UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de São José do Rio Preto

Gustavo Leite

Performance Evaluation of Code Optimizations in FPGA Accelerators

São José do Rio Preto

2019

Gustavo Leite

Performance Evaluation of Code Optimizations in FPGA Accelerators

Orientador: Prof. Dr. Alexandro José Baldassin

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiamento: CAPES e FAPESP (Proc.: 2017/09065-9 e 2018/08116-1)

São José do Rio Preto

2019

L533p

Leite, Gustavo
Performance Evaluation of Code Optimizations in FPGA Accelerators / Gustavo Leite. -- São José do Rio Preto, 2019
66 p. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto
Orientador: Alexandro José Baldassin

1. FPGA. 2. High-Performance Computing. 3. Code Optimization. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Gustavo Leite

Performance Evaluation of Code Optimizations in FPGA Accelerators

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Financiamento: CAPES e FAPESP (Proc.: 2017/09065-9 e 2018/08116-1)

Comissão Examinadora

- Prof. Dr. Alexandro José Baldassin (Orientador)
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Orlando de Andrade Figueiredo
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Emilio de Camargo Francesquini
  Centro de Matemática, Computação e Cognição
  Universidade Federal do ABC – UFABC

Rio Claro

28 de agosto de 2019

To my family and my professors

Acknowledgements

Thanks to Professor Alexandro Baldassin for his guidance, inspiration, immeasurable patience and all the opportunities he created for me.

Thanks to my family Sandra Regina Vieira, Sergio Luiz Leite, Bruna Karina Leite, Alexandre Locatelli Bertoline Filho, Amanda Tofolo, Carla Cristiane Garutti and Daniel Basilio Dias for the unconditional support.

Thanks to all the professors from the Department of Statistics, Applied Mathematics and Computer Science (DEMAC) for helping me grow not only intellectually but also as a person. Thanks to Professor José Nelson Amaral for the contributions and guidance during my visit to the Computing Science Department in the University of Alberta. Thanks to Professor Guido Araújo for the insightful comments and contributions. Thanks to professors Daniel Carlos Guimarães Pedronette, Ivan Rizzo Guilherme and Orlando de Andrade Figueiredo for inspiration.

Thanks to the defense committee for the contributions.

Thanks to all my friends from Laboratório de Inteligência Artificial Aplicada ao Petróleo (LIAAP), University of Alberta Systems Laboratory (UASYS) and also my friends from Araras-SP and Rio Claro-SP for support. Thanks to the Department of Statistics, Applied Mathematics and Computer Science (DEMAC), the São Paulo State University (UNESP) and the University of Alberta.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This study was also funded by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grants 2017/09065-9 and 2018/08116-1.

Resumo

Com o crescimento contínuo do consumo de energia em microprocessadores, cientistas e engenheiros da computação redirecionaram atenção a arquiteturas heterogêneas, onde dispositivos de classes diferentes são usados para acelerar a computação. Dentre eles, existem as FPGAs (*Field-Programmable Gate Arrays*) cujo hardware pode ser reconfigurado após sua fabricação. Esta classe de dispositivos demonstra desempenho comparável aos processadores convencionais enquanto consomem apenas uma fração de energia. O uso de FPGAs vem se proliferando nos últimos anos e a perspectiva é que o nível de adoção continue a crescer. No entanto, programar FPGAs e aprimorar os programas para obter maior desempenho continua uma tarefa não trivial. Este trabalho apresenta uma compilação das principais transformações de código para otimização de programas direcionados à FPGAs. Neste trabalho também é avaliado o desempenho de programas executando em FPGAs. Mais especificamente, um subconjunto das transformações de código são aplicadas em um kernel OpenCL e os tempos de execução são medidos em um dispositivo da Intel®. Os resultados mostram que, sem a aplicação das transformações, o desempenho dos dispositivos é abaixo do que é observado quando as transformações são de fato aplicadas.

Palavras-Chave: FPGA, computação de alto desempenho, otimização de código.

Abstract

With the ever increasing power wall in microprocessor design, scientists and engineers shifted their attention to heterogeneous architectures, wherein several classes of devices are used for different kinds of computation. Among them are FPGAs whose hardware can be reconfigured after manufacturing. These devices offer comparable performance to CPUs while consuming only a fraction of energy. In fact, the use of FPGAs have been proliferating in recent years and should continue to do so considering the amount of attention these devices are receiving. Still, programmability and performance engineering in FPGAs remain hard. This work presents a compilation of the most prominent code transformations for optimizing code aimed at FPGAs. In this work we also evaluate the performance of programs running on FPGAs. More specifically, we apply a subset of the code transformations to an OpenCL kernel and measure the execution time on a Intel® FPGA. We show that, without applying these transformations before execution, poor performance is observed and the devices are underutilized.

Keywords—FPGA, high-performance computing, code optimization.

List of Figures

List of Tables

List of abbreviations and acronyms

AI	Artificial Intelligence
ALM	Adaptive Logic Module
ALUT	Arithmetic and Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
CB	Connect Box
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
CPU	Central Processing Unit
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FF	Flip Flip
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GPU	Graphic Processing Unit
HARP	Hardware Accelerator Research Program
HDL	Hardware Description Language
HLS	High-Level Synthesis
I/O	Input/Output
IC	Integrated Circuit
IR	Intermediate Representation
LLVM	Low Level Virtual Machine
LUT	Lookup Table
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi Processing
PE	Programmable Element

PLD	Programmable Logic Devices
PL	Programmable Logic
PROM	Programmable Read-Only Memory
PS	Programmable Switch
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
TPC	Thread Pool Composer
TPU	Tensor Processing Unit
VHDL	VHSIC Hardware Description Language

Contents

1 Introduction

With Moore's Law (7) and Dennard Scaling (8) approaching their end, the high-performance market shifted its focus towards heterogeneous computing systems, where each device is specialized to accelerate domain-specific applications (9). Among the many classes of devices, the most noteworthy are graphical processing units (GPUs), tensor processing units (TPUs), and field-programmable gate arrays (FPGAs) (10). In particular, FPGAs have existed since the mid-1980s and have been used to create logic circuits for embedded systems. More recently, FPGAs also have been commercialized as accelerators, integrating multicore ARM-processors, DRAM, digital signal processors (DSPs) and storage onto a single board usually referred to as "System on Chip" (SoC). Due to their reconfigurable nature, FPGAs offer comparable performance compared to CPUs and usually higher performance per watt compared to GPUs.

Power efficiency is listed in various technical reports as the top hardware challenge in the road to achieve exascale computing (11, 12, 13). According to a report published by U.S. Department of Energy (11), the cost of ownership of a petascale computer sits between $5-10M annually. For exascale computers, however, this cost can reach $2.5B per year with current technology. With that in mind, cloud providers and other players in the HPC market started investigating reconfigurable hardware as a way to make exascale computing feasible. This will be critical going forward considering that electricity and cooling equipments costs sum up to 26% of the total cost of ownership of a data center's infrastructure (14). Amazon Web Services, for instance, one of the leading cloud providers, recently upgraded their servers and now offer instances with up to 8 FPGAs (15, 16). Microsoft Research is using FPGAs for running deep neural networks for real time AI (17).

1.1 Motivation

Despite the clear benefits, FPGA devices are still not as widespread as GPUs, for instance. This is due to three main reasons: (i) the long time to perform hardware synthesis; (ii) the lack of a high-level programming model; and (iii) lack of portability. The process of translating a circuit written in hardware description language into a bitstream—a hardware configuration file—is called hardware synthesis. Synthesizing hardware can last from minutes to a few days (18). This characteristic limits the ability to write incremental code and fast prototyping. Although the programmability of these devices has improved since their first appearance, current software development kits adopted OpenCL (2). For novices and non-experts, even that can be fairly low-level. Before that, reconfigurable hardware was mainly programmed using hardware description languages (VHDL, Verilog), which was even worse since these languages work in the register transfer level (RTL), a paradigm most software developers are unfamiliar with. Lastly, porting a design from one device to another requires a considerable amount of work to be redone. The challenge is aggravated when porting across devices from different vendors.

Performance tuning is not trivial and usually requires several iterations. With other classes of devices, it is straightforward to experiment with different implementations. With FPGAs, however, the long synthesis time severely delays the design cycle. The challenge is scaled when moved to a cluster—now programmers needs to worry about data transfer and workload balancing among the nodes. With this scenario in mind, it is important that we better understand how to tune applications and evaluate trade-offs.

1.2 Objectives

In this work, we provide a thorough analysis of existing code optimizations aimed at OpenCL applications for FPGAs. We not only show how to apply them, but also explain the architectural features of FPGAs that make these optimizations effective in the first place. We provide a discussion related to these optimizations regarding portability to other devices and the difficulty of tuning its parameters. We also present a research on related works regarding code optimization for FPGA and perform an experimental analysis aimed at evaluating the performance of individual code optimizations. For this matter, we select a subset of optimizations, apply them individually to a naive implementation of a matrix multiplication and measure the execution time. In particular, this work makes the following contributions:

1. It compiles and explains a collection of code optimizations found in the literature that can benefit the performance of OpenCL applications running on Intel® FPGAs;
2. It analyzes and summarizes the current state of research about FPGAs for high-performance computing;
3. It presents a performance evaluation of a subset of these transformations on a real application.

We stress the importance of this work in upcoming years because adoption of FPGA devices is rising but not enough programmers are capable of working with such devices. Unlike GPUs which are widespread and have a broad set of open-source tools for profiling and characterization, FPGA development is still in its infancy and the tools are mainly proprietary. There is also the difficulty of running experiments due to the long compilation time, which poses a barrier to thorough characterization of performance. Therefore, this work serves as a first step towards understanding the performance of

code optimizations for FPGAs with the aim of automating them through a compiler infrastructure.

1.3 Text Organization

This document is organized as follows: we present the necessary background about FPGAs and OpenCL in Chapter 2; on Chapter 3 we present code optimizations found in the literature and also the related work that evaluate some of these optimizations; on Chapter 4 we present and discuss experimental data gathered by writing benchmarking applications; finally, on Chapter 5 we present our final considerations and plans for future work.

2 Background

This chapter provides the background information related to this research. In Section 2.1 we present the basic architecture of FPGAs and how they are programmed. Section 2.2 describes the OpenCL architecture, including its four models.

2.1 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays are silicon devices that can be reconfigured to emulate any combinational and sequential logic the user desires. Its ability to be reprogrammed after manufacturing grants its name “field-programmable”. The FPGA chip is composed of hundreds of thousands of logic elements (LEs) organized as a mesh—hence the name “gate array”—and a programmable interconnect that links these elements.

FPGA devices date back to the 1980s when the two major manufacturers, Altera (acquired by Intel in 2015 (19)) and Xilinx, were founded. Before that, programmable hardware existed in the forms of Programmable Read-Only Memory (PROM) and Programmable Logic Devices (PLD). Besides using different technology, this kind of reconfigurable devices had a smaller number of logic elements and the interconnect was less flexible (20).

We can imagine a spectrum of hardware devices where, on one extreme there are general-purpose devices, like CPUs, that can be programmed via software. On the other extreme are Application-Specific Integrated Circuits (ASICs) which are hard-wired low power designs built for a specific purpose. FPGAs sit in the middle. They are still general-purpose because of their reconfigurable traits while maintaining comparable

power efficiency in relation to ASICs (18).

The basic building blocks of FPGAs are logic elements. These elements are usually composed of LUTs (lookup tables) for combinational logic, registers for sequential logic and adders for arithmetic. Figure 1 presents the organization of the logic modules found in the Intel® Stratix® V chip, called "adaptive logic modules" (ALMs). This model features an 8-way adaptive LUT, two full adders and four general-purpose registers.

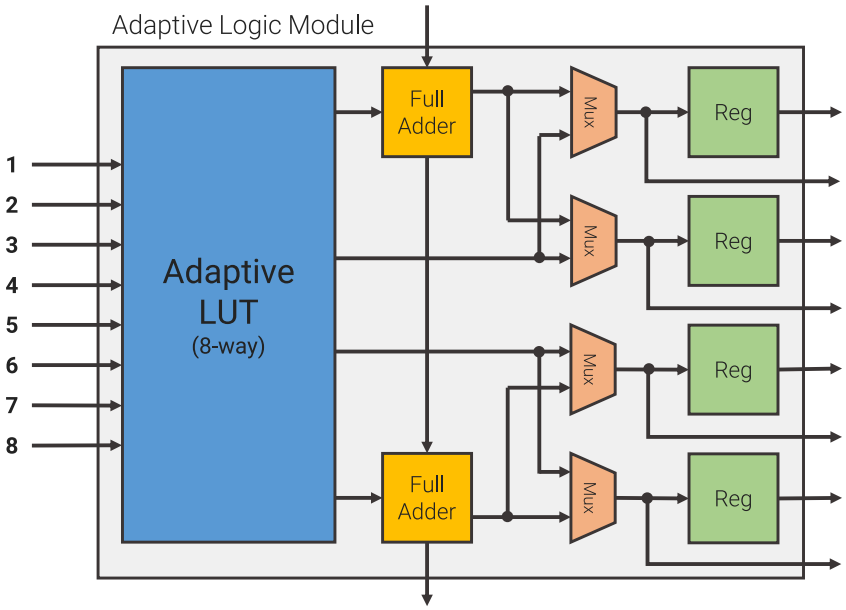


Figure 1 – Block diagram of an adaptive logic module from an Intel® Stratix® V device. This particular model exhibits a 8-way adaptive LUT, two full adders and four general-purpose registers (Adapted from: (1)).

Figure 2 shows a 3-way lookup table. LUTs are generally implemented using multiplexers. The inputs to the LUT are connected to the control lines of the multiplexer, while the inputs of the multiplexers themselves are hard-wired (the case with ASICs) or stored in D-latches (the case with FPGAs). In simpler terms, the LUT's memory stores the output column of the desired function's truth table and the inputs are used as indices to query this table, hence the name "lookup table". As with the Intel® Stratix® V example

Truth Table

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

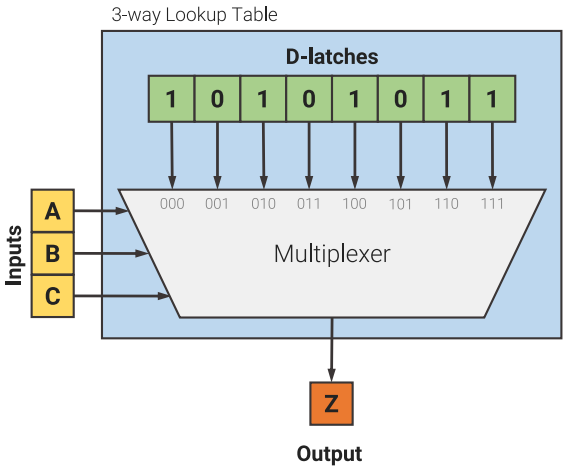


Figure 2 – Boolean function $Z = A \cdot B + \bar{C}$ implemented using a 3-way lookup table. The inputs to the combinational logic A, B, C are connected to the control lines of a multiplexer which selects the corresponding bit of the "table" stored in the D-latches.

given above, the LUT is said to be adaptive, meaning that it is possible to configure one boolean function that takes 8 inputs, or two booleans functions that take 4 inputs, or still, other configurations.

On a larger scale, logic elements are organized into a mesh and linked by a programmable interconnect network. Observe Figure 3 for instance. Around the mesh are the I/O (yellow) blocks that connect logic elements to the rest of the board (memory, main processor, etc). Inside, the green blocks represent the programmable switches that can be reconfigured to create any path in the mesh. Finally, orange blocks are instances of logic elements as described above. The programmable switches can be reconfigured in any way as to link I/O blocks and LEs, forming a cascade organization that resembles a pipeline. The number of LEs in modern devices range from hundreds of thousands to a few millions. This can be a limiting factor if the circuit does not fit the FPGA or, as it is common to say, the FPGA does not have enough area to hold the circuit.

Note that, if a particular function cannot be configured using a single ALM,

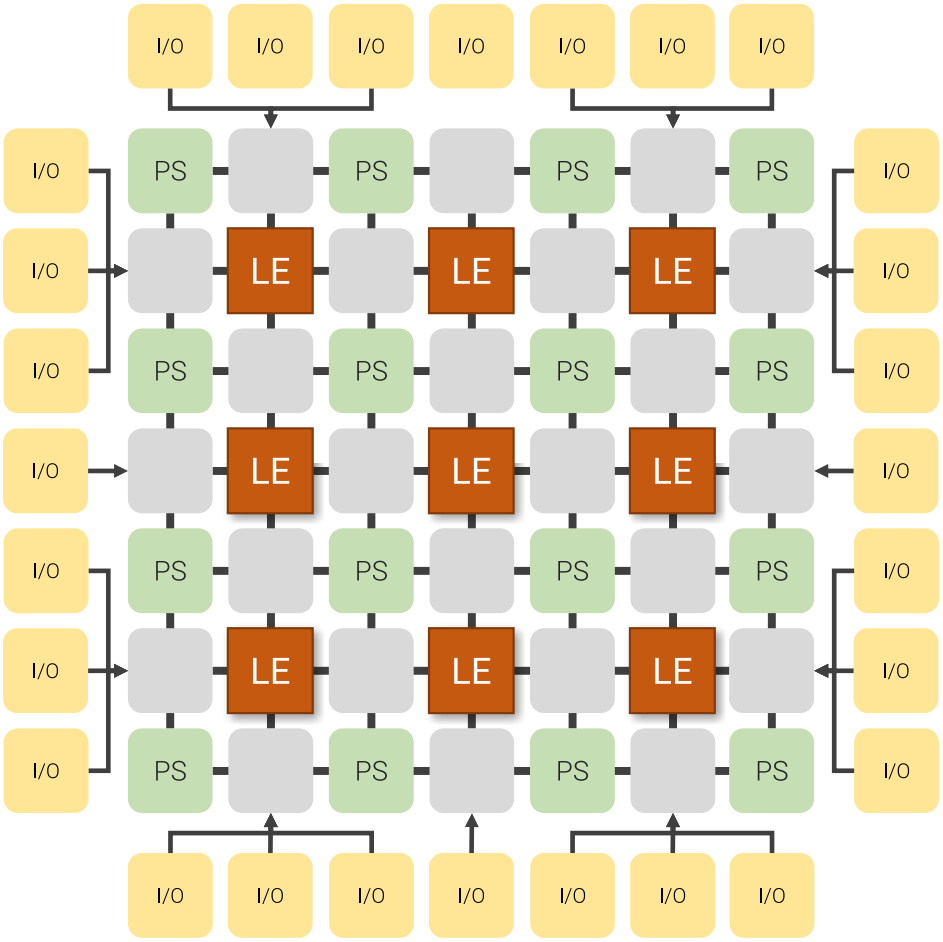


Figure 3 – Simplified organization of an FPGA chip: on the edge are the input/output blocks (IO); the logic elements (LE) do the actual computing, as shown in Figure 1; there are also programmable switches (PS) and connect boxes (CB) for routing the I/O blocks to the logic elements.

multiple of these can be connected using the mesh to form more complex circuits. In real devices, there are still other blocks types in the mesh, *e.g.* block RAM, DSPs, besides logic elements and programmable switches. These were omitted for simplicity.

For many years, the use of FPGAs has been restricted to engineers with hardware expertise. This is attributed to the lack of high-level abstractions to program such devices. Formerly, engineers would design the circuit in a hardware description language (HDL)

and obtain the concrete representation of the circuit through a process called synthesis. These representations are commonly referred to as *bitstreams*, and they inform how bits in the logic elements and programmable switches should be set to compose the circuit.

As stated previously, the synthesis step takes the hardware description and produces a concrete representation of the circuit for a particular device. For that matter, gates, multiplexers and registers are mapped from the description to specific logic elements in the FPGA mesh, a process called *placement*. Then, the placed elements are connected to one another by setting the programmable switches, a process called *routing* (21). Synthesis usually takes a long time—from hours to days, depending on the size of the circuit—because it is an NP-hard problem (18, 22, 23). The large number of possible placements and possible routings lead to a combinatorial explosion. Ideally, the circuit should occupy the minimum amount of area and the routes should be as short as possible in order to minimize wire delay. Because searching the entire space is unfeasible, approximation algorithms have been employed, such as: simulated annealing (24) when IC design was at the order of thousands of gates; recursive hypergraph partitioning (25) was used when million-scale design started to be developed; currently, the state-of-the-art is the ePlace algorithm (26).

Returning to the problem of designing circuits, the absence of high-level abstractions has kept FPGAs from being widespread. HDLs are very different from what software developers are used to because they work on the Register Transfer Level (RTL). Instead of variables, functions and control flow, RTL languages provide primitives such as registers, gates and multiplexers. The semantic gap is simply too wide for most software engineers. Observing the growing interest in reconfigurable hardware for high-performance computing, manufacturers developed tools capable of doing high-level synthesis (27, 28), *i.e.*, translate C/C++ code into HDL. Eventually the market settled on software development kits (SDKs) based on OpenCL (presented in Section 2.2) to ease the task of programming such devices.

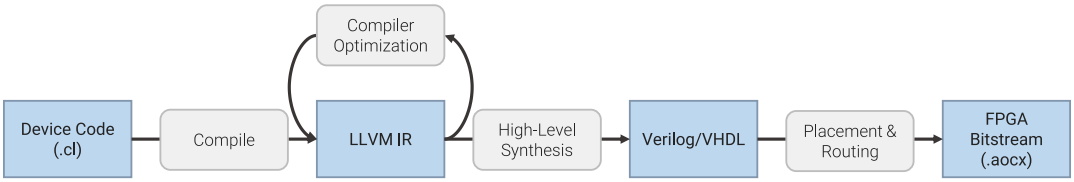


Figure 4 – Compilation flow of OpenCL code for FPGAs.

With modern SDKs, programmers implement the algorithm as OpenCL kernels as one would normally do for other types of accelerators. The only exception being that, in the case of FPGAs, programmers can only run pre-compiled binaries because doing just-in-time compiling is impractical. The vendor tool chain gets the OpenCL source and translates it to HDL via high-level synthesis. The HDL description, in turn, goes through the long process of synthesis. At the end of this process, a bitstream file is produced that can be used to program the circuit onto the device. This process is depicted in Figure 4.

An important caveat to keep in mind is that, although functionally portable, executing code targeted for GPUs on FPGAs is unlikely to yield any benefits both in terms of speedup and power efficiency. The code is said to lack performance portability (3). This is due to architectural differences: while GPUs are massively parallel devices with thousands of computing cores suited for data-parallel computation, FPGAs are more suitable for executing pipelined, task-parallel computations. Even though it is possible to replicate the data path in reconfigurable accelerators—thus enabling data-parallelism—depending on the size of the circuit, the area available in the FPGA may pose a restriction. If the area is sufficient, GPUs are still expected to perform better while processing data-parallel applications because of the sheer amount of floating point operations these accelerators are capable of executing every second.

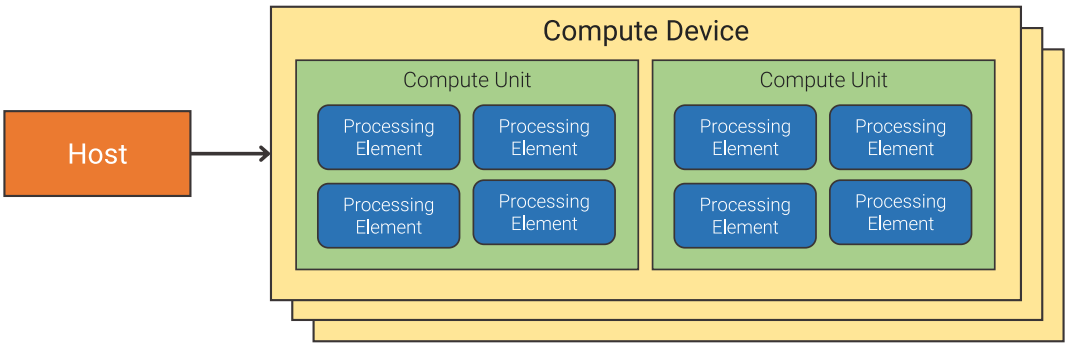


Figure 5 – OpenCL Platform Model (Adapted from: (2)).

2.2 OpenCL Architecture

OpenCL (*Open Computing Language*) is a framework for general purpose parallel programming standardized and maintained by Khronos Group (2). The first version of OpenCL was released by Apple in 2009. The standard defines a portable programming abstraction that is compatible between devices of different classes like CPUs, GPUs, FPGAs and more. The framework is divided in two parts: (i) a low-level API for managing and coordinating parallel and execution; and (ii) a C-based programming language that is interoperable and consistent across different classes of devices.

Conceptually, this framework can be better explained using the platform, execution, memory and programming models discussed in the following sections.

2.2.1 Platform Model

The OpenCL platform model describes how multiple devices are arranged in the system. As shown in Figure 5, a host device, usually a CPU, interacts with one or more compute devices. Compute devices can be GPUs, FPGAs and even CPUs. Compute devices are logically composed of compute units, which, in turn, are composed of processing elements. The host manages compute devices through the OpenCL API.

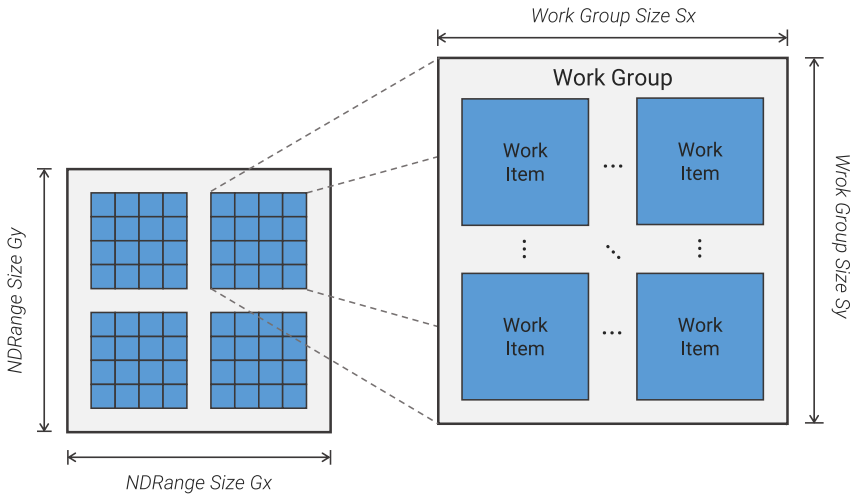


Figure 6 – OpenCL NDRange Execution Model (Adapted from: (2)).

Despite the compute unit's resemblance to a SIMD processor, the physical hardware implementation need not be organized in this manner. The platform model only defines a unified way by which devices of many different categories can be programmed seamlessly. This can cause problems of performance portability if the user does not take into consideration actual hardware features to optimize his code, as will be discussed in depth in Section 3.2.

2.2.2 Execution Model

The execution model defines how the workload is mapped to the compute devices. Figure 6 pictures this mapping. The workload is mapped to a structure called *NDRange*. NDRange can be specified using up to three dimensions. This tuple is commonly referred to as *grid geometry*.

The workload is partitioned in *work groups* that are scheduled in compute units. Work groups are further subdivided into *work items* which are scheduled in processing elements. All work items have access to their global IDs (relative to the total workload)

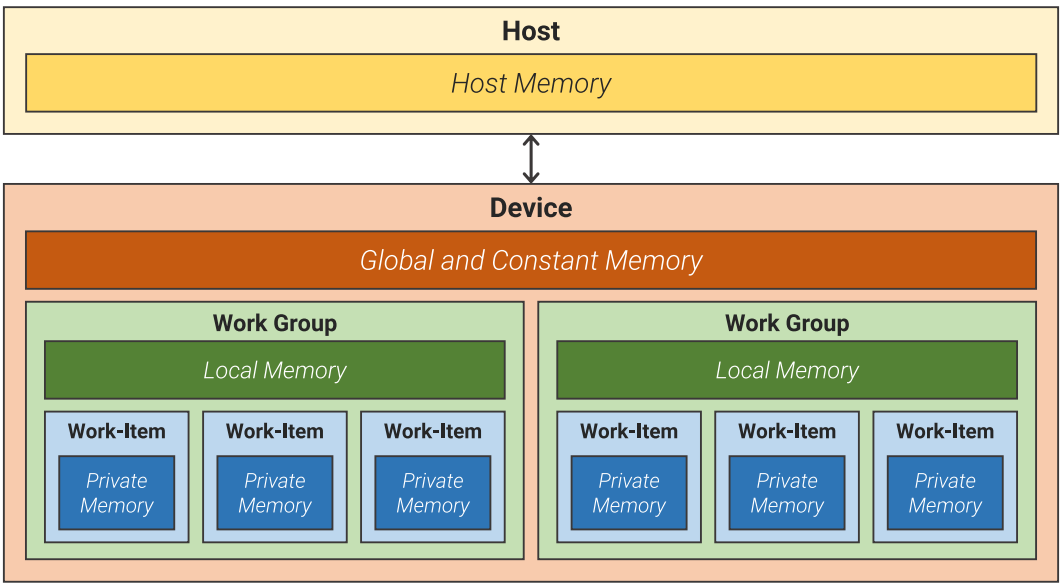


Figure 7 – OpenCL Memory Model (Adapted from: (2)).

and local IDs (relative to its work group). Work groups are scheduled independently and it is thus possible to execute them in parallel, however, synchronization can only be done in work items inside the same work group.

2.2.3 Memory Model

The memory model provides an abstraction of the memory hierarchy in the compute devices. Consider Figure 7, for instance. The device's memory is divided into four categories: global, constant, local and private.

Global memory is a memory region shared by all work items from all work groups and allows both reading and writing. It is usually the larger memory in the device, but also the slower. Co-located is the constant memory region, which is similar to the global memory but, as the name suggests, cannot be modified. This level in the hierarchy exists because some devices can optimize the reads knowing the data will not be modified. Further in the hierarchy, there is the local memory which is replicated in each work group

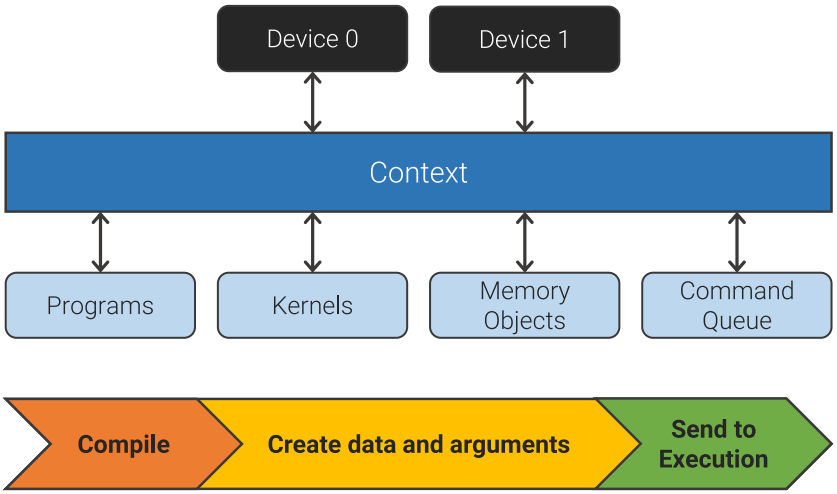


Figure 8 – OpenCL Programming Model (Adapted from: (2)).

and its data is only visible to work items within that group. They can be faster than global memory, but are also smaller. Finally, each work item has a its own private memory that is used for local variables.

2.2.4 Programming Model

The programming model defines a collection of objects created and managed by the host in order to execute an OpenCL program. These objects are summarized in Figure 8. Every OpenCL application need a *Context* which holds information about available platforms and devices and allows the creation of other objects such as *Programs*, *Kernels*, *Memory Objects* and *Command Queues*.

Programs are a collection of kernels. Kernels are individual functions that will be executed on the device. These functions are written in OpenCL language, that is a subset of C99. Memory objects are abstractions of allocated memory on the device. These objects can be explicitly transferred from host to device or mapped. The standard offers high-level abstractions of memory objects like images and buffers. Command queues are

used to manage and schedule tasks associated with a particular device. Some common commands are: transfer a buffer to or from the device, execute a kernel and etc.

The host application is in charge of creating these objects. First, the host program should create the program from a source code or binary file. As previously stated, the compilation can be done just-in-time (with source code) or ahead-of-time (with binaries). Since we are concerned with FPGAs, all the compilation is done ahead-of-time using the compiler provided by the hardware manufacturer. With the program set up, the host program loads the input data, set kernel arguments and specifies the grid geometry. Finally, the kernel is sent to execution through the command queue.

3 Code Optimizations and Related Work

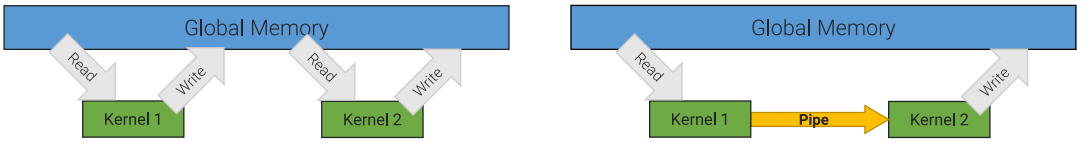
This chapter presents several code transformations for FPGAs and surveys related work that evaluates the impact of these transformations. Section 3.1 list code optimizations that can potentially improve performance of code running on FPGAs. Section 3.2 discusses recent works in the field of high-performance computing with FPGAs.

3.1 Code Optimizations

This section describes a collection of code transformations that can potentially increase performance in FPGAs. These transformations are often proposed by the hardware vendors in the form of optimization or best practices guides, as is the case with (4). Programmers should identify common code idioms or opportunities to apply the transformations. In this research project we limited the study to Intel® FPGAs, therefore some techniques described in this section are manufacturer specific. By the end of this section, we provide a discussion about their applicability and portability across devices.

3.1.1 Kernel Pipelining

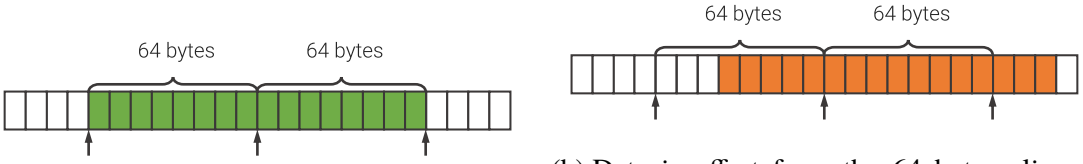
Frequently when writing OpenCL kernels, the computation is split between two or more kernels that execute in sequence. When one kernel finishes execution, the output data must be written to global memory, and transferred back to local memory, this time to another buffer so that the next kernel can begin execution. These back and forth transfers are, of course, expensive. To bypass this inefficiency, the Intel® FPGA SDK for OpenCL (29, 1) offers a library that allows data to be channeled from one kernel to the next directly as shown in Figure 9. Programmers are supposed to identify such redundant reads and writes then change the code accordingly. The OpenCL standard



(a) Global memory access without pipes

(b) Global memory access with pipes

Figure 9 – Pipelining data between kernels (Adapted from: (3)).



(a) Data is aligned to 64 bytes in main memory

(b) Data is offset from the 64 bytes aligned position in main memory.

Figure 10 – Aligned versus non-aligned memory.

version 2.0 later added similar functionality called “pipes”. Both pipes and channels are supported but pipes are standardized and should work with other SDKs. Instead of writing directly to a memory position, the programmer should call `read_pipe(ptr, val)` and `write_pipe(ptr, val)` to read and write to the pipe, respectively.

3.1.2 Direct Memory Access Alignment

When transferring data between host and device memory, it is possible to increase efficiency using direct memory access (DMA). However, in order to Intel® FPGAs make of use DMA hardware, the data is required to be 64 bytes aligned, as depicted in Figure 10. If this condition is not met, the CPU will stay busy transferring the data byte-to-byte. This way, it is recommended that the user allocates aligned memory by calling `_aligned_malloc` (C99, Windows), `posix_memalign` (C99, Linux) or `std::aligned_alloc` (C++17) instead of `malloc`.

3.1.3 Loop Unrolling

Loops inside the kernel can be optimized using the well known loop unrolling transformation (30). Intel® FPGA SDK for OpenCL provides the `#pragma unroll X` directive where *X* is the unroll factor, as the example shown in Listing 1. The compiler will try to unroll loops even if the user did not insert this pragma. However, it is possible to force no unrolling by using a factor of 1. According to the best practices guide, the FPGA compiler arranges the operations inside a loop in a pipeline fashion respecting the data flow semantics. Normally, one loop iteration is issued per cycle; therefore, if the loop is unrolled, more iterations can be finished each cycle, increasing its performance. Besides making the pipeline wider, the compiler will also coalesce memory accesses, further improving performance. The downside of this transformation is that it increases area resource utilization.

```
1:  #pragma unroll 2
2:  for (int k = 0; k < n; k++) {
3:    acc += A[i * n + k] * B[k * n + j];
4:  }
```

Listing 1 – Example of loop unrolling.

3.1.4 Replication and SIMD

It is possible to increase throughput by replicating the pipeline or creating SIMD (Single Instruction Multiple Data) units. The user can control these parameters by inserting attributes in the source code. First, `num_compute_units(N)` instructs the compiler to replicate the pipeline *N* times, so that multiple work-items can be computed in parallel. There is also the attribute `num_simd_work_items(N)` which controls the width of SIMD units. This process is demonstrated in Listing 2 These attributes works best when the computation is embarrassingly parallel, i.e., no synchronization is needed.

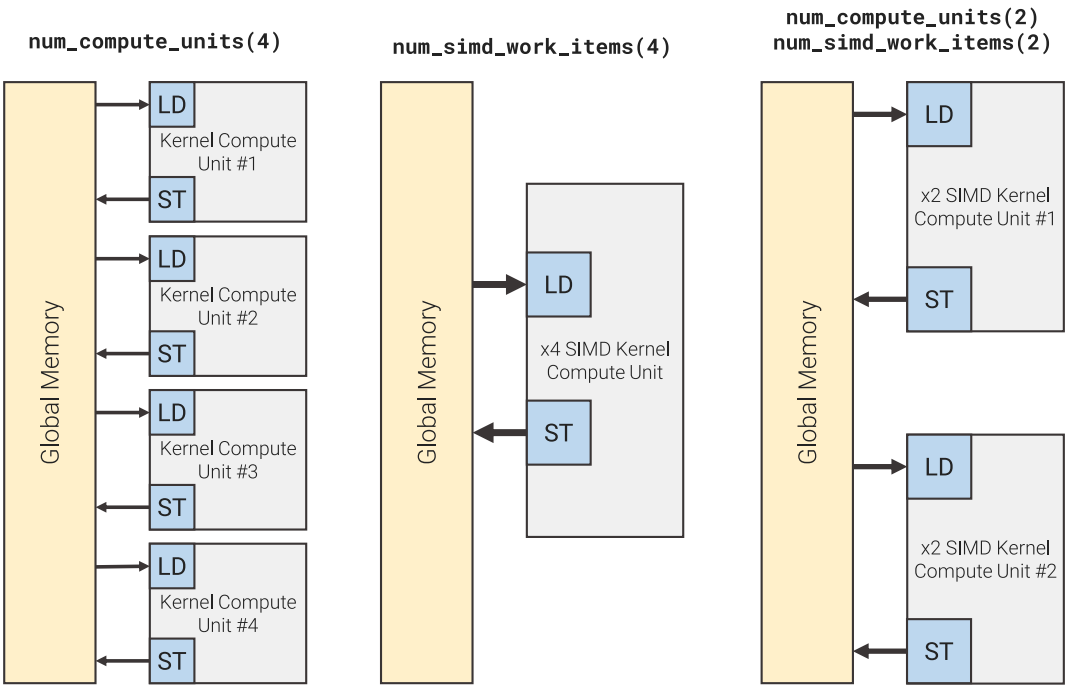


Figure 11 – Replication and SIMD (Adapted from: (4)).

```
1:  __attribute__((num_compute_units(4)))
2:  __kernel void kernel_replication() { /* ... */ }
3:
4:  __attribute__((num_simd_work_items(4)))
5:  __kernel void kernel_simd() { /* ... */ }
```

Listing 2 – Example of replication and SIMD.

Replicating the pipeline can occupy more area because load/store units are also replicated besides the pipeline, as is shown in Figure 11. Having excessively many pipelines can harm performance by putting too much pressure in the memory subsystem. To alleviate this problem, it is possible to use SIMD: a single and wider pipeline is kept while the load/store units are able to coalesce memory accesses thus improving memory throughput. These optimizations are not mutually exclusive and can be used together.

3.1.5 Reduction with Shift Registers

In scientific applications it is common to reduce a collection of data into a single variable using some operator. However, sequential reduction is slow because it creates a long string of data dependencies making it hard to parallelize the reduction loop. In the case of GPUs, it is commonplace to use the tree-based reduction to increase throughput. In the case of FPGAs, on the other hand, the device architecture provides shift registers, i.e., a register file dedicated to accelerate reduction and stencil computations. The Intel® FPGA SDK for OpenCL defines a code idiom that, when found in the code, the compiler infers the use of shift register and optimize the circuit.

```
1:
2:  __kernel void double_add(__global double *arr,
3:                           __global double *result,
4:                           int N)
5:  {
6:      double temp_sum = 0;
7:      for (int i = 0; i < N; i++)
8:          temp_sum += arr[i];
9:      *result = temp_sum;
10: }
```

Listing 3 – Redution without shift registers (Adapted from: (4))

```
1:  #define II_CYCLES 12
2:
3:  __kernel void double_add(__global double *arr,
4:                           __global double *result,
5:                           int N)
6:  {
7:      double shift_reg[II_CYCLES+1];
8:      for (int i = 0; i < II_CYCLES; i++)
9:          shift_reg[i] = 0;
10:     for (int i = 0; i < N; i++) {
11:         shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
12:         for (int j = 0; j < II_CYCLES; j++)
13:             shift_reg[j] = shift_reg[j+1];
14:     }
15:     double temp_sum = 0;
16:     #pragma unroll
17:     for (int i = 0; i < II_CYCLES; i++)
```

```
18:             temp_sum += shift_reg[i];
19:         *result = temp_sum;
20:   }
```

Listing 4 – Reduction with shift registers (Adapted from: (4))

Listing 3 shows a single work-item kernel that performs reduction of an array of doubles with no optimization. Listing 4 presents the same code optimized to infer shift registers. First, the programmer should define the variable that will represent the register (line 7) and set all its elements to 0 (lines 8-9). The size of the shift register must be larger than the number of cycles it takes to execute a single iteration of the reduction loop (Listing 3, lines 6-7). This number, called initiation interval (II), can be obtained in the kernel compilation report generated by the Intel® compiler (in this particular example, $II = 11$). Then, the programmer should add every element of the array being reduced to the shift register (line 11) and shift its positions (lines 12-13). Finally, a normal reduction can be performed on the shift register without loop-carried dependence (lines 15-19). Despite the optimized code being longer, the elimination of the loop-carried dependence enables overlapping of memory accesses and sums.

3.1.6 Sliding Window

Applications where the data are organized in a grid and the computation of each cell requires using the value of neighboring cells are commonly referred to as stencil applications. These programs usually put high demand on the memory subsystem since many redundant loads are executed across the computation of different cells. To address this problem, it is possible to use shift registers (as described in Section 3.1.5) to lower the number of redundant memory accesses. The shift registers keep recently used values in a window that slides as the iterations advance. The code idiom expected by the Intel® compiler is the same in Listing 4. The difference between the reduction and sliding window optimizations lie in the nature of the computation being optimized.

3.1.7 Single Work-Item Conversion

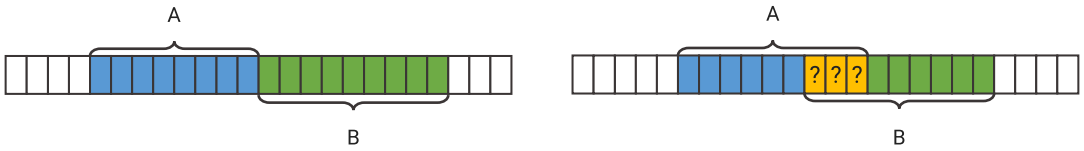
As stated previously, FPGAs are more suitable for single-threaded, pipelined work items, contrary to CPUs and GPUs that harness performance mostly by executing code in parallel (5). In OpenCL, kernels are commonly enqueued using the NDRange model which launches several work-items to be spread across the multiple computing units. Particularly for FPGAs, if there is a single compute unit, the work-items are issued sequentially to that pipeline. A better approach is to get rid of the NDRange model altogether and launch a single work-item. To achieve this, the user must wrap the kernel in loops (one for each dimension of the NDRange) and use the induction variables in place of the function `get_global_id(x)`, as is exemplified in Listings 5 (before transformation) and 6 (after transformation). On the host code, the call to `clEnqueueNDRangeKernel(...)` must be changed to `clEnqueueTask(...)`.

```
1:  __kernel void sum_ndrange(__global const int *restrict A,
2:                            __global const int *restrict B,
3:                            __global int *restrict C) {
4:    int i = get_global_id(0);
5:    C[i] = A[i] + B[i];
6:  }
```

Listing 5 – Sum of arrays A and B into C using the NDRange model.

```
1:
2:  __kernel void sum_singlewi(__global const int *restrict A,
3:                             __global const int *restrict B,
4:                             __global int *restrict C, const int n)
                                  {
5:    for (int i = 0; i < n; i++)
6:      C[i] = A[i] + B[i];
7:  }
```

Listing 6 – Sum of arrays A and B into C using the single work-item model.



(a) Arrays pointed by A and B do not alias.

(b) Arrays pointed by A and B do alias.

Figure 12 – Arrays without and with aliasing, respectively. Note that, when the arrays alias, data in the intersect region belongs to both arrays at the same time. Therefore, a write operation on A can potentially modify B, creating a memory dependence. If the dependence is not present, the arrays can be modified independently.

3.1.8 Restrict Parameters

The Intel® optimization guide also encourages users to mark pointers as `restrict` in kernel parameters. This keyword tells the compiler that the pointer does not alias with other pointers in the program. Without this information, the compiler must be conservative and create data dependencies to ensure correctness. These dependencies are potentially useless and will prevent the compiler to issue loads and stores in parallel, degrading throughput. Of course, the user itself must guarantee that the pointers do not alias otherwise undefined behavior can be expected.

Take Figure 12a for instance. We see two pointers A and B that do not alias, *i.e.*, there is no overlapping. As a matter of fact, writing to any position in A will not change any position in B. In this case, the pointers are eligible to be marked `restrict`. In Figure 12b, however, the pointers do overlap and a memory dependency must exist between pointers A and B to ensure correctness.

3.1.9 Loop Collapse

Loop nests can be converted to a single loop using the well known loop collapsing transformation (30) (also referred to as "loop coalescing"). The Intel® FPGA SDK for OpenCL provides the `#pragma loop_coalesce` directive followed by an optional

nesting level. According to the Intel® Programming Guide, coalescing loops can reduce area usage by reducing the overhead needed for loop control. For NDRange kernels, loops can potentially be coalesced automatically by the compiler, increasing throughput and saving area.

```
1:  #pragma loop_coalesce
2:  for (int i = 0; i < n; i++) {
3:      for (int j = 0; j < m; j++) {
4:          // ...
5:      }
6:  }
```

Listing 7 – Collapsing loops using the pragma available in Intel® SDK for FPGAs.

3.1.10 Branch Variant Code Motion

It is sometimes the case when both branches of the same if-else structure execute intersecting code. When this happens, the compiler will generate redundant circuits for both sides of the branch. In some cases, however, it is possible to move the common code outside of the if-else structure thus eliminating the need for replication. This is called branch variant code motion and it can reduce the area usage of the circuit. Less area usage means lower power consumption or more area available for other types of computation. Observe Listings 8 and 9 for instance. The calls to functions `before()` and `after()` can be moved outside the loop in order to occupy a smaller device area while preserving program correctness.

```
1:  if (condition) {
2:      before();
3:      var = var + 1;
4:      after();
5:  } else {
6:      before();
7:      var = 0;
8:      after();
```

```
9:  }
```

Listing 8 – Before branch variant code motion

```
1:  before();
2:  if (condition) {
3:      var = var + 1;
4:  } else {
5:      var = 0;
6:  }
7:  after();
```

Listing 9 – After branch variant code motion

3.1.11 Static Memory Coalescing

If the kernel accesses consecutive non-private memory locations then it is a good candidate for memory coalescing. Suppose a kernel accesses four consecutive memory locations therefore issuing four separate loads or stores. If this pattern can be determined statically, the user can redeclare the vector of type `float*` to the type `float4*`, which should be a smaller vector where each element is four floats wide. The benefit of this transformation it that, when loading or storing values of this type, there is a single coalesced memory access. Coalescing can increase memory throughput, raising kernel performance. The Intel® SDK tools will try to automatically coalesce memory accesses when the kernel is vectorized, although memory coalescing might also be possible in non-vectorized kernels. Listing 10 shows an OpenCL kernel without static memory coalescing whereas Listing 11 show the transformed version of the same kernel capable of loading/storing four floating point elements with a single access.

```
1:  __kernel void summation(__global const float *restrict a,
2:                          __global const float *restrict b,
3:                          __global float *restrict answer)
4:  {
5:    size_t gid = get_global_id(0);
6:    answer[gid * 4 + 0] = a[gid * 4 + 0] + b[gid * 4 + 0];
```

```
 7:    answer[gid * 4 + 1] = a[gid * 4 + 1] + b[gid * 4 + 1];
 8:    answer[gid * 4 + 2] = a[gid * 4 + 2] + b[gid * 4 + 2];
 9:    answer[gid * 4 + 3] = a[gid * 4 + 3] + b[gid * 4 + 3];
10:  }
```

Listing 10 – No static memory coalescing (Adapted from: (4))

```
1:  __kernel void summation(__global const float4 *restrict a,
2:                          __global const float4 *restrict b,
3:                          __global float4 *restrict answer)
4:  {
5:    size_t gid = get_global_id(0);
6:    answer[gid] = a[gid] + b[gid];
7:  }
```

Listing 11 – Memory accesses coalesced (Adapted from: (4))

3.1.12 Discussion

In the previous sections, several code optimizations that benefit kernel performance were shown. Nonetheless, the programmer is still faced with the question as to which optimizations to apply and how to tune them. This is not a trivial question for multiple reasons. A naive approach could consist of applying them all—when applicable—however, most likely the hardware limits (*i.e.* FPGA area) will pose a barrier. For example, consider the optimizations replication, SIMD and loop unrolling. In theory, both of them increase the kernel throughput by duplicating elements. Notwithstanding, it remains unclear as to which cases one should be favored in spite of the others. Additionally, the long time to place and route a circuit slows down research that aims at completely characterizing the performance of such code idioms.

Another relevant question regards applicability of such optimizations to code running on hardware from a different vendor, like Xilinx®. Similarly to Intel®, Xilinx® also provides a document with guidelines for profiling and optimizing kernels (31, 32). However, not every optimization available to Intel® FPGAs are available to Xilinx®'s. For those that are indeed available in both vendor's compilers, they are often expressed

differently in code. As mentioned before, in order to unroll a loop in an OpenCL kernel for Intel® the programmer should write `#pragma unroll <X>`. On the other hand, the same is achieved with the attribute `__attribute__((openc_unroll_hint))` preceding the loop. We argue that because FPGA adoption is still in its infancy, there was not enough time for standards to be developed. A unified way to program such devices would greatly benefit further research. Meanwhile, developing a tool capable of taking platform agnostic code and automatically optimize it for specific hardware/vendor would accelerate progress and adoption, like some works leaned in that direction (3, 33).

Last, we argue that the lack of open source tools hinders innovation. Both Intel® and Xilinx®'s SDKs require a license to unlock all features. The existence of open source compilers capable of fully place and route circuits would enable these optimizations to be done in the intermediate representation level and possibly reveal previously unknown opportunities for optimizing code.

3.2 Related Research

In the context of optimizing code for FPGAs, Zohouri *et al.* (5) evaluate the impact some optimizations has on power and performance when comparing FPGA against a CPU and GPU. For that matter, the author applied four different transformations by hand on Rodinia applications: (a) loop unrolling; (b) replication and SIMD; (c) reduction with shift registers; and (d) sliding window optimization.

Figure 13 presents the flow of compilation proposed by Zohouri *et al.*. In this scheme, the programmer should manually inspect the OpenCL kernel in search of opportunities to apply transformations. As described in Section 3.1, there is a handful of common code idioms that can be rewritten in order to harness improved kernel performance. The authors selected few kernels from Rodinia (34) and applied transformations incrementally, measuring execution time and power consumption. The

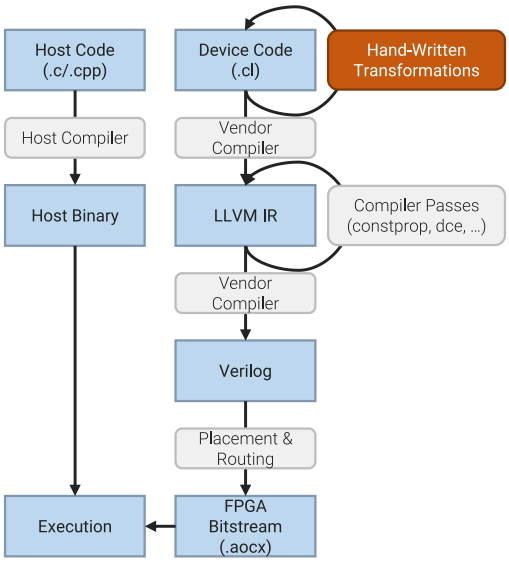


Figure 13 – Flow of optimization proposed by Zohouri *et al.*. In this scheme, the user manually modify the device code by applying transformations by hand. The rest of the workflow continues normally (Adapted from: (5)).

results are summarized in Figure 14, where they compared a CPU (Intel® E5-2670), a GPU (Nvidia® K20c) and a FPGA (Intel® Stratix V).

From Figure 14, we can observe that FPGAs excel CPUs in terms of power efficiency in every benchmark, and loose only once to GPUs (CFD). The most significant power efficiency achieved was 12.3x (NW). As expected, the FPGA stays below the GPU in terms of speedup in every benchmark. In relation to the CPU: three instances show comparable speedup (Pathfinder, SRAD, LUD); two cases where there was a large slowdown (Hotspot, CFD); and one case where a significant speedup of 3.07x was measured (NW).

Lloyd *et al.* (6) took the process of transforming kernels one step further and automated it using compiler passes. The authors describe a method for integrating host and device compilers by propagating information from one another thus revealing new optimization opportunities. Their solution only works with the Intel FPGA SDK for

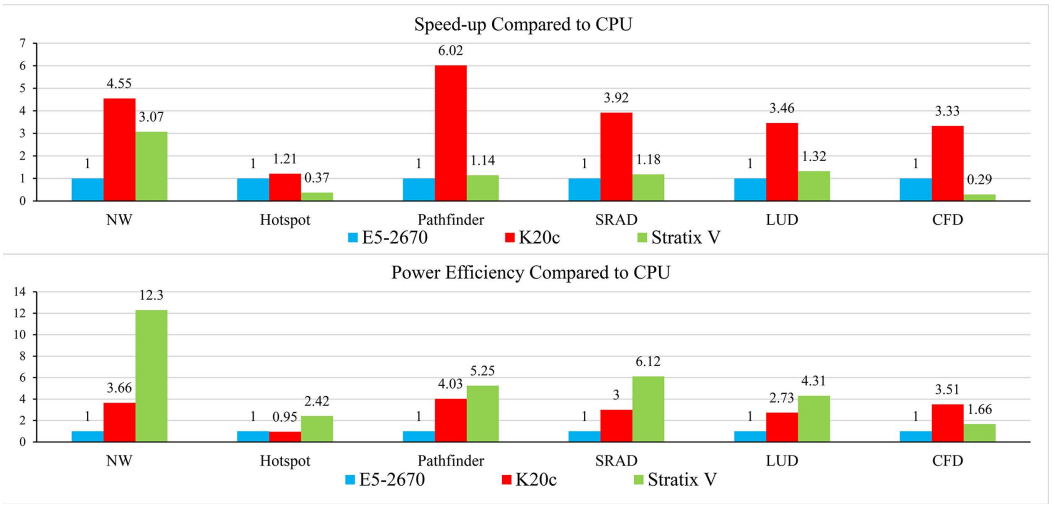


Figure 14 – Comparison of speedup and power efficiency between a CPU (Intel® E5-2670), GPU (Nvidia® K20c) and FPGA (Intel® Stratix V) (Source: (5)).

OpenCL (29) since the device passes are embedded into the SDK tools. Three transformations were selected: (i) NDRange to single work-item conversion (Section 3.1.7); (ii) reduction with shift registers (Section 3.1.5); and (iii) restrict parameters (Section 3.1.8).

As can be seen in Figure 15, this solution is more elaborate. With both host and device code translated to LLVM's intermediate representation (IR), the OpenCL invocation pass ❶ is run on the host, and it produces a report with the following information: (a) is the kernel a task or NDRange?; (b) in case of the second, can the grid geometry be determined statically?; (c) can the pointers passed to the kernel be marked as `restrict`? In other words, can we guarantee that they will not alias[1]?. The second step consists of transforming the kernel from NDRange to single work-item ❷. Using the grid geometry information from the host, the pass wraps the kernel with loops,

[1] In the actual implementation, memory pointers passed to `setKernelArg` are traced back to their definitions. If each memory region was allocated from different calls to `malloc`, the pointers are assumed not to alias. Otherwise, the compiler takes the conservative approach and assume that the pointers do alias. This is simple yet effective since the majority of scientific workloads have similar structure: allocate memory, read data, compute, write data, free memory.

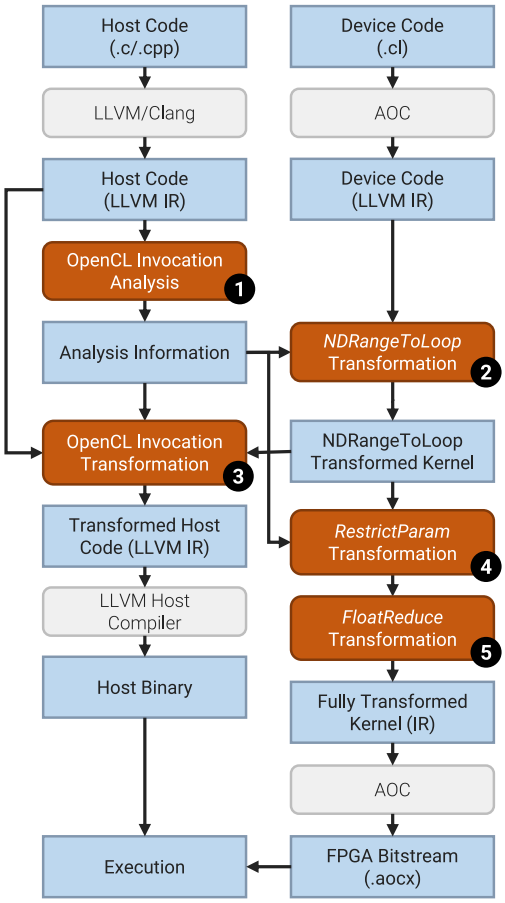


Figure 15 – Flow of transformations proposed by Lloyd *et al.*. In this scheme, information if propagated between host and device compiler. In order to decide if a transformation can be applied in device code, the compiler has to perform an analysis on the host first. (Adapted from: (6)).

eliminating calls to `get_global_id` and `get_local_id`, since the loop indices now serve as the grid geometry ids. If the grid geometry was available statically, the LLVM pass also performs constant propagation, otherwise, these values are passed as parameters to the kernel. With the kernel transformed to single work-item, the host invocations must be updated ❸. Calls to `clEnqueueNDRange` must be changed to `clEnqueueTask` and new calls to `setKernelArg` must be placed for the grid geometry variables. If the host analysis pass can guarantee that the memory pointers do not alias, the kernel parameters

are marked with keyword `restrict` ❹. Finally, a transformation to modify reduction operations to use shift registers ❺ is applied given that the code has the necessary structure. The remaining of the compilation flow runs normally for both host and device.

The proof-of-concept was tested on a Terasic DE5-Net board, which contains an Intel Stratix V GX FPGA with 4GB of DDR3 memory. The SDK version used was 16.1. The selected applications from Rodinia were: Gaussian, Hotspot3D, Kmeans, NN and SRAD. The highest speedup achieved was in gaussian (6.69x) in which the NDRangeToLoop and FloatReduce transformations were applied. The second best speedup was with hotspot3D (2.6x) (RestrictParam and NDRangeToLoop). With nn and srad, small differences in execution time were measured (0.98x and 1.06x respectively). A large slowdown of 0.36x was measured with kmeans applying RestrictParam and NDRangeToLoop. No power efficiency measurement was provided in this work.

Although the results look less promising in this approach, it has the benefit of being automatic. Still, the prototype works by sneaking those passes into the script that drives the synthesis of OpenCL to bitstream. Because of that, this solution is not guaranteed to work with newer versions of the SDK. Reusing this prototype with other toolchains (e.g. Xilinx's Vivado HLS (35)) can prove to be even more challenging, if possible at all.

Another relevant branch of research consists of using directive-based frameworks for reconfigurable computing. Lee *et al.* (3) proposes a programming framework that takes OpenACC (36) annotated code and synthesize high-performance hardware from it. Their solution, implemented in the OpenARC (37) compiler, is capable of performing source-to-source translation of C code to OpenCL while performing optimizations during this process. Initially, the kernel pipelining and DMA alignment optimizations were implemented. In a follow-up publication by the same group (33), another five transformations were implemented: reduction with shift registers, sliding window, loop

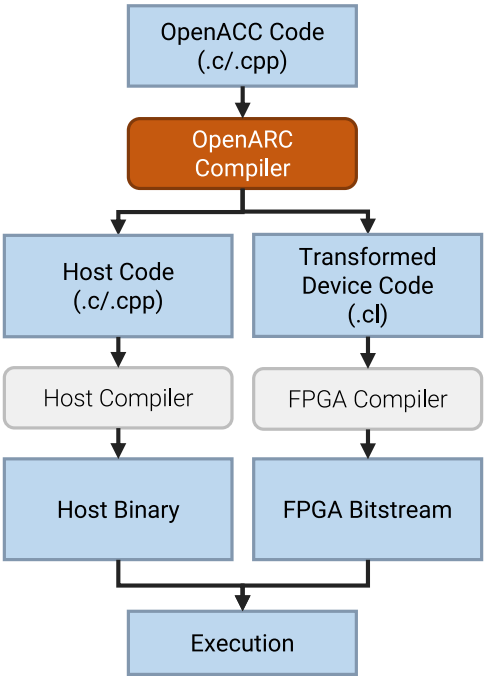


Figure 16 – Inferred flow of optimization proposed by Lee *et al.*. In this scheme, the user writes OpenACC annotated code and the OpenARC compiler generates optimized host and devices sources automatically. (Adapted from: (3)).

collapsing, conversion to single work-item and branch variant code-motion.

As shown in Figure 16, unlike the challenge faced by Lloyd *et al.* (6), where they needed host and device compiler integration to enable automatic optimizations, the compiler has control over the OpenCL code generated, therefore it can perform transformations directly into the abstract syntax tree effortlessly and output the kernel compilation unit ready for synthesis.

In (3), the authors tested their solution on a Intel Stratix V GS D5 FPGA, 8GiB DDR3 memory with the Intel SDK version 15.0. For two Rodinia OpenACC benchmarks (FFT-1D, FFT-2D), the speedup was two orders of magnitude larger over sequential CPU, surpassing even GPUs. In Jacobi, MatMul, Hospot, NW and SRAD, there was no speedup

or a speedup of one order of magnitude larger, but below GPUs. The only benchmark to present a slow down of one order of magnitude is Spmul. These results were obtained by selecting the fastest version of the code by applying one or more optimizations (loop unrolling, kernel pipelining and DMA alignment).

Additionally, the results presented in (33) shows that Sobel application surpass both CPUs (baseline) and GPUs in terms of speedup and power efficiency: 6.3x and 42.3x respectively. FD3D presents similar behavior with 1.4x speedup and 8.8x power efficiency. Benchmark NW performed poorly in both measurements with 0.1x slowdown and 0.6x power efficiency.

Table 1 – Summary of code transformations studied in previous works.

Code Transformation		Reference			
		(3)	(5)	(6)	(33)
3.1.1.	Kernel Pipelining	✓			
3.1.2.	Direct Memory Access Alignment	✓			
3.1.3.	Loop Unrolling	✓	✓		
3.1.4.	Replication and SIMD	✓	✓		
3.1.5.	Reduction with Shift Registers		✓	✓	✓
3.1.6.	Sliding Window		✓		✓
3.1.7.	Single Work-Item			✓	✓
3.1.8.	Restrict Parameters			✓	
3.1.9.	Loop Collapsing				✓
3.1.10.	Branch Variant Code-Motion				✓
3.1.11.	Static Memory Coalescing				
Automatic		✓		✓	✓
Directive-Based		✓			✓

To conclude this section, Table 1 presents a summary of which transformation each previous works evaluated. Apart from (5), all of the other related work apply

transformations automatically, i.e., it is required little to no intervention from the user. The works (3) and (33) uses OpenACC for directive-based computing. Also, we could not find any related work that explores memory coalescing and its benefits.

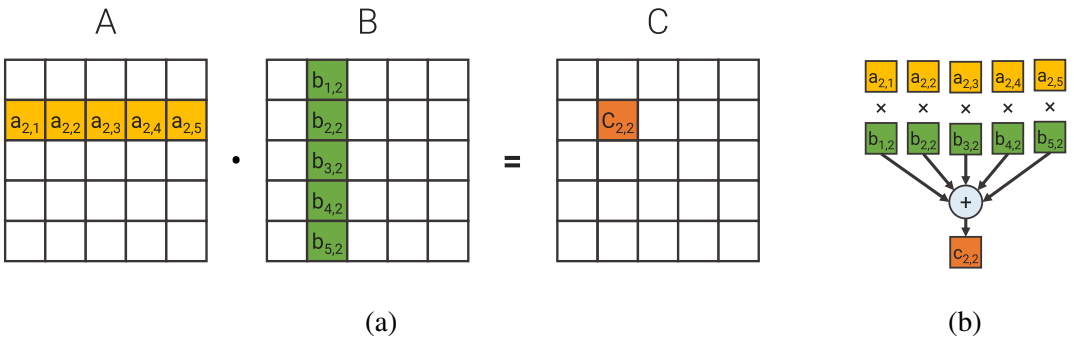
4 Experimental Analysis

In this chapter we demonstrate the effects of some optimizations on performance. This chapter is organized as follows: in Section 4.1 we present the materials used for this experimental analysis, like hardware, operating system and other software; in Section 4.2, we describe the methodology adopted in this experimental analysis, show our results and also provide a discussion of the results we obtained.

4.1 Materials

This experimental analysis was tested on a computer provided by the University of Alberta, Canada. The machine is powered by an Intel® Core™ i7-4770 @ 3.40GHz with 32GB of RAM running on Ubuntu 16.04.5 LTS Codename Xenial. This machine has also a Terasic DE5-Net board that holds an Intel® Stratix® V FPGA (5SGXA7) and 4GB 1600MHz DDR3 RAM connected to the CPU through a x16 PCIe 2.0 bus capable of transferring 8000MB per second. The FPGA has 234720 ALMs, 938880 registers, 2560 memory blocks, and 256 DSP blocks (38). The organization of the ALM was shown in Section 2.1, Figure 1.

The host C and C++ compilers used were the ones available in the GNU Compiler Collection (GCC) version 7.4.0. Our benchmarks were written in C++17. For device code compilation we used the Altera Offline Compiler bundled in the Intel® FPGA SDK for OpenCL version 18.0.0.614.



(a) (b)

Figure 17 – Visual representation of the matrix multiplication operation $C = AB$. Element $c_{i,j}$ equals the vector product between the i-th row in A and the j-th column in B.

4.2 Results

In order to evaluate the impact of code transformations, we selected the naive matrix multiplication algorithm as the baseline. Let A and B be matrices with sizes $n \times m$ and $m \times o$, respectively. The multiplication of A and B results in a matrix C of size $n \times o$. Element $c_{i,j}$ of C equals the dot product between i-th row in A and j-th column in B. This algorithm pseudocode is shown in Listing 12 and it can be noted that this algorithm has three nested loops. If $n = m = o$, then we can express its time complexity as $O(n^3)$. Figure 17 depicts this computation visually.

```
1:  for (int i = 0; i < n; ++i) {
2:    for (int j = 0; j < o; ++j) {
3:      C[i,j] = 0;
4:      for (int k = 0; k < m; ++k) {
5:        C[i,j] += A[i,k] * B[k,j];
6:      }
7:    }
8:  }
```

Listing 12 – Algorithm of the naive matrix multiplication algorithm.

The naive matrix multiplication algorithm implemented as an OpenCL kernel is shown in Listing 13. For simplicity, this implementation only multiplies square matrices

of size $n \times n$. It shall be noted that this kernel follows the NDRange execution model (Section 2.2.2). Upon execution, multiple work-items will be launched with different IDs to perform the computation. This is the reason this code does not have the familiar structure with three nested loops.

```
1:  __kernel void matrix_multiply(__global const int *A,
2:                                __global const int *B,
3:                                __global int *C,
4:                                const unsigned int n) {
5:
6:    int i = get_global_id(0);
7:    int j = get_global_id(1);
8:    int acc = 0;
9:
10:   for (int k = 0; k < n; k++) {
11:     acc += A[i * n + k] * B[k * n + j];
12:   }
13:
14:   C[i * n + j] = acc;
15: }
```

Listing 13 – OpenCL kernel for computing matrix multiplication.

We use the code in Listing 13 as our baseline and apply optimizations incrementally. The code optimizations evaluated are: (i) DMA Alignment; (ii) Loop Unrolling; (iii) Replication; (iv) Restrict Parameters.

The time measurement was done through the OpenCL profiling API that allows the programmer to measure how long a command in the queue took to execute. A command can be an NDRange kernel or a buffer transfer to/from the device. For each execution, we collect 16 samples and discard the first one. This is important because the bitstream is programmed onto the FPGA when the kernel is actually run for the first time, incurring overhead. For subsequent runs of the same kernel, the FPGA does not need to be reprogrammed. With the 15 remaining samples, we calculate the arithmetic mean and standard deviation. In the case of loop unrolling and replication, we use this method twice: once with kernel parameters marked as `restrict` and once without it.

4.2.1 Memory transfer

Our first benchmark compares transfer speeds of aligned vs unaligned data. We start by allocating unaligned memory, fill it with random data, and finally write and read a buffer stored in the FPGA memory. This is achieved by calling the OpenCL functions `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`, respectively. The initial buffer is 10MiB and it is increased in steps of 10MiB until 100MiB. The data is presented graphically in Figures 18a (unaligned) and 18b (aligned).

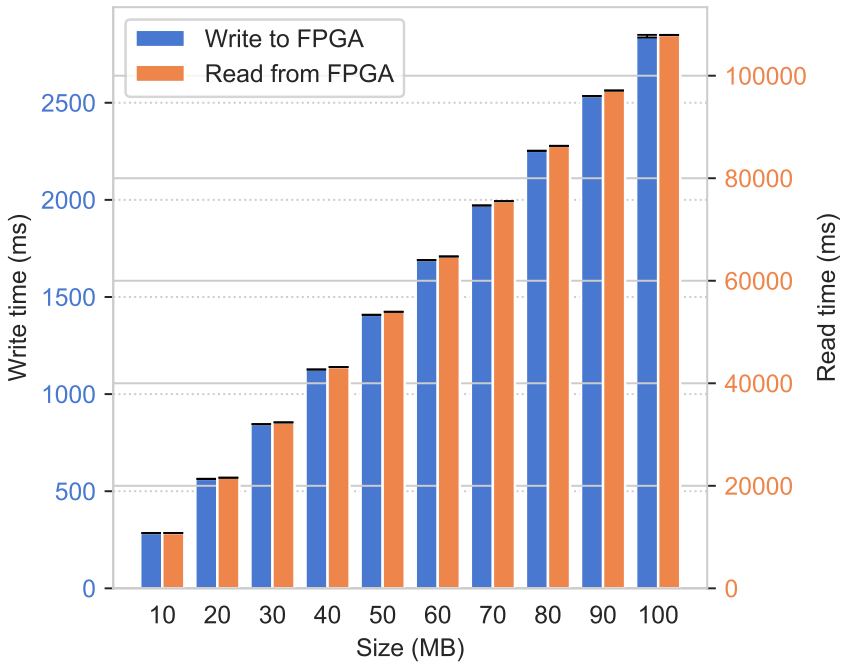
As can be observed from Figure 18a, time to read data from the FPGA is one order of magnitude larger compared to writing when the data is not aligned. The PCIe interface bandwidth is symmetrical, meaning that data is transferred through the bus with the same speed in both directions. We would need access to the concrete hardware design of the FPGA in order to find out the causes of this anomaly. Of course, the chip design the is intellectual property of Intel®.

As shown in Figure 18b, data transfer times are indeed reduced and both reading and writing to FPGA memory show similar magnitude in case of aligned data transfer. The speedup for transferring data from FPGA to host (read) goes up to 412.5x whereas when transferring data from host to the device (write) a speedup of 8.3x is achieved.

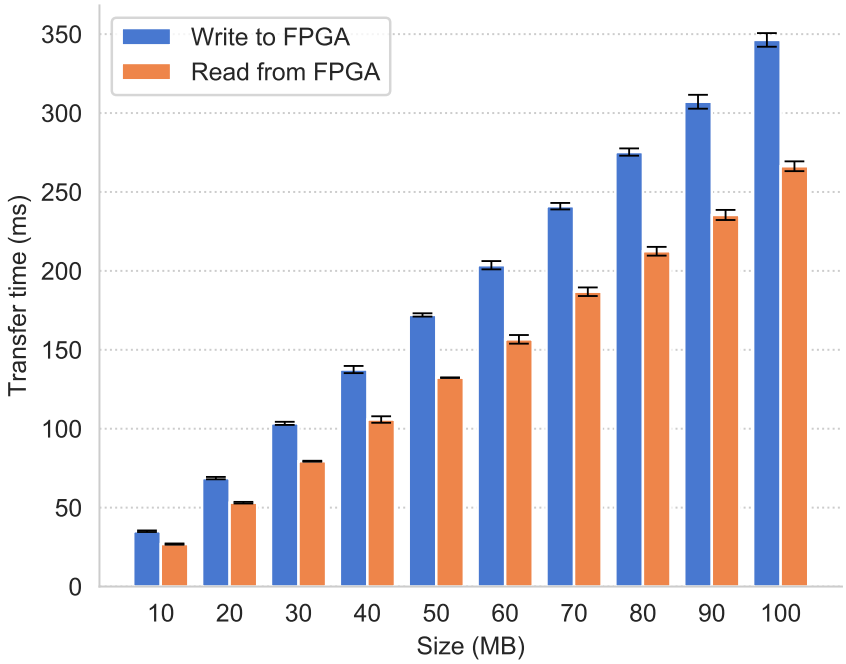
4.2.2 Loop Unrolling

The loop unrolling optimization was evaluated in the matrix multiplication kernel. In order to transform the source, it suffices to add the unroll pragma shown in Section 3.1.3 to the line preceding the loop. The kernel was recompiled with factors 1 (no unrolling), 2, 4 and 8 and tested with matrices containing 1024×1024 elements. The computation was divided into 32 work-items per work group[1]. The kernel execution time measurements can be observed in Figure 19.

[1] The OpenCL work group size was determined empirically.



(a) Unaligned data transfer.



(b) Aligned data transfer.

Figure 18 – Aligned versus unaligned data transfer.

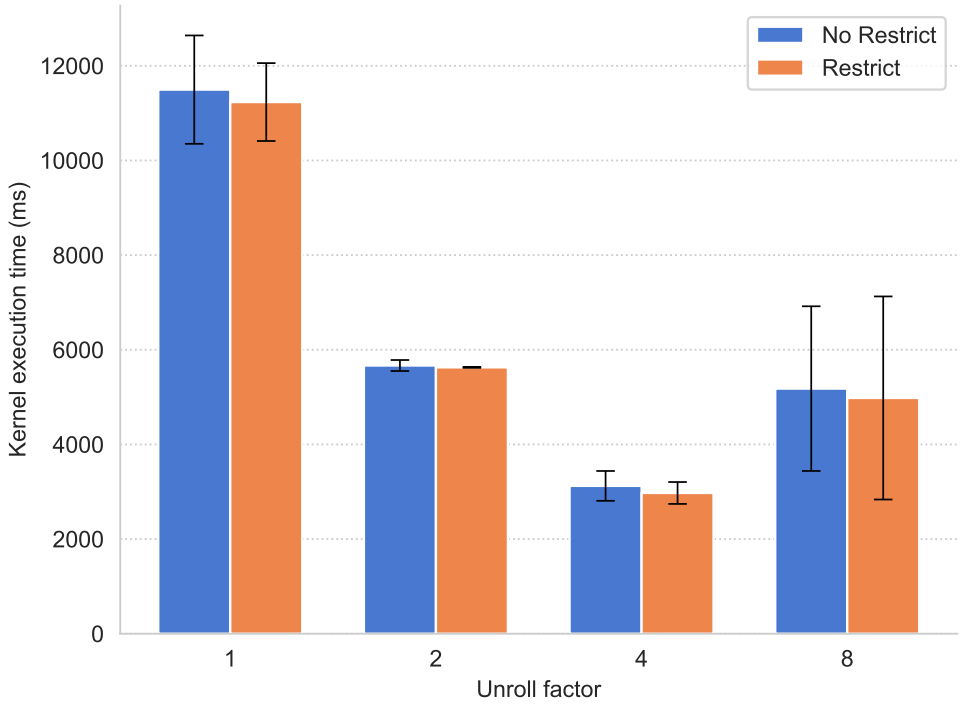


Figure 19 – Matrix multiplication kernel with loop unrolling. The graph shows unroll factor versus execution time. Matrices A and B sizes are 1024×1024 elements and the computation was divided into 32 work-items per work group.

Without unrolling the loop the execution time measured is 11.23 seconds on average. Unrolling the loop two times reduces the execution time by half, resulting in 5.63 seconds on average (1.99x speedup). With loop unrolling factor 4, the average execution time is further reduced to 2.97 seconds (3.78x speedup). Unrolling 8 times show a speedup of 2.25x, however, apart from yielding a lower speedup when compared to unrolling 4 times, the measurements indicate the high deviation from the mean.

Table 2 contains the area information for the loop unrolling kernel with parameters marked as `restrict`. This table presents how many components were necessary to assemble the circuit. The components are: arithmetic and logic units (ALUTs); Flip-Flops (FFs); RAM blocks (RAMs); and DSPs. It is possible to observe that unrolling the matrix multiplication kernel does not impose a great penalty in area utilization. The critical

Table 2 – Area usage of matrix multiplication kernels with loop unrolling and parameters marked with `restrict` keyword. No significant difference in area usage was observed when compiling without `restrict`.

Kernel	ALUTs	FFs	RAMs	DSPs
Unroll 1	9229 (2%)	12877 (1%)	102 (4%)	8 (3%)
Unroll 2	13153 (3%)	16877 (2%)	145 (6%)	14 (5%)
Unroll 4	20593 (4%)	26494 (3%)	231 (9%)	26 (10%)
Unroll 8	35846 (8%)	45740 (5%)	403 (16%)	50 (20%)

component in this circuit are DSPs. These blocks are capable of doing multiplication and, since there is a multiplication inside the loop, these blocks tend to grow very fast. Unrolling the loop eight times uses 20% of the DSPs in the board. The second most used component are the memory blocks that reach 16% in the worst case. No significant difference in area utilization was noticed without marking parameters as `restrict`.

4.2.3 Compute Unit Replication

As introduced in Section 3.1.4, compute unit replication make an identical copy of the pipeline so that more computation can be done in parallel. This is done by prepending the kernel declaration with the `num_compute_units` attribute. The matrix multiplication kernel was replicated with 1, 2, 4, 8 and 16 compute units and executed on matrices of size 1024×1024 elements with 16 work-items per work group[2]. The data is shown in Figure 20.

Using a single compute unit, the matrix multiplication is computed in 14.1 seconds on average. With 2 compute units, however, kernel execution time is halved to 6.73 seconds (2.09x speedup). With 4 compute units, execution time drops to 4.44 seconds (3.17x). Above 4 compute units, returns start to diminish—4.5 seconds (3.13x) with 8 compute units and 4.02 seconds (3.51x) with 16 units. Again, no significant

[2] The OpenCL work group size was determined empirically.

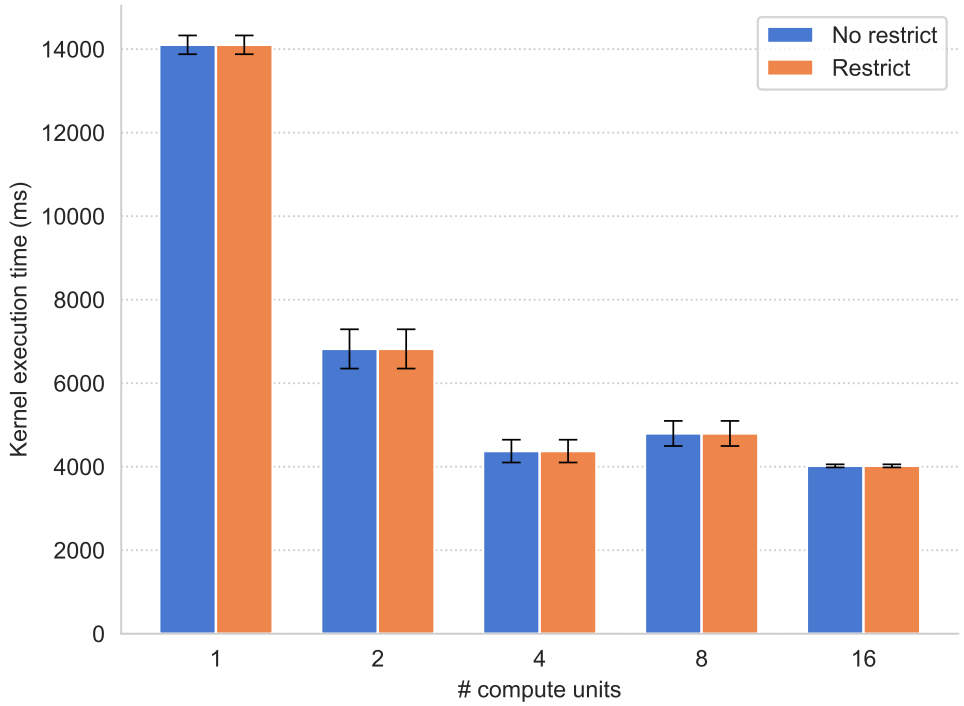


Figure 20 – Matrix multiplication kernel with compute unit replication. The graph shows number of units versus execution time. Matrices A and B sizes are 1024×1024 elements and the computation was divided into 16 work-items per work group.

difference in execution time was noticed when using `restrict` keyword.

The area utilization data is presented in Table 3. Memory (RAM) blocks are critical when replicating compute units. The reason is that not only the pipeline is replicated but also the memory subsystem. Apart from using more area, having multiple load and store units can put too much pressure on the board memory. This is made evident when comparing the best execution time with loop unrolling (2.97s) against the best execution time when replicating compute units (4.02s). Also, observe that replicating the pipeline 4 times uses more DSPs (64) than unrolling the loop 4 times (50).

Table 3 – Area usage of matrix multiplication kernels with compute unit replication and parameters marked with `restrict` keyword. No significant difference in area usage was observed when compiling without `restrict`.

Kernel	ALUTs	FFs	RAMs	DSPs
1 units	9245 (2%)	12909 (1%)	102 (4%)	8 (3%)
2 units	17629 (4%)	24670 (3%)	204 (8%)	16 (6%)
4 units	34333 (7%)	48064 (5%)	408 (16%)	32 (13%)
8 units	67805 (14%)	94980 (10%)	816 (32%)	64 (25%)
16 units	135005 (29%)	189324 (20%)	1632 (64%)	128 (50%)

4.2.4 Loop Unrolling + Compute Unit Replication

The previous two optimizations were combined: we unroll the loops two times for every kernel and also replicate compute units 1, 2, 4 and 8 times, respectively. We test this code with matrices of 1024×1024 elements and 16 work-items per work group. The data is represented in Figure 21.

The baseline kernel with 1 unit executes, on average, in 6.5 seconds. No significant improvement is observed when executing the kernel with 2 units. Using 4 compute units a speedup of 1.30x is measured (4.98s on average). Finally, replicating the pipeline 8 times produces a speedup of 1.27x (5.14s). One more time, the restrict parameter optimization seems to be irrelevant to performance.

Table 4 compiles the area utilization for this benchmark. Memory blocks and DSPs are even more critical with this benchmark. As expected, the kernel unrolled and replicated 8 times uses 1160 (45%) memory blocks and 112 (44%) DSPs compared to 816 (32%) RAMs and 64 (25%) DSPs for the kernel only replicated 8 times.

4.2.5 Discussion

In this experimental analysis we showed that kernel performance can be significantly improved by applying the right optimizations. By allocating aligned memory,

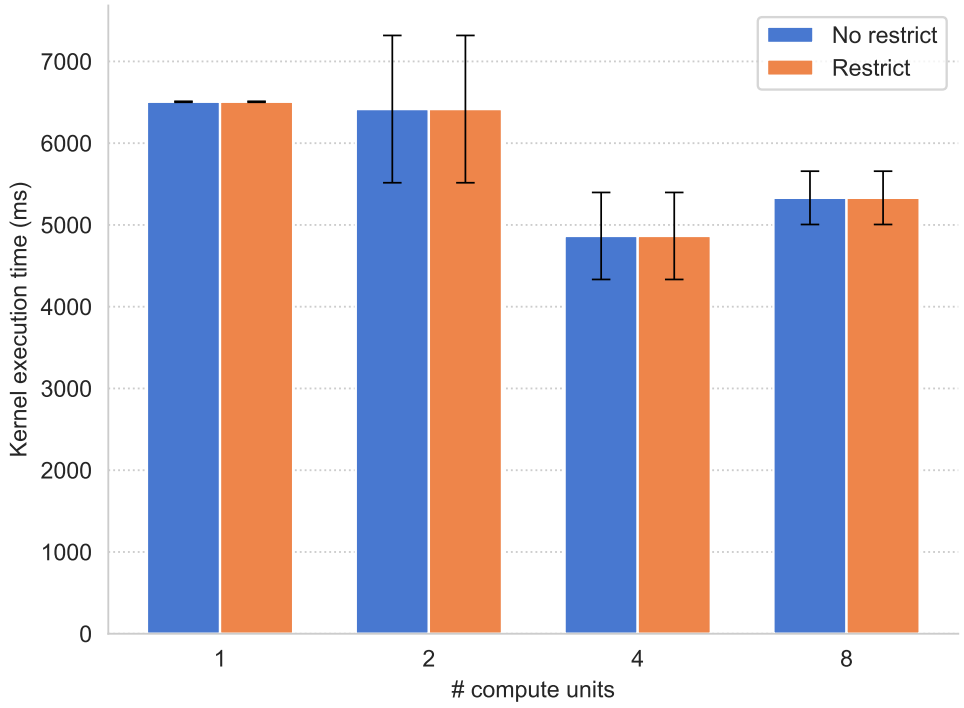


Figure 21 – Matrix multiplication kernel with compute unit replication and loop unrolling. The graph shows number of units versus execution time. The unroll factor is fixed at 2 for every kernel. Matrices A and B sizes are 1024×1024 elements and the computation was divided into 16 work-items per work group.

it was observed up to 412.5x speedup for transfer from the device to the host and 8.3x from the host to the device. It was also observed that it is possible to speedup a matrix multiplication kernel by 3.78x by unrolling loops and 3.17x by replicating compute units. It should be noted that both of these optimizations aims at increasing kernel throughput, however, by unrolling loops a higher performance is achieved with less penalty in circuit area when compared to compute unit replication. We conclude that, for kernels that have loops, the loop unrolling optimization should be favored over replication. Nonetheless, for kernels that do not have loops, replicating compute units should provide a good enough speedup. We also observed that the performance gain was much lower (1.30x) and the area usage was much higher when combining both optimizations. From that we conclude

Table 4 – Area usage of matrix multiplication kernels with compute unit replication, loop unrolling (factor 2) and parameters marked with `restrict` keyword. No significant difference in area usage observed when compiling without `restrict`.

Kernel	ALUTs	FFs	RAMs	DSPs
1 units/unroll 2	13153 (3%)	16877 (2%)	145 (6%)	14 (5%)
2 units/unroll 2	25445 (5%)	32606 (3%)	290 (11%)	28 (11%)
4 units/unroll 2	50029 (11%)	64064 (7%)	580 (23%)	56 (22%)
8 units/unroll 2	99325 (21%)	127236 (14%)	1160 (45%)	112 (44%)

that over-optimizing code can cause side effects like excessive memory access, causing performance to decrease when inserting more optimizations. Finally, no improvement was observed when marking parameters as restrict in the matrix multiplication kernel. It is possible that the compiler is automatically inferring the absence of aliasing between the pointers and optimizing the code under the hood. A more thorough experimental analysis is needed to characterize program performance when applying this optimization.

5 Conclusion

The challenge of increasing performance-per-watt in high-performance computing systems led engineers to adopt reconfigurable accelerators. However, most programmers are unfamiliar with this class of devices, therefore the task of optimizing applications for FPGAs is not trivial. One of the reasons for this difficulty is the lack of performance portability in OpenCL kernels. In other words, a kernel written for GPUs may show low performance when executed on FPGAs. In that sense, this work compiles and explains a collection of code optimizations available in the literature that can benefit code targeted at reconfigurable accelerators. Besides that, this work contribute with a summary of the current stat of research about FPGAs in high-performance computing and also quantifies the performance gain of some of these optimizations. This work provides the first step towards a full performance characterization for FPGAs that will later be used to build compiler-based tools for automatic optimization.

5.1 Future Work

As a next step we envision a directive-based solution for executing high-performance applications on FPGAs. The solution, based on the Clang/LLVM compiler infrastructure, should take an OpenMP code with target directives, optimize the target region for FPGA execution and finally offload this code to a cloud of reconfigurable accelerators.

A general overview of the approach can be visualized in Figure 22. Starting with the original OpenMP source code, the compiler extracts the target region and apply transformations to it, such as described in Section 3.1. These transformations could be applied in the source level using Clang's LibTooling (39) library. The library allows code

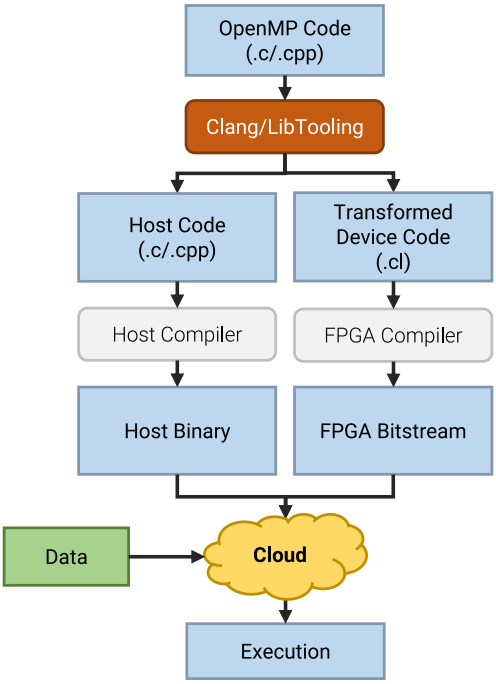


Figure 22 – Our proposed compilation flow that takes OpenMP annotated code, perform source-to-source transformations and generates the host and devices sources that can finally be compile separately using standard compiler tools.

to be inserted, removed or modified at a particular location in the Abstract Syntax Tree (AST). The tree will later be translated back to source code as an OpenCL kernel. On the host side, the compiler will insert runtime calls to the OpenMP target library the same way it is done for other types of accelerators and then continue until an executable binary is produced. Finally, triggering the execution locally will upload the code and data to the cloud, remotely trigger execution and, at the end, transfer the output back to the local machine. This process shall be carried seamlessly, requiring little configuration from the user.

References

1 Intel Corporation. *Intel(R) Arria(R) 10 Device Overview*. [S.l.], 2018.

2 Khronos Group. *Open Computing Language (OpenCL)*. 2019. [Online]. Available: <https://www.khronos.org/opencl/>. (Acessed Feb. 15, 2019).

3 LEE, S.; KIM, J.; VETTER, J. S. OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. [S.l.: s.n.], 2016. p. 544–554. ISSN 1530-2075.

4 Intel Corporation. *Intel(R) FPGA SDK for OpenCL(TM) Pro Edition: Best Practices Guide*. [S.l.], 2018.

5 ZOHOURI, H. R.; MARUYAMA, N.; SMITH, A.; MATSUDA, M.; MATSUOKA, S. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. [S.l.: s.n.], 2016. p. 409–420. ISSN 2167-4337.

6 LLOYD, T.; CHIKIN, A.; OCHOA, E.; ALI, K.; AMARAL, J. N. A case for better integration of host and target compilation when using OpenCL for FPGAs. In: *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*. [S.l.: s.n.], 2017. p. 1–9.

7 MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, p. 56–59, abr. 1965.

8 DENNARD, R. H.; GAENSSLEN, F. H.; RIDEOUT, V. L.; BASSOUS, E.; LEBLANC, A. R. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, v. 9, n. 5, p. 256–268, Oct 1974. ISSN 0018-9200.

9 HENNESSY, J.; PATTERSON, D. *Computer architecture: A Quantitative Approach*. Cambridge, MA: Morgan Kaufmann Publishers, 2019. ISBN 0128119055.

10 HENNESSY, J. L.; PATTERSON, D. A. A new golden age for computer architecture. *Commun. ACM*, ACM, New York, NY, USA, v. 62, n. 2, p. 48–60, jan. 2019. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/3282307>.

11 U.S. Department of Energy. *The Opportunities and Challenges of Exascale Computing*. [S.l.], 2010.

12 KOGGE, P.; BERGMAN, K.; BORKAR, S.; CAMPBELL, D.; CARLSON, W.; DALLY, W.; DENNEAU, M.; FRANZON, P.; HARROD, W.; HILLER, J.; KARP, S.; KECKLER, S.; KLEIN, D.; LUCAS, R.; RICHARDS, M.; SCARPELLI, A.; SCOTT, S.; SNAVELY, A.; STERLING, T.; WILLIAMS, R. S.; YELICK, K.; BERGMAN, K.; BORKAR, S.; CAMPBELL, D.; CARLSON, W.; DALLY, W.; DENNEAU, M.; FRANZON, P.; HARROD, W.; HILLER, J.; KECKLER, S.; KLEIN, D.; KOGGE, P.; WILLIAMS, R. S.; YELICK, K. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. [S.l.], 2008.

13 DONGARRA, J.; BECKMAN, P.; MOORE, T.; AERTS, P.; ALOISIO, G.; ANDRE, J.-C.; BARKAI, D.; BERTHOU, J.-Y.; BOKU, T.; BRAUNSCHWEIG, B.; CAPPELLO, F.; CHAPMAN, B.; CHI, X.; CHOUDHARY, A.; DOSANJH, S.; DUNNING, T.; FIORE, S.; GEIST, A.; GROPP, B.; HARRISON, R.; HERELD, M.; HEROUX, M.; HOISIE, A.; HOTTA, K.; JIN, Z.; ISHIKAWA, Y.; JOHNSON, F.; KALE, S.; KENWAY, R.; KEYES, D.; KRAMER, B.; LABARTA, J.; LICHNEWSKY, A.; LIPPERT, T.; LUCAS, B.; MACCABE, B.; MATSUOKA, S.; MESSINA, P.; MICHIELSE, P.; MOHR, B.; MUELLER, M. S.; NAGEL, W. E.; NAKASHIMA, H.; PAPKA, M. E.; REED, D.; SATO, M.; SEIDEL, E.; SHALF, J.; SKINNER, D.; SNIR, M.; STERLING, T.; STEVENS, R.; STREITZ, F.; SUGAR, B.; SUMIMOTO, S.; TANG, W.; TAYLOR, J.; THAKUR, R.; TREFETHEN, A.; VALERO, M.; STEEN, A. V. D.; VETTER, J.; WILLIAMS, P.; WISNIEWSKI, R.; YELICK, K. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 25, n. 1, p. 3–60, fev. 2011. ISSN 1094-3420. Disponível em: <http://dx.doi.org/10.1177/1094342010391989>.

14 RASMUSSEN, N. *Determining Total Cost of Ownership for Data Center and Network Room Infrastructure*. [S.l.], 2011.

15 Amazon Web Services. *Amazon EC2 F1 Instances*. 2019. [Online]. Available: <https://aws.amazon.com/ec2/instance-types/f1/>. (Acessed Feb. 11, 2019).

16 BARR, J. *EC2 F1 Instances with FPGAs – Now Generally Available*. 2017. [Online]. Available: <https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/>. (Acessed Feb. 11, 2019).

17 FOWERS, J.; OVTCHAROV, K.; PAPAMICHAEL, M.; MASSENGILL, T.; LIU, M.; LO, D.; ALKALAY, S.; HASELMAN, M.; ADAMS, L.; GHANDI, M.; HEIL, S.; PATEL, P.; SAPEK, A.; WEISZ, G.; WOODS, L.; LANKA, S.; REINHARDT, S. K.; CAULFIELD, A. M.; CHUNG, E. S.; BURGER, D. A configurable cloud-scale dnn processor for real-time ai. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. Piscataway, NJ, USA: IEEE Press, 2018. (ISCA '18), p. 1–14. ISBN 978-1-5386-5984-7. Disponível em: <https://doi.org/10.1109/ISCA.2018.00012>.

18 BACON, D.; RABBAH, R.; SHUKLA, S. FPGA programming for the masses. *Queue*, ACM, New York, NY, USA, v. 11, n. 2, p. 40:40–40:52, fev. 2013. ISSN 1542-7730. Disponível em: <http://doi.acm.org/10.1145/2436696.2443836>.

19 Intel Corporation. *Intel Acquisition of Altera*. 2015. [Online]. Available: <https://newsroom.intel.com/press-kits/intel-acquisition-of-altera/>. (Acessed Feb. 10, 2019).

20 TRIMBERGER, S. M. S. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *IEEE Solid-State Circuits Magazine*, v. 10, n. 2, p. 16–29, Spring 2018. ISSN 1943-0582.

21 VENUGOPAL, N.; MANIMEGALAI, R. Survey on fpga routing techniques. *International Journal on Computer Science and Engineering*, Engg Journals Publications, v. 4, n. 7, p. 1304, 2012.

22 PRADO, D. F. G. *Tutorial on FPGA Routing*. [S.l.], 2006.

23 GUDISE, V. G.; VENAYAGAMOORTHY, G. K. Fpga placement and routing using particle swarm optimization. In: *IEEE Computer Society Annual Symposium on VLSI*. [S.l.: s.n.], 2004. p. 307–308.

24 KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*, American Association for the Advancement of Science, v. 220, n. 4598, p. 671–680, 1983. ISSN 0036-8075. Disponível em: <http://science.sciencemag.org/content/220/4598/671>.

25 KARYPIS, G.; AGGARWAL, R.; KUMAR, V.; SHEKHAR, S. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 7, n. 1, p. 69–79, March 1999. ISSN 1063-8210.

26 LU, J.; CHEN, P.; CHANG, C.-C.; SHA, L.; HUANG, D. J.-H.; TENG, C.-C.; CHENG, C.-K. ePlace: Electrostatics based placement using nesterov's method. In: *Proceedings of the 51st Annual Design Automation Conference*. New York, NY, USA: ACM, 2014. (DAC '14), p. 121:1–121:6. ISBN 978-1-4503-2730-5. Disponível em: <http://doi.acm.org/10.1145/2593069.2593133>.

27 MARTIN, G.; SMITH, G. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, v. 26, n. 4, p. 18–25, July 2009. ISSN 0740-7475.

28 NANE, R.; SIMA, V.; PILATO, C.; CHOI, J.; FORT, B.; CANIS, A.; CHEN, Y. T.; HSIAO, H.; BROWN, S.; FERRANDI, F.; ANDERSON, J.; BERTELS, K. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 35, n. 10, p. 1591–1604, Oct 2016. ISSN 0278-0070.

29 Intel Corporation. *Intel(R) FPGA SDK for OpenCL(TM) Pro Edition: Programming Guide*. [S.l.], 2018.

30 AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

31 Xilinx Corporation. *SDAccel Programmers Guide*. [S.l.], 2019.

32 Xilinx Corporation. *SDAccel Environment Profiling and Optimization Guide*. [S.l.], 2019.

33 LAMBERT, J.; LEE, S.; KIM, J.; VETTER, J. S.; MALONY, A. D. Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In: *Proceedings of the 2018 International Conference on Supercomputing*. New York, NY, USA: ACM, 2018. (ICS '18), p. 160–171. ISBN 978-1-4503-5783-8. Disponível em: <http://doi.acm.org/10.1145/3205289.3205324>.

34 CHE, S.; BOYER, M.; MENG, J.; TARJAN, D.; SHEAFFER, J. W.; LEE, S.; SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. [S.l.: s.n.], 2009. p. 44–54.

35 FEIST, T. *Vivado Design Suite*. [S.l.], 2012.

36 OpenACC. *OpenACC: Directives for Accelerators*. 2019. [Online]. Available: <https://www.openacc.org/>. (Acessed Feb. 15, 2019).

37 LEE, S.; VETTER, J. S. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. New York, NY, USA: ACM, 2014. (HPDC '14), p. 115–120. ISBN 978-1-4503-2749-7. Disponível em: <http://doi.acm.org/10.1145/2600212.2600704>.

38 Intel Corporation. *Intel(R) Stratix(R) V Device Overview*. [S.l.], 2015.

39 Clang. *LibTooling – Clang Documentation*. 2019. [Online] Available: <https://clang.llvm.org/docs/LibTooling.html>. (Accessed Feb. 17, 2019).