIDIA CUDA”. In: *ACM SIGGRAPH 2007 Papers - International Conference on Computer Graphics and Interactive Techniques* (2007) (cit. on p. 14).

[14] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, et al. “A Cloud-Scale Acceleration Architecture”. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016 (cit. on p. 13).

[15] Doris Chen and Deshanand Singh. “Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAs for information filtering”. In: *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012* (2012), pp. 5–12 (cit. on p. 17).

[16] Chen X. et al. “CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment”. In: *BMC Bioinformatics*. 2017 (cit. on p. 89).

[17] Artem Chikin, Jose Nelson Amaral, Karim Ali, and Ettore Tiotto. “Toward an analytical performance model to select between GPU and CPU Execution”. In: *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019* (2019), pp. 353–362 (cit. on p. 15).

[18] Derek Chiou. “The microsoft catapult project”. In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 2017, pp. 124–124 (cit. on p. 6).

[19] Hyojin Choi, Jongbok Lee, and Wonyong Sung. “Memory access pattern-aware DRAM performance model for multi-core systems”. In: *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. 2011, pp. 66–75 (cit. on pp. 36, 41).

[20] Young-kyu Choi, Jason Cong, Zhenman Fang, et al. “A quantitative analysis on microarchitectures of modern CPU-FPGA platforms”. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6 (cit. on p. 19).

[21] Young Kyu Choi, Peng Zhang, Peng Li, and Jason Cong. “HLScope+,: Fast and accurate performance estimation for FPGA HLS”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017 (cit. on pp. 19, 34–36, 40, 41, 45, 46, 51, 59).

[22] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. *A DSL Compiler for Accelerating Image Processing Pipelines on FPGAS*. 2016 (cit. on p. 64).

[23] E. Chung, J. Fowers, K. Ovtcharov, et al. “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave”. In: *IEEE Micro* 38.2 (2018), pp. 8–20 (cit. on p. 18).

[24]Chung, E. S. et al. “Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?” In: *Proc. of the 43rd Ann. Int. Symp. on Microarchitecture*. MICRO ’43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 225–236 (cit. on p. 88).

[25]Florina M. Ciorba, Christian Iwainsky, and Patrick Buder. *OpenMP Loop Scheduling Revisited: Making a Case for More Schedules*. 2018. arXiv: `1809.03188 [cs.DC]` (cit. on pp. 91, 92).

[26]J. Cong, B. Liu, S. Neuendorffer, et al. “High-Level Synthesis for FPGAs: From Prototyping to Deployment”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491 (cit. on p. 19).

[27]Jason Cong, Zhenman Fang, Michael Lo, et al. “Understanding Performance Differences of FPGAs and GPUs”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 93–96 (cit. on p. 35).

[28]Denisa-Andreea Constantinescu, Angeles Navarro, Francisco Corbera, Juan-Antonio Fernández-Madrigal, and Rafael Asenjo. “Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SoCs”. In: *The Journal of Supercomputing* 77.1 (2021), pp. 44–65 (cit. on p. 90).

[29]Stephen Craven and Peter Athanas. “Examining the viability of FPGA supercomputing”. In: *Eurasip Journal on Embedded Systems* 2007 (2007) (cit. on p. 11).

[30]Gabor Csordas, Mikhail Asiatici, and Paolo Ienne. “In search of lost bandwidth: Extensive reordering of DRAM accesses on FPGA”. In: *2019 International Conference on Field-Programmable Technology, ICFPT* (2019) (cit. on p. 35).

[31]Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, et al. “From OpenCL to high-performance hardware on FPGAs”. In: *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012* (2012), pp. 531–534 (cit. on p. 17).

[32]Bruno Da Silva, An Braeken, Erik H. D’Hollander, and Abdellah Touhafi. “Performance and resource modeling for FPGAs using high-level synthesis tools”. In: *Advances in Parallel Computing*. 2014 (cit. on p. 35).

[33]Maria Angélica Dávila-Guzmán, Raúl Nozal, Rubén Gran, et al. “First Steps Towards CPU, GPU, and FPGA Parallel Execution with EngineCL”. In: *Proceedings of the 18th International Conference on Computational and Mathematical Method in Science and Engineering, CMMSE 2018* (2018) (cit. on p. 15).

[34]María Angélica Dávila Guzmán, Raúl Nozal, Rubén Gran Tejero, et al. “Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL”. In: *The Journal of Supercomputing* 75.3 (2019), pp. 1732–1746 (cit. on p. 14).

[35]R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268 (cit. on p. 5).

[36] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), pp. 1369–1386 (cit. on p. 14).

[37] A. Duran, E. Ayguadé, R. M. Badia, et al. “OmpSs: A proposal for programming heterogeneous multi-core architectures”. In: *Parallel Processing Letters* 21.2 (2011), pp. 173–193 (cit. on pp. 14, 15).

[38] Maria Angélica Dávila-Guzmán, Rubén Gran Tejero, María Villarroya-Gaudó, et al. “A Cross-Platform OpenVX Library for FPGA Accelerators”. In: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2021, pp. 75–83 (cit. on pp. 58, 71).

[39] Fernando A Escobar, Xin Chang, and Carlos Valderrama. “Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC”. In: *IEEE Transactions On Parallel And Distributed Systems* 9219.c (2015), pp. 1–18 (cit. on p. 14).

[40] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), 365–376 (cit. on p. 12).

[41] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, et al. “A Configurable Cloud-Scale DNN Processor for Real-Time AI”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 1–14 (cit. on p. 18).

[42] B. Gaster, L. Howes, D. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing With OpenCL*. Heterogeneous Computing with OpenCL. Morgan Kaufmann, 2012 (cit. on p. 14).

[43] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011 (cit. on p. 88).

[44] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. “Spector: An OpenCL FPGA benchmark suite”. In: *2016 International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 141–148 (cit. on p. 34).

[45] R. Giduthuri and K. Pulli. “OpenVX: A Framework for Accelerating Computer Vision”. In: *SIGGRAPH ASIA 2016 Courses*. 2016, 14:1–14:50 (cit. on pp. 63, 65, 73).

[46] Juan Gómez-Luna, Izzat El Hajj, Victor Chang Li-Wen Garcia-Flores, et al. “Chai: Collaborative Heterogeneous Applications for Integrated-architectures”. In: *ISPASS*. IEEE. 2017 (cit. on pp. 51, 58).

[47] V. Grover. *Parallel programming with OpenACC*. 2012, pp. 315–337 (cit. on p. 14).

[48] Amir HajiRassouliha, Andrew J. Taberner, Martyn P. Nash, and Poul M.F. Nielsen. “Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms”. In: *Signal Processing: Image Communication* 68.June (2018), pp. 101–119 (cit. on pp. 18, 64, 65).

[49] HardwareBee. *Xilinx vs. Intel High-End FPGA Series Comparison*. 2020 (cit. on p. 70).

[50]Kenneth Hill, Stefan Craciun, Alan George, and Herman Lam. “Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA”. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2015, pp. 189–193 (cit. on p. 17).

[51]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)*. 2014, pp. 10–14 (cit. on p. 87).

[52]HSA Fundation. *Heterogeneous System Architecture*. 2021 (cit. on pp. 12, 13).

[53]Sitao Huang, Li-Wen Chang, Izzat El Hajj, et al. “Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures”. In: *ICPE ’19*. 2019, 79—90 (cit. on pp. 6, 15, 17, 83).

[54]Hwu Wen-mei. *Programming Massively Parallel Processors: A Hands-on Approachl*. Morgan Kaufmann, 2010 (cit. on p. 13).

[55]Francisco D. Igual, Luis M. Jara, José Ignacio Gómez Pérez, Luis Piñuel, and Manuel Prieto-Matías. “A power measurement environment for PCIe accelerators”. In: *Computer Science - R&D* 30.2 (2015), pp. 115–124 (cit. on p. 96).

[56]Intel. *Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface*. 2018 (cit. on p. 6).

[57]Intel. *Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads*. 2021 (cit. on p. 31).

[58]Intel. *Detecting Memory Bandwidth Saturation in Threaded Applications*. 2010 (cit. on p. 36).

[59]Intel. *External Memory Interface Handbook Volume 3: Reference Material*. 2017 (cit. on p. 37).

[60]Intel. *External Memory Interfaces Intel ® Stratix ® 10 FPGA IP User Guide*. 2019 (cit. on pp. 45, 52).

[61]Intel. *High Bandwidth Memory (HBM2) Interface Intel FPGA IP User Guide*. 2019 (cit. on p. 18).

[62]Intel. *Intel Acquisiton of Altera*. 2015 (cit. on p. 6).

[63]Intel. *Intel Agilex FPGA and SoC*. 2021 (cit. on p. 18).

[64]Intel. *Intel FPGA SDK for OpenCL: Intel Stratix 10 GX FPGA Development Kit Reference Platform Porting Guide*. 2019 (cit. on p. 28).

[65]Intel. *Intel FPGA SDK for OpenCL Pro Edition: Getting Started Guide 19.1*. 2019 (cit. on pp. 31, 34, 36–39, 48, 58).

[66]Intel. *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide 19.4*. 2020 (cit. on p. 81).

[67]Intel. *Intel High Level Synthesis Compiler Pro Edition: Reference Manual*. 2019 (cit. on pp. 13, 37, 46).

[68] Intel. *Intel® Agilex® I-Series SoC FPGA Product Table*. 2019 (cit. on p. 40).

[69] Intel. *Intel® Hyperlflex Architecture High-Performance Design Handbook*. 2019 (cit. on p. 18).

[70] Intel. *Intel® Stratix® 10 Logic Array Blocks and Adaptive Logic Modules User Guide*. 2020 (cit. on p. 16).

[71] Intel. *Intel® Stratix® 10 TX Product Table*. 2019 (cit. on pp. 16, 40).

[72] Intel. *Intel® StratixV Device Overview*. 2019 (cit. on p. 17).

[73] Intel. *Intel® High Level Synthesis Compiler Pro Edition 19.4, Best Practice Guide*. 2020 (cit. on p. 71).

[74] Intel. *Running Average Power Limit Energy Reporting*. 2019 (cit. on p. 30).

[75] Jiantong Jiang, Zeke Wang, Xue Liu, et al. "Boyi: A systematic framework for automatically deciding the right execution model of OpenCL applications on FPGAs". In: *FPGA 2020* (2020) (cit. on pp. 18, 35, 40, 58, 65).

[76] Z. Jin and H. Finkel. "Optimizing an Atomics-Based Reduction Kernel on OpenCL FPGA Platform". In: *IPDPSW*. 2018, pp. 532–539 (cit. on p. 40).

[77] G. Juckeland, W. Brantley, S. Chandrasekaran, et al. *SPEC ACCEL: A standard application suite for measuring hardware accelerator performance*. Vol. 8966. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, 2015, pp. 46–67 (cit. on p. 14).

[78] David Kaeli, Perhaad Mistry, Dana Schaa, and Dong Zhang Ping. *Heterogeneous computing with OpenCL 2.0*. Vol. 148. Morgan Kaufmann, 2015, pp. 148–162 (cit. on pp. 12, 32).

[79] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, et al. "Adaptive heterogeneous scheduling for integrated GPUs". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14* (2014), pp. 151–162 (cit. on p. 15).

[80] Kaleem,R. et al. "Adaptive Heterogeneous Scheduling for Integrated GPUs". In: *PACT*. New York, NY, USA: ACM, 2014, pp. 151–162 (cit. on p. 89).

[81] L. Kalms, P. Amini Rad, M. Ali, and A. Iskander D. Göhringer. "A Parametrizable High-Level Synthesis Library for Accelerating Neural Networks on FPGAs". In: *Journal of Signal Processing Systems* (Feb. 2021), pp. 1–27 (cit. on p. 67).

[82] L. Kalms and D. Göhringer. "Accelerated High-level Synthesis Feature Detection for FPGAs using HiFlipVX". In: *Towards Ubiquitous Low-power Image Processing Platforms*. Springer International Publishing, Jan. 2021, pp. 115–135 (cit. on p. 67).

[83] L. Kalms, A. Podlubne, and D. Göhringer. "HiFlipVX: an Open Source High-Level Synthesis FPGA Library for Image Processing". In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, Apr. 2019, pp. 149–164 (cit. on pp. 19, 64, 67, 68, 70, 71).

[84]Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. “High Bandwidth Memory on FPGAs: A Data Analytics Perspective”. In: *FPL* (2020). eprint: 2004.01635 (cit. on p. 41).

[85]Katranovet, A. et al. “Intel Threading Building Block (TBB) Flow Graph As a Software Infrastructure Layer for OpenCL-based Computations”. In: *ACM IWOCL*. Vienna, Austria, 2016, 9:1–9:3 (cit. on p. 89).

[86]T. Kenter, G. Mahale, S. Alhaddad, et al. “OpenCL-Based FPGA Design to Accelerate the Nodal Discontinuous Galerkin Method for Unstructured Meshes”. In: *Proceedings - 26th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018*. 2018, pp. 186–196 (cit. on p. 17).

[87]Jungwon Kim, Sangmin Seo, Jun Lee, et al. “SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters”. In: *Proceedings of the 26th ACM international conference on Supercomputing - ICS ’12* (2012), p. 341 (cit. on p. 14).

[88]Dirk Koch and et al., eds. *FPGAs for Software Programmers*. Springer, Cham, 2016 (cit. on p. 89).

[89]J. Kuskin, D. Ofelt, M. Heinrich, et al. “The Stanford FLASH Multiprocessor”. In: *SIGARCH Comput. Archit. News* 22.2 (Apr. 1994), 302–313 (cit. on p. 12).

[90]Brock J. LaMeres. *Quick Start Guide to Verilog*. Springer, 2019 (cit. on p. 19).

[91]Janghaeng Lee and et al. “Orchestrating Multiple Data-Parallel Kernels on Multiple Devices”. In: *Intl. Conf. on Parallel Architectures and Compilation Techniques*. 2016, pp. 355–366 (cit. on pp. 88, 89).

[92]Jiajie Li, Yuze Chi, and Jason Cong. “HeteroHalide: From image processing DSL to efficient FPGA acceleration”. In: *FPGA 2020 - 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2020), pp. 51–57 (cit. on p. 64).

[93]Shang Li, Dhiraj Reddy, and Bruce Jacob. “A Performance And Power Comparison of Modern High-Speed DRAM Architectures”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS ’18. Association for Computing Machinery, 2018, 341–353 (cit. on pp. 41, 49, 50).

[94]Y. Liang, S. Wang, and W. Zhang. “FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs”. In: *IEEE Transactions on Computers* 67.12 (2018), pp. 1750–1764 (cit. on pp. 6, 15, 19, 34, 35, 40).

[95]Yun Liang, Kyle Rupnow, Yinan Li, et al. “High-level synthesis: Productivity, performance, and software constraints”. In: *Journal of Electrical and Computer Engineering* 2012 (2012) (cit. on pp. 20, 23).

[96]S.-W. Liao, S.-Y. Kuang, C.-L. Kao, and C.-H. Tu. “A Halide-based Synergistic Computing Framework for Heterogeneous Systems”. In: *Journal of Signal Processing Systems* (2017) (cit. on pp. 6, 15).

[97]David G. Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”. In: *Int. J. Comput. Vision* 60.2 (Nov. 2004), 91–110 (cit. on p. 75).

[98] Luk, C.-K. et al. “Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping”. In: *IEEE/ACM Micro-42* (2009), p. 45 (cit. on pp. 88, 89).

[99] Grant Martin and Gary Smith. “High-Level Synthesis: Past, Present, and Future”. In: *IEEE Design Test of Computers* 26.4 (2009), pp. 18–25 (cit. on pp. 6, 20).

[100] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. “FBLAS: Streaming Linear Algebra on FPGA”. In: *CoRR* (2019) (cit. on pp. 51, 58).

[101] Michael Parker. *Understanding Peak Floating-Point Performance Claims*. Intel White Paper. 2017 (cit. on p. 28).

[102] Micron-Technology. *DDRA SDRAM MT40A2G4*. 2015 (cit. on pp. 28, 49, 50).

[103] Microsoft. *Recommended 8-Bit YUV Formats for Video Rendering*. 2018 (cit. on p. 83).

[104] Waldrop Mitchell. “More Than Moore”. In: *Nature* 530 (2016), p. 145 (cit. on p. 5).

[105] Sparsh and Mittal. “A Survey of CPU-GPU Heterogeneous Computing Techniques”. In: *ACM Computing Surveys* 47.4 (2015), pp. 1–35 (cit. on pp. 88, 89).

[106] Momeni, A. et al. “Hardware thread reordering to boost OpenCL throughput on FPGAs”. In: *ICCD*. Oct. 2016, pp. 257–264 (cit. on p. 23).

[107] "Monitor Insider". *HBM2 Deep Dive*. 2016 (cit. on pp. 18, 49, 50).

[108] Gordon E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (1965), pp. 33–35 (cit. on pp. 5, 12).

[109] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Erik Hochapfel, and Sandrine Vaton. “Energy-efficient FPGA implementation for binomial option pricing using OpenCL”. In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–6 (cit. on p. 17).

[110] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 2011, pp. 1–83 (cit. on p. 14).

[111] Muslim, F. B. et al. “Efficient FPGA Implementation of OpenCL High-Performance Computing Applications via High-Level Synthesis”. In: *IEEE Access* 5 (2017) (cit. on p. 87).

[112] S. W. Nabi and W. Vanderbauwhede. “MP-STREAM: A Memory Performance Benchmark for Design Space Exploration on Heterogeneous HPC Devices”. In: *IPDPSW*. May 2018 (cit. on p. 35).

[113] Syed Waqar Nabi and Wim Vanderbauwhede. “FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis”. In: *Journal of Parallel and Distributed Computing* (2016) (cit. on pp. 17, 35).

[114] Razvan Nane, Vlad Mihai Sima, Christian Pilato, et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604 (cit. on p. 20).

[115]Nane, R, et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (Oct. 2016), pp. 1591–1604 (cit. on p. 89).

[116]Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. “Heterogeneous parallel_for Template for CPU–GPU Chips”. In: *International Journal of Parallel Programming* 47.2 (2019), pp. 213–233 (cit. on p. 15).

[117]Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. “Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures”. In: *The Journal of Supercomputing* 70.2 (2014), pp. 756–771 (cit. on p. 6).

[118]Raúl Nozal and Jose Luis Bosque. “Exploiting Co-execution with OneAPI: Heterogeneity from a Modern Perspective”. In: *Euro-Par 2021: Parallel Processing*. Ed. by Leonel Sousa, Nuno Roma, and Pedro Tomás. Cham: Springer International Publishing, 2021, pp. 501–516 (cit. on p. 90).

[119]Raúl Nozal and Jose Luis Bosque. “Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime”. In: *Electronics* 10.19 (2021) (cit. on p. 90).

[120]Nozal, R. et al. “EngineCL: Usability and Performance in Heterogeneous Computing”. In: *arXiv* (May 2018). arXiv: 1805.02755 (cit. on pp. 6, 14, 15, 88, 90).

[121]Nozal, R. et al. “Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels”. In: *The Journal of Supercomputing* (Mar. 2018) (cit. on pp. 88, 89).

[122]Jose et. al. Nunez-Yanez. “Simultaneous multiprocessing in a software-defined heterogeneous FPGA”. In: *The Journal of Supercomputing* (Apr. 2018) (cit. on p. 89).

[123]Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, et al. “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?” In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, 5–14 (cit. on p. 18).

[124]NVIDIA. *NVIDIA OpenCL SDK Code Samples*. 2020 (cit. on p. 58).

[125]K. O’Neal and P. Brisk. “Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey”. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2018 (cit. on p. 59).

[126]OpenACC. *OpenACC. More Science. Less Programming*. 2021 (cit. on p. 14).

[127]M. Akif Özkan, Burak Ok, Bo Qiao, Jürgen Teich, and Frank Hannig. “HipaccVX: wedding of OpenVX and DSL-based code generation”. In: *Journal of Real-Time Image Processing* (2020) (cit. on p. 64).

[128]Pandit, P. et al. “Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014) (cit. on pp. 6, 88, 89).

[129] Borja et. al. Pérez. “Energy efficiency of load balancing for data-parallel applications in heterogeneous systems”. In: *The Journal of Supercomputing* 73.1 (Jan. 2017), pp. 330–342 (cit. on p. 89).

[130] Pérez, B. et al. “Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems”. In: *GPGPU*. New York, NY, USA: ACM, 2016, pp. 42–51 (cit. on p. 91).

[131] A. Podlubne, J. Haase, L. Kalms, et al. “Low power image processing applications on FPGAs using dynamic voltage scaling and partial reconfiguration”. In: *International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Oct. 2018, pp. 64–69 (cit. on pp. 65, 68).

[132] Jing Pu, Steven Bell, Xuan Yang, et al. “Programming heterogeneous systems from an image processing DSL”. In: *ACM Transactions on Architecture and Code Optimization* 14.3 (2017), pp. 1–25. arXiv: 1610.09405 (cit. on p. 64).

[133] Andrew Putnam, Dave Bennett, Eric Dellinger, et al. “CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures”. In: *2008 International Conference on Field Programmable Logic and Applications*. 2008, pp. 173–178 (cit. on p. 15).

[134] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, et al. “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”. In: *IEEE Micro* 35.3 (2015), pp. 10–22 (cit. on p. 13).

[135] *Qualcomm Snapdragon Heterogeneous Compute SDK*. 2018 (cit. on p. 89).

[136] J. Ragan-Kelley, C. Barnes, A. Adams, et al. “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530 (cit. on p. 14).

[137] Oliver Reiche, M. Akif Ozkan, Richard Membarth, Jurgen Teich, and Frank Hannig. “Generating FPGA-based image processing accelerators with Hipacc: (Invited paper)”. In: *ICCAD* (2017) (cit. on pp. 19, 64).

[138] James Reinders, Ben Ashbaugh, Alexey Bader, et al. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL James*. Apress open, 2020, pp. 1–2 (cit. on pp. 11, 13–15, 23, 57, 90).

[139] Santhosh Kumar Rethinagiri, Oscar Palomar, Javier Arias Moreno, Osman Unsal, and Adrian Cristal. “Trigeneous Platforms for Energy Efficient Computing of HPC Applications”. In: *International Conference on High Performance Computing Trigeneous*. IEEE, 2015 (cit. on pp. 6, 15, 89).

[140] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, et al. “Parallel multiprocessing and scheduling on the heterogeneous Xeon+FPGA platform”. In: *The Journal of Supercomputing* 76.6 (2020), pp. 4645–4665 (cit. on p. 90).

[141] Jose Carlos Romero, Angeles Navarro, Antonio Vilches, et al. “Efficient heterogeneous matrix profile on a CPU + High Performance FPGA with integrated HBM”. In: *Future Generation Computer Systems* 125 (2021), pp. 10–23 (cit. on pp. 6, 15).

[142]A. Sadek, A. Muddukrishna, L. Kalms, et al. “Supporting utilities for heterogeneous embedded image processing platforms (sthem)): An overview”. In: *International Symposium on Applied Reconfigurable Computing (ARC)*. Springer, May 2018, pp. 737–749 (cit. on pp. 65, 68).

[143]A. Sanaullah and M. C. Herbordt. “Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–8 (cit. on p. 18).

[144]B. C. Schafer and Z. Wang. “High-Level Synthesis Design Space Exploration: Past, Present, and Future”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (2020) (cit. on pp. 34, 44).

[145]Xilinx SDACCEL. *OpenCL Devices and FPGAs*. 2021 (cit. on p. 19).

[146]*SDSoC Environment User Guide* (cit. on p. 93).

[147]Hércules Cardoso da Silva, Flávia Pisani, and Edson Borin. “A Comparative Study of SYCL, OpenCL, and OpenMP”. In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 2016, pp. 61–66 (cit. on p. 6).

[148]John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science Engineering* 12.3 (2010), pp. 66–73 (cit. on p. 14).

[149]Makoto Sugawara, Shoichi Hirasawa, Kazuhiko Komatsu, Hiroyuki Takizawa, and Hiroaki Kobayashi. “A Comparison of Performance Tunabilities between OpenCL and OpenACC”. In: *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. 2013, pp. 147–152 (cit. on p. 14).

[150]S. Taheri, J. Heo, P. Behnam, et al. “Acceleration Framework for FPGA Implementation of OpenVX Graph Pipelines”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 227–227 (cit. on p. 64).

[151]Sajjad Taheri, Payman Behnam, Eli Bozorgzadeh, Alexander Veidenbaum, and Alexandru Nicolau. “AFFIX: Automatic acceleration framework for FPGA implementation of OpenVX vision algorithms”. In: *FPGA’19*. 2019, 252–261 (cit. on pp. 17, 64, 66, 80).

[152]TOP500. *GREEN500, The list*. 2021 (cit. on pp. 5, 12).

[153]TOP500. *TOP500, The list*. 2021 (cit. on pp. 5, 12).

[154]S. M. Trimberger. “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology”. In: *Proceedings of the IEEE* (2015) (cit. on p. 33).

[155]Tsoi, K. H. et al. “Axel: A Heterogeneous Cluster with FPGAs and GPUs”. In: *ACM/SIGDA FPGA*. Monterey, California, USA: ACM, 2010, pp. 115–124 (cit. on p. 89).

[156]Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. “Simultaneous Multithreading: Maximizing on-Chip Parallelism”. In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. ISCA ’95. S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, 392–403 (cit. on p. 12).

[157] Anshuman Verma, Ahmed E. Helal, Konstantinos Krommydas, and Wu-chun Feng. "Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs". In: *Computer Science Technical Reports* (2016) (cit. on p. 34).

[158] Vilches, A. et al. "Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips". In: *Procedia Computer Science, ICCS* 51 (2015), pp. 140 –149 (cit. on p. 89).

[159] Nils Voss, Tobias Becker, Simon Tilbury, et al. "Performance Portable FPGA Design". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, p. 324 (cit. on pp. 65, 66).

[160] Shuo Wang, Yun Liang, and Wei Zhang. "Poly: Efficient heterogeneous system and application management for interactive applications". In: *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019* (2019), pp. 199–210 (cit. on pp. 6, 15, 18).

[161] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa. "DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead". In: *HPCA*. 2014, pp. 380–391 (cit. on p. 46).

[162] Z. Wang, B. He, W. Zhang, and S. Jiang. "A performance analysis framework for optimizing OpenCL applications on FPGAs". In: *HPCA*. 2016, pp. 114–125 (cit. on pp. 17, 23, 33–36, 40, 45, 51, 59, 88).

[163] Zeke Wang, Bingsheng He, and Wei Zhang. "A study of data partitioning on OpenCL-based FPGAs". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–8 (cit. on p. 17).

[164] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. "Melia: A MapReduce Framework on FPGAs, OpenCL-based FPGAs". In: *IEEE Transactions on Parallel and Distributed Systems* 9219.c (2016), pp. 1–14 (cit. on p. 17).

[165] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms". In: *2014 21st International Conference on High Performance Computing (HiPC)*. 2014 (cit. on p. 6).

[166] Skyler Windh, Xiaoyin Ma, Robert J. Halstead, et al. "High-level language tools for reconfigurable computing". In: *Proceedings of the IEEE* 103.3 (2015), pp. 390–408 (cit. on p. 21).

[167] Mike Wissolik, Anthony Torza, and Brandon Day. *Virtex UltraScale+ HBM FPGA:A Revolutionary Increase in Memory Performance*. 2019 (cit. on pp. 123, 124).

[168] Roger Woods, John McAllister, Gaye Lightbody, and Ying Yi. *FPGA-based implementation of signal processing systems*. Wiley Online Library, 2017 (cit. on pp. 16, 82).

[169] Xilinx. *Xilinx OpenCV User Guide*. 2019 (cit. on p. 64).

[170] Xilinx Vivado. *Introduction to FPGA Design with Vivado High-Level Synthesis*. 2019 (cit. on p. 33).

[171] Xilinx Vivado. *Vivado Design Suite User Guide: High-Level Synthesis*. 2017 (cit. on p. 58).

[172]Wang Yong, Zhou Yongfa, Wang Scott, et al. “Developing Medical Ultrasound Imaging Application across GPU, FPGA, and CPU Using OneAPI”. In: *International Workshop on OpenCL*. IWOCL’21. Munich, Germany: Association for Computing Machinery, 2021 (cit. on p. 90).

[173]Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. “VirtCL: A Framework for OpenCL Device Abstraction and Management”. In: *SIGPLAN Not.* 50.8 (Jan. 2015), 161–172 (cit. on p. 14).

[174]Ramin Zabih and John Woodfill. “Non-parametric local transforms for computing visual correspondence”. In: *Computer Vision — ECCV ’94*. Springer, 1994, pp. 151–158 (cit. on p. 69).

[175]Mohamed Zahran. *Heterogeneous Computing: Hardware and Software Perspectives*. New York, NY, USA: Association for Computing Machinery, 2019 (cit. on p. 13).

[176]C. Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. “Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks”. In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016 (cit. on p. 35).

[177]Jialiang Zhang and Jing Li. “Improving the Performance of OpenCL-Based FPGA Accelerator for Convolutional Neural Network”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: Association for Computing Machinery, 2017, 25–34 (cit. on p. 17).

[178]J. Zhao., L. Feng, S. Sinha, et al. “COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2017), pp. 430–437 (cit. on p. 35).

[179]Jieru Zhao, Liang Feng, Sharad Sinha, et al. “Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.7 (2020) (cit. on p. 35).

[180]H. Zheng and Z. Zhu. “Power and Performance Trade-Offs in Contemporary DRAM System Designs for Multicore Processors”. In: *IEEE Transactions on Computers* 59.8 (2010), pp. 1033–1046 (cit. on p. 39).

[181]G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. “Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators”. In: *DAC*. 2016, pp. 1–6 (cit. on p. 35).

[182]Shijie Zhou and Viktor K. Prasanna. “Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform”. In: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2017, pp. 137–144 (cit. on p. 89).

[183]Hamid Reza Zohouri, Naoya Maruyamay, Aaron Smith, Satoshi Matsuoka, and Motohiko Matsuda. “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2016.November (2016), p. 35 (cit. on pp. 17, 18, 34, 40, 51, 58, 95).

[184] Hamid Reza Zohouri and Satoshi Matsuoka. “The Memory Controller Wall: Benchmarking the Intel FPGA SDK for OpenCL Memory Interface”. In: *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)* (2019), pp. 11–18 (cit. on pp. 18, 35).

[185] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. “Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’18. Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, 153–162 (cit. on pp. 18, 65).

8 Appendix: Memory aware co-execution in heterogeneous system with CPU and FPGA with HBM2 memory

The advances in FPGA accelerators with the inclusion of High Memory Bandwidth (HBM) in 3D-based FPGA chips extend the opportunities with its massive bandwidth that can fuel the large number of functional units of the FPGAs. The novelty devices introduce additional dimensions to the system, expanding the exploration space, which in turn increases the complexity required to achieve an efficient collaborative execution, fulfilling the performance and energy consumption constraints.

In this Appendix, we introduce an FPGA + HBM technology, in co-execution with a CPU, showing the main findings and challenges.

8.1 Introduction

For FPGAs, the memory bandwidth capabilities have increased slowly compared with the number of computational units, and even Ethernet interfaces have outpaced DDR4 memory technologies. For example, for video broadcasting the industry has increased the resolution and a significant gap is evident among the bandwidth application requirements and those provided by a DDR4 DIMM as Figure 8.1 shows [167].

In Chapter 4, we showed that many FPGA applications are memory-bound. The introduction of HBM2 memories is an alternative to reduce the bandwidth gap, increasing memory bandwidth with more DRAM banks and package integration.

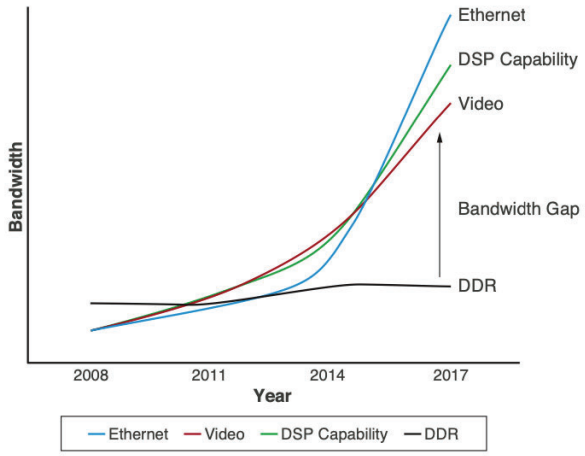


Figure 8.1: Relative memory bandwidth requirements [167].

The package integration can increase the power efficiency 3.8× compared with a DDR4 DIMM memory.

An FPGA with HBM2 memory can potentially exploit better the FPGA architecture, theoretically increasing the bandwidth, requiring more compute capabilities of the FPGA device, that is one of the limits detected in Chapter 6 to achieved a better energy efficiency balance when the work load is shared among different devices in a heterogeneous system.

In this appendix we explore the integration of the FPGA with an HBM2 memory in a heterogeneous system and discuss the main problems of this approach.

8.2 FPGA with HBM memory in a Heterogeneous system

The FPGA model with HBM2 support is the Stratix 10 MX. One of the major problems of this board is the early support offered by Intel manufacturer and the lack of support of the programming frameworks. The management of the HBM2 pseudo-channels is the programmer's responsibility, increasing the complexity of application development which should be aware of the memory resource assignment.

For this reason, we have introduced new features to EngineCL to allow a better explotation of the memory banks in Stratix 10 MX. The applications are designed in OpenCL using kernel replication to increase both the memory bandwidth and compute demand. Each variable in a kernel could be assigned to a different HBM2 pseudo-channel bank; no arbitration is required compared with a DDR4 with a single bank. Listing 8.1 shows an example of kernel replication on a kernel with three global variables in different HBM2 pseudo-channels. The kernel replication limit is the amount of FPGA resources and a maximum of 32 HBM2 pseudo-channels. In host side, in EngineCL framework, we created a command queue per replicated kernel to allow them work in parallel. In the load balancers policies, each FPGA chunk is divided among kernel replicas.

```
1  // device code
2  void kernel0(
3    __global __attribute((buffer_location( "HBM0")) uint A,
4    __global __attribute((buffer_location( "HBM1")) uint B,
5    __global __attribute((buffer_location( "HBM2")) uint C){
6         //compute
7   }
8   ...
9  void kerneln(
10   __global __attribute((buffer_location( "HBM28")) uint A,
11   __global __attribute((buffer_location( "HBM29")) uint B,
12   __global __attribute((buffer_location( "HBM30")) uint C){
13        //compute
14 }
15
16 //Host with EngineCL framework
17 // replicated kernels in main host program
18 FPGA.setKernel(binary_file,"kernel0");
19 ...
20 FPGA.setKernel(binary_file,"kerneln");
```

Listing 8.1: OpenCL kernel replication for FPGA + HBM2 memory, in Line 1 is the device code and in the Line 16 is the host code with EngineCL framework

To evaluate the performance we used the Host 3 in Table 3.2 with a Intel Xeon Bronce 3204 and a Stratix 10 MX, the OpenCL driver version in this case is only compatible with Intel processors. The first results, using a vector add kernel to tune the framework show a communication bandwidth problem in the OpenCL driver for CPU device with only 1350 MB/s, 6× worst than FPGA communication through a PCIe. This performance difference among devices are compensated applying kernel

replication in CPU device. Figure 8.2 shows the result for the kernel replication in CPU and FPGA devices, in CPU the performances increases up 1.8×, where the number of kernel copies are equals to the number of cores. Instead in FPGA the the speed-up gains is minimal with 16% since the vector-add kernel is bound by PCIe communication.

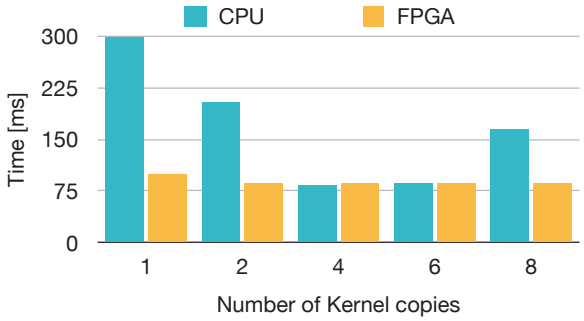


Figure 8.2: Performance impact of Kernel replication in CPU and FPGA devices for the vector add kernel application.

The performance differences with kernel replication increase the complexity in the load balancing algorithms, and evidence the necessity of awareness load balancers. Exploring the kernel replication with a static strategy in Figure 8.3 shows that the best performance is achieved with a 92% of the load in the FPGA, which is the best device. In the overall, the maximum speed-up of the heterogeneous execution is 22%. Although with kernel replication the CPU and FPGA have a similar performance, the interaction between device drivers limits the CPU, losing the kernel replication gains. In terms of energy efficiency, FPGA is the most efficient closely followed by the best heterogeneous combination with one kernel FPGA replication. Please note that this results could be replicated on memory bound kernels because PCIe communication dominates the FPGA execution time.

In opposition to vector add benchmark, we evaluated a irregular Mandelbrot application which is compute bound. In this case, the FPGA kernel is no replicated since just one kernel replica exhausts DSPs available in the entire FPGA. In CPU, even leveraging the available cores, the performance is 15 × worst than FPGA. The heterogeneous execution with a 95% of workload for FPGA achieves a 5 % of speed-up compared with the FPGA alone. In terms of energy, FPGA alone is the best configuration, and the best heterogeneous execution is 32% worsse with a measurement of 230 J.

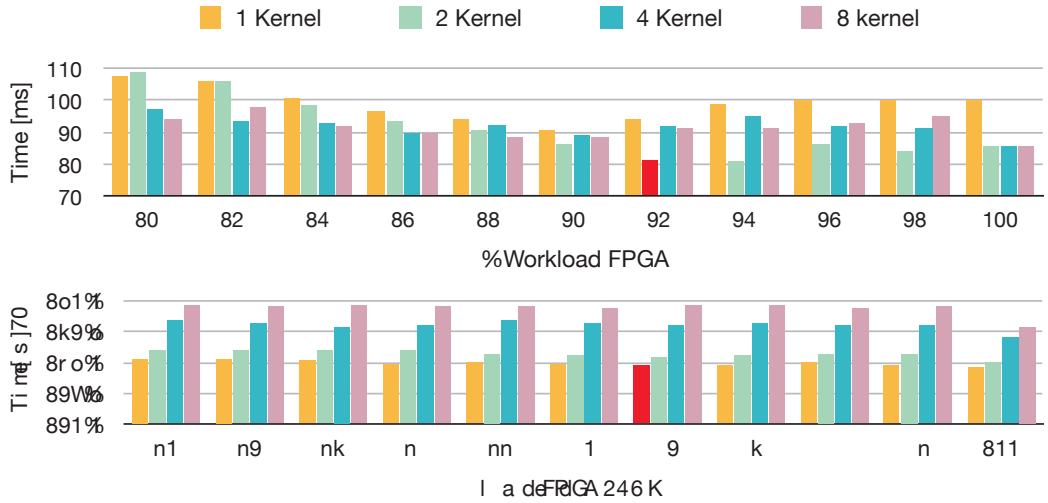


Figure 8.3: Static load balancing exploration for a CPU and FPGA+HBM2 varying the workload distribution and the number of kernel copies in FPGA. The bar in red shows the best heterogeneous execution in time and energy.

In conclusion, the co-execution in heterogeneous systems requires a performance balance between the devices, otherwise distributing workload among them is useless. Although the advances in specialized processors could achieve more performance and energy efficiency, it continuously requires expensive time design and programmers with advances skills from FPGA hardware to host drivers to achieve a high performance heterogeneous system. In the FPGA hardware, the kernel replication reduce kernel performance since the frequency is reduced with more HBM2 banks and newer interconnection networks are necessary to fully exploit the HBM2 bandwidth.

List of Figures

2.1 Intel Stratix 10 internal block diagram architecture. 16
2.2 FPGA with an HBM memory as Intel Stratix 10 MX with 32 pseudo-channels and 8 GB of memory across top and bottom interfaces. 18
2.3 HLS synthesis tool flow for Intel FPGAs. The lower part compares the timing effort between HLS and traditional RTL design methodologies. . 20
2.4 Main elements of OpenCL BSP for FPGAs. ❶ and ❷ represent the BSP, and ❸ the kernel logic. 22

3.1 Heterogeneous system with an FPGA board and a CPU. 25
3.2 FPGA power measurement methods. 30
3.3 OpenCL compilation flow for Intel FPGA board. The box in yellow correspond to generated files, in blue are FPGA execution stages and the rest are the aoc compilation commands in each design stage: Emulation, Intermediate compilation and Full deployment. 31
3.4 OpenCL Flow for a heterogeneous system with a host CPU, GPU and FPGA devices. The host program in this thesis uses two host programming frameworks: OpenCL and EngineCL with less code lines. 32

4.1 Memory model proposal(in blue) in the optimization process of FPGA kernels using HLS. 34
4.2 FPGA block units for Intel OpenCL SDK with a) DDR4 and b) HBM memory. 37
4.3 Simplified model of a read operation in a single DRAM bank with an Burst-Coalesced Aligned LSU. The parameter names in blue are used in the model in Table 4.2. 39
4.4 Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced aligned LSU varying the vectorization factor (v) and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. The bars with dots and stripes represent T_{ideal} and T_{ovh}, respectively. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. 52

4.5 Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{meas} for $\delta = 1$. The experiment varies δ with $\#lsu = 3$ and $v = 16$ for burst-coalesced aligned LSUs in two types of external memory: a) DDR4 1866 and b) HBM2, adjusting for special cases. 53
4.6 Measured (T_{Meas}) and Estimated ($T_{ideal} + T_{ovh}$) time for the burst-coalesced non-aligned LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a)DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. 54
4.7 Measured (T_{meas}) and estimated (T_{est}) time are normalized to T_{Meas} in $\delta = 1$. The experiment varies δ with fixed values of $\#lsu = 3$ and $v = 16$ for burst-coalesced non-aligned LSU in two types of external memory: a) DDR4 1866 and b) HBM2. 54
4.8 Measured (T_{meas}) and estimated ($T_{ideal} + T_{ovh}$) time for burst-coalesced write-acknowledge LSU varying the vectorization factor v and global access ($\#ga$) in two types of external memory: a) DDR4 1866 and b) HBM2. Kernels with non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. 55
4.9 Measured (T_{meas}) and estimated ($T_{ideal}+T_{ovh}$) time for Atomic-pipelined LSU varying the vectorization factor (v) and global access ($\#ga$) in a DDR4 1866 memory. Non-saturated memory bandwidth (NS) are detected (empty bars) and not estimated. The time axis is in seconds and logarithmic. 56
4.10 Frequencies Histogram for 18 kernel applications; the red dotted line shows the required minimum frequency for maximizing bw_dram. . . . 57
4.11 Estimation error of the execution time and frequency error from pre-synthesis report and after place-and-route in kernels that are limited by a frequency under $400MHz$ after synthesis in HBM cases. 57

5.1 Programming flow alternatives for OpenVX using HLS for FPGA devices. The yellow boxes show OpenVX functions implemented as OpenCL functions and the green ones the OpenVX functions implemented as kernels. The bottom boxes show host command queues, Qn, that manage the kernels. 66
5.2 Image functions categories implemented in HiFlipVX. 68

5.3 Resource comparison between Intel and Xilinx [83] FPGA at 100MHz and vectorization equal to 1, for 6 sample OpenVX functions. 71
5.4 Latency and Initiation Interval for interface optimizations on edges in a 3x3 filter function (lower is better). 72
5.5 DRAM memory interconnection to a HiFlipVX graph. 74
5.6 Efficiency (Frames Per Second) for canny edge detector with a HD image varying coalescing to read DRAM memory (LSU width), higher is better. 74
5.7 HiFlipVX programming and compilation flow for Xilinx and Intel FPGAs. 75
5.8 Impact of node scalability on a) execution time; b) frequency; c) resource utilization; and d) power consumption for the Multi-Gaussian synthetic benchmark using the Stratix 10 GX. 76
5.9 OpenVX application graph diagrams. a) Canny edge detector, b) Census transform, c)Autocontrast image, d) Skin tone detection. 77
5.10 Latency of canny edge for HiFlipVX and AFFIX using an Stratix 10 GX FPGA. 80
5.11 Code comparison between a reduced version of OpenVX, AFFIX, and HiFlipVX. OpenVX definitions and FPGA optimization parameters are marked in orange and grey, respectively. 81
5.12 Resource usage per logic unit relative the total units on Stratix 10 GX and Stratix 10 MX for AFFIX and HiFlipVX implementations. 83
5.13 HiflipVX tiling with HBM2 memory bandwidth. A) Measured time and estimated ideal time. B) Memory bandwidth efficiency is the relation between measured and maximum expected memory bandwidth; and frequency efficiency is the relation between kernel frequency and minimum required frequency (400MHz). C)Resource performance is the relation between total memory bandwidth vs. logic resources usage (ALM). . . . 84

6.1 EngineCL architecture: tiers, modules and applied patterns. The highlighted modules are extended to support FPGAs. 91
6.2 Three different alternatives of schedulers to alleviated the differences among applications and devices [25]. 92
6.3 Overview of command queue overlapping with the two command queues and the two buffer sets (A and B). 94
6.4 overlapping time performance varying the number of work load partitions(chunks) of the problem in the case of communication-bound and computation-bound problems in an FPGA. 94

6.5 Normalized execution time to the worst for the Static scheduler. Work proportions go up to 50% for two devices, and the third device performs the remainder work. 96
6.6 Dynamic scheduler throughput (GB/s) of CPU, GPU and FPGA with chunk size variation. 97
6.7 Overall median performance of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. Times are normalized to CPU in every benchmark. 99
6.8 Overall energy of cooperative execution CPU(C)+GPU(G)+FPGA(F) with Static(St), Dynamic (Dy) and HGuided (HG) load balancers for each benchmark. Energy are normalized to the CPU device in every benchmark. 101
6.9 Overall normalized energy-delay product of cooperative execution CPU+GPU+FPGA with three load balancers for each benchmark. The lower the better, and Matrix Multiplication results have been zoomed for clarity. 101

8.1 Relative memory bandwidth requirements [167]. 124
8.2 Performance impact of Kernel replication in CPU and FPGA devices for the vector add kernel application. 126
8.3 Static load balancing exploration for a CPU and FPGA+HBM2 varying the workload distribution and the number of kernel copies in FPGA. The bar in red shows the best heterogeneous execution in time and energy. . 127

List of Tables

2.1 CPU, GPU and FPGA characteristics as accelerators in a heterogeneous system, classifying in three categories: best, worst, and intermediate. . . 15
2.2 Intel FPGAs used in board accelerators. 17

3.1 Device combinations used as heterogeneous system in this thesis. The host, FPGA and GPU capabilities are described in Tables 3.2, 3.4, and 3.3. 26
3.2 Main characteristics of the three host systems. 27
3.3 Characteristic of NVIDIA GeForce GTX TITAN X GPU board. 28
3.4 Characteristic of Terasic DE5-net, Intel Stratix 10 GX, and Intel Stratix 10 MX (early version) board development kits. 29

4.1 LSU types and their modifiers in global memory interconnect. The code snippets are from Intel FPGA SDK [65]. 38
4.2 Description of model parameters. The *param* label for the Verilog source refers to a variable name in a Verilog instance. 42
4.3 Fixed variable value to evaluate the LSU model on Stratix 10 GX and Stratix 10 MX with a DDR4 1866 and HBM2 memory respectively. All variables are defined in Table 4.2. 50
4.4 Kernel applications and estimated time in two memories: DDR4 1866 and HBM2.GMI- global memory interconnect BCA- burst-coalesced aligned LSU. BCNA- burst-coalesced non-aligned LSU. ACK- burst-coalesced write-acknowledge LSU. M- Measured. E- Estimated. 58
4.5 Execution time estimated error; μb, BCA, BCNA, and ACK refer to microbenchmark, burst-coalesced aligned, burst-coalesced non-aligned, and burst-coalesced write-acknowledge LSUs, respectively. 60

5.1 Programming flow alternatives to implement the OpenVX standard. . . . 67
5.2 Estimated resource usage for each OpenVX function in Canny edge graph using HiFlipVX with a 4K image and vectorization factor of 8 on a Stratix 10 GX. 78

5.3 Estimated resource usage for each OpenVX function in Autocontrast graph using HiFlipVX, with a HD image and vectorization factor of 1. 78
5.4 Estimated resource usage for each function in Census transform using HiFlipVX, with a 4K Image and vectorization factor of 8. 79
5.5 Estimated resource usage for each OpenVX function in Skin tone graph using HiFlipVX, with an HD image and vectorization factor of 1. 79
5.6 HiFlipVX results on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image. 81
5.7 Comparison between HiFlipVX and AFFIX on a Intel Stratix 10 GX and Intel Stratix 10 MX using a 4K image. 82

6.1 Work-items and FPGA characteristics: Clock frequency(CF), initialization interval (II), and FPGA kernel resources: adaptive logic module (AL), logic registers (LR), memory blocks (MB), and DSP. 95
6.2 Median performance improvement (PI) relative to the fastest single device execution, imbalance (IM), and average number of chunks (#C) for Static, Dynamic, and HGuided policies per benchmark. 98
6.3 Average power (W) for single device configurations. I, P, DR, HR represents Idle, Programmed, Device Running, and Host Running power, respectively. 100

Listings

4.1 OpenCL Code for access patterns in Table 4.1 37
4.2 Atomic-pipelined add prototype function 48
4.3 OpenCL template microbenchmark to vary global access number. 50

5.1 vx_image for virtual image implementation with Intel streams support . . 72
5.2 vx_image for image implementation with Intel DRAM support 73

8.1 OpenCL kernel replication for FPGA + HBM2 memory, in Line 1 is the device code and in the Line 16 is the host code with EngineCL framework 125