Exploiting Thread-Level Parallelism on Reconfigurable Architectures: a Cross-Layer Approach

A Dissertation Presented
by
Amir Momeni
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

May 2017
This dissertation is dedicated to my brilliant and outrageously loving and supportive wife, Mahsa, our sweet little girl, Nika, and to my always encouraging, ever faithful parents, Hamidreza and Mansoureh.
Contents

List of Figures iv
List of Tables vi
List of Acronyms vii
Acknowledgments ix
Abstract of the Dissertation x

1 Introduction 1
  1.1 Motivation 3
  1.2 contributions 5
  1.3 Thesis Outline 6

2 Background 7
  2.1 OpenCL Execution on GPUs 8
  2.2 OpenCL Execution on FPGAs 10

3 Source Optimization Approach 14
  3.1 Related Work 15
  3.2 Case Studies 16
    3.2.1 MeanShift Object Tracking (MSOT) 16
    3.2.2 ODVF 17
    3.2.3 AFIM 18
  3.3 OpenCL Pipes 19
    3.3.1 Background 20
    3.3.2 Kernel Pipelining Methods 21
    3.3.3 2D Communication Wrapper 25
    3.3.4 Experimental Results 26
  3.4 Parallelism Granularity 31
    3.4.1 Serial Implementation 31
    3.4.2 Parallel Implementation 32
    3.4.3 Experimental Results 34
3.5 Parallelism Type
3.5.1 Spatial Parallelism Semantic
3.5.2 Temporal Parallelism Semantic
3.5.3 Experimental Evaluation
3.5.4 Discussion
3.6 Summary

4 Synthesis Optimization Approach
4.1 Related Work
4.2 Hardware Thread Reordering
4.2.1 Background and Motivation
4.2.2 Hardware Thread Reordering
4.2.3 Optimizations Methods
4.2.4 Implementation Method
4.2.5 Experimental Results
4.3 Summary

5 Architectural Optimization Approach
5.1 Background
5.1.1 Multi2sim
5.1.2 MIAOW
5.2 FP-GPU High Level Architecture
5.3 FP-GPU CU Implementation
5.4 Evaluation
5.4.1 Experimental Setup
5.4.2 Performance Comparison
5.4.3 Area Comparison
5.5 Discussion
5.6 Summary

6 Conclusions and Future Work
6.1 Contributions of this Thesis
6.1.1 Source-level optimization
6.1.2 Synthesis optimization
6.1.3 Architectural optimization
6.2 Directions for Future Work

Bibliography
# List of Figures

1.1 various types of parallel processors ......................................... 2

2.1 OpenCL Platform Model ..................................................... 8
2.2 OpenCL Execution Model .................................................... 9
2.3 AMD Radeon HD 7970 GPU with 32 Compute-Units .................... 10
2.4 Altera OpenCL compilation framework .................................. 11
2.5 A generic synthesized architecture for OpenCL kernels on FPGAs. 12

3.1 MSOT algorithm .............................................................. 17
3.2 ODVF case study ............................................................. 18
3.3 Inter-kernel communication using OpenCL Pipes ...................... 21
3.4 Sequential vs. concurrent kernel execution ......................... 22
3.5 Kernel synchronization ..................................................... 24
3.6 Kernel synchronization using control signals for data transfer through Pipes 25
3.7 Dimension transform module method ................................... 25
3.8 Kernel synchronization using the dimension transform module .... 26
3.9 Dimension Transfer Module and 2d Kernel algorithms .............. 27
3.10 Speed-up and performance ................................................. 29
3.11 Number of accesses to different types of memory ................... 30
3.12 Resource Utilization ....................................................... 31
3.13 Parallelism granularity in Mean-shift ................................ 33
3.14 WLP speed-up on GPU and FPGA ...................................... 36
3.15 Speedup of the hybrid approach on a GPU ......................... 36
3.16 Homogeneous approach on FPGA ..................................... 38
3.17 OpenCL kernel and synthesized data-path ........................... 39
3.18 OpenCL kernel and synthesized data-path in CU replication ....... 40
3.19 An OpenCL kernel and synthesized data-path using data-path replication 42
3.20 An OpenCL kernel and synthesized data-path applying partial data-path replication 43
3.21 Exploiting temporal parallelism for pipelined execution of multiple kernels 45
3.22 OpenCL kernel and synthesized data-path sub-kernel temporal parallelism 46
3.23 OpenCL kernel and synthesized data-path exploiting sub-kernel temporal parallelism with P-DP replication 48
3.24 Baseline Implementations ............................................... 50
3.25 DP replication impact on the AFIM application: 52
3.26 CU replication impact on the ODVF and MSOT applications: 53
3.27 The impact of temporal parallelism on our case studies: 54
3.28 The impact of P-DP replication for the AFIM application: 55
3.29 The impact of P-DP replication for the MSOT application: 57

4.1 SPMV OpenCL kernel: 63
4.2 SPMV LLVM: 64
4.3 Generated data-path for SPMV kernel: 65
4.4 Pipeline timing diagram of the SPMV datapath: 65
4.5 Context variables per pipeline stages of SPMV: 66
4.6 Out-of-Order execution in SPMV kernel: 67
4.7 Extended Pipeline stage for HTR approach: 67
4.8 Generated HTR-enhanced datapath for SPMV kernel: 68
4.9 Extended pipeline stage with stall signal: 68
4.10 Memory request handler: 71
4.11 HTR implementation process: 73
4.12 Speed-up: 74
4.13 Memory Bandwidth Utilization: 76
4.14 Type and number of stalls: 76
4.15 Logic Utilization: 77
4.16 Register Utilization: 77

5.1 Four independent phases of multi2sim’s simulation paradigm: 81
5.2 MIAOW compute unit block diagram and its submodules: 82
5.3 FP-GPU high level architecture: 83
5.4 Binary Search OpenCL kernel: 85
5.5 The pipeline implementation of the Binary Search OpenCL kernel: 86
5.6 The Load/Store Unit: 88
5.7 FP-GPU and SI GPU performance comparison for five benchmarks: 92
5.8 FP-GPU speed-up for five benchmarks: 93
5.9 FP-GPU and SI GPU area comparison: 94
5.10 FP-GPU performance per area improvement over SI GPU for five benchmarks: 95
5.11 The OpenCL Barrier API and its implementation in FP-GPU: 97
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Details of the implemented designs and associated features.</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>System characteristics used in this study.</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Serial execution of MSOT</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>System characteristics</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>NLP speedup on a GPU and FPGA.</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>GPU and FPGA characteristics comparison</td>
<td>79</td>
</tr>
<tr>
<td>5.2</td>
<td>Cache hierarchy configuration</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>Xilinx Virtex7 XC7VX485T FPGA device specification</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Number of instructions versus number of pipeline stages</td>
<td>90</td>
</tr>
</tbody>
</table>
List of Acronyms

FPGA  Field Programmable Gate Array.
CPU   Central Processing Unit.
GPU   Graphics Processing Unit.
ALU   Arithmetic and Logic Unit.
CU    Compute Unit.
LSU   Load/Store Unit.
PE    Processing Element.
DP    Data-path.
SIMT  Single Instruction Multiple Thread.
SIMD  Single Instruction Multiple Data.
SI    Southern Island.
HDL   Hardware Description Language.
HLS   High-Level Synthesis.
RTL   Register Transfer Language.
VPI   Verilog Procedural Interface.
LUT   Lookup Table.
DSP   Digital Signal Processing.
ALM   Adaptive Logic Module.
FPS   Frame per Second.
SQE   Sequential.
PPE   Partially Pipelined.
SDT  Synchronized Data Transfer.
DTM  Dimension Transfer Module.
CST  Control Signal Transfer.
OLP  Object Level Parallelism.
NLP  Neighbor Level Parallelism.
WLP  Window Level Parallelism.
ILP  Instruction Level Parallelism.
HTR  Hardware Thread Reordering.
MSOT Mean-shift Object Tracking.
ODVF Object Detection Vision Flow.
AFIM Apriori Frequent Itemset mining.
SMT  Smoothing.
MOG  Mixture of Gaussians.
ERO  Erosion.
DIL  Dilation.
SPMV Sparse Matrix Vector Product.
CONV Convolution.
KM  K-Means.
BFS  Breath-First Search.
BS  Binary Search.
VEC  Vector add.
MT  Matrix Transpose.
Acknowledgments

Here I would like to express my sincere gratitude to my advisor Prof. David Kaeli for the continuous support of my PhD study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Gunar Schirner, and Prof. Rafael Ubal, for their encouragement, and insightful comments.

Finally, I would like to thank my wife, Mahsa. She was always there cheering me up and stood by me through the good times and bad.
Abstract of the Dissertation

Exploiting Thread-Level Parallelism on Reconfigurable Architectures: a Cross-Layer Approach

by

Amir Momeni
Doctor of Philosophy in Electrical and Computer Engineering
Northeastern University, May 2017

David Kaeli, Adviser

Field Programmable Gate Arrays (FPGAs) are one major class of architectures commonly used in parallel computing systems. FPGAs provide a massive number (i.e., millions) of programmable logic blocks and I/O cells, as well as programmable interconnects, which can be configured for a particular application. This reconfigurable architecture is flexible and power efficient, and potentially, provides better floating-point operations per watt rates versus general purpose architectures, such as CPUs and GPUs. However, programming an FPGA can be challenging and time-consuming, requiring hardware description language (HDL) experience and digital design expertise. High-level synthesis (HLS) tools have been designed to ease the FPGA programming task by generating HDL (e.g., Verilog or VHDL) codes from high-level languages (e.g., C/C++, OpenCL). In particular, there have been recent developments in OpenCL-based HLS tools (OpenCL-HLS) to enable programmers to construct a customized data-path that can best match a parallel application, relieving the programmer of many implementation details.

Given the availability of OpenCL-HLS tools for FPGAs creates many new opportunities, as well presents new challenges, in order to fully utilize these new capabilities. The primary challenge lies in the difference between the OpenCL parallelism semantics and parallel execution model on FPGA devices. OpenCL is primarily developed for GPU devices, which have many spatially-parallel cores. We need to explore new classes of optimization in order to fully leverage OpenCL execution on FPGAs.

This thesis explores and addresses OpenCL-HLS challenges using three different approaches. In the first approach we consider source-level optimization, where we evaluate the impact of OpenCL source-level decisions on the resulting data-path and FPGA execution efficiency. Our aim is to analyze the correlation between OpenCL parallelism semantics and parallel execution on
FPGA devices. We want to be able to guide OpenCL programmers to develop optimized code on an FPGA. We study the impact of different grains (fine and coarse-grained), and forms of parallelism (spatial and temporal), exposed by OpenCL on the generated data-path. We also study the efficiency of the OpenCL Pipe semantic when targeting an FPGA.

In the second approach called synthesis optimization, we introduce novel optimization techniques for synthesis of OpenCL kernels targeted for FPGA devices. We propose a Hardware Thread Reordering (HTR) technique to improve the performance of irregular kernels. The goal is to guide OpenCL-HLS tool developers to design a more efficient data-path for a given OpenCL kernel. Using our HTR technique, we achieve up to a 11X speed-up, with less than a 2X increase in resource utilization.

In our third approach called architectural approach, we propose a novel device named an FP-GPU (field-programmable GPU), a new class of architecture that utilizes the benefits of both GPU and FPGA architectures. FP-GPU utilizes the GPU memory hierarchy, but introduces a novel thread switching mechanism, which helps to hide long memory latencies. The FP-GPU device includes reconfigurable fabric that can serve as an application-specific compute unit, maximizing the efficiency of OpenCL kernel execution. Our evaluation of FP-GPU finds that we can achieve up to a 4x speed-up, while utilizing 88% less resources as compared to a general-purpose GPU.
Chapter 1

Introduction

In the past, microprocessor designers have always been able to scale the frequency of their processors to increase performance. Over the past decade, power and thermal limits of CMOS technology have posed real challenges and led the designers to place multiple processors on a chip, abandoning frequency scaling. Since then, the number of cores on a single chip and the processing capabilities of parallel computing systems have increased dramatically [19, 24, 22, 46, 14].

Parallel computing systems consist of multiple processing units (homogeneous or heterogeneous), connected via an interconnection network. They perform computation in a divide-and-conquer fashion, where a host processor distributes the computation across multiple devices. Each device is a parallel processor performing a part of the total computation. The host also manages the data transfer across the devices, gathers the result from each device as needed, and generates the final output.

Today’s parallel processors vary in terms of parallelism capability. Figure 1.1 compares CPUs, GPUs, and FPGAs, these three popular parallel processing options available today. Contemporary CPUs contain multiple cores (see Figure 1.1a), with each core equipped with powerful ALUs, and advanced branch prediction mechanisms to execute heavyweight threads with sophisticated flow control. Executing heavyweight threads makes the context switching process very slow and expensive in CPU cores. The CPU architecture takes advantage of large caches to hide memory latency. CPU cores are designed to minimize the execution latency of a single thread. This design style is referred to as latency-oriented design and is suitable for programs with task-level parallelism [35].

In contrast to a CPU, a GPU contains a massive number of cores to execute many lightweight threads in a SIMT (Single Instruction Multiple Threads) fashion. On a GPU, threads share the same
(a) Latency-oriented CPU architecture with 4 cores. (b) Throughput-oriented GPU architecture with a massive number of cores. (c) FPGA architecture with a massive number of programmable logic cells.

Figure 1.1: various types of parallel processors

control logic and execute the same instruction. Since the threads share the same control logic, complex control flow and branches can result into thread divergence, and thus, degrade performance. GPU cores, however, are designed to reduce memory latency by switching between blocks of threads if one block must wait for a long-latency memory access. In this design style, an individual thread potentially takes much longer time to execute, but the total execution throughput of a large number of threads is maximized. This design style is referred to as throughput-oriented and is suitable for executing massive thread-level parallelism (see Figure 1.1b) [33].

Both CPUs and GPUs are general-purpose processors with fixed ALUs. FPGAs on the other hand, are programmable logic devices that can be configured for a particular application. Fig-
CHAPTER 1. INTRODUCTION

Figure 1.1c shows an FPGA architecture with a massive number of logic blocks, and I/O cells, as well as interconnection resources. The logic blocks can be configured to implement merely simple gates or complex combinational functions. They may also include flip-flops or more sophisticated memory elements. The programmer can utilize the logic and interconnection resources to implement a parallel application. Based on the application, the programmer can expose parallelism by creating a deeply pipelined processing unit (temporal parallelism) or multiple smaller processing units for massive concurrent thread execution (spatial parallelism) [32].

In comparison to a CPU and a GPU, a FPGA can provide more flexibility and better power efficiency. The drawback, however, is the complicated and time-consuming programming process. The development of an application for FPGA can take a month, while the same application can be developed for a CPU or GPU in a few days. High-level synthesis (HLS) tools have been designed to ease the FPGA programming overhead by generating HDL (e.g. Verilog or VHDL) codes from high-level languages (e.g. C/C++, OpenCL). With the availability of HLS tools, FPGAs have become a more attractive architecture for high-performance computing. In particular, there have been recent developments in OpenCL-HLS by the two major FPGA companies (Altera and Xilinx) [1, 2]. OpenCL-HLS enables parallel programmers to construct a customized data-path that can best match an application, without getting drowned by implementation details. Furthermore, OpenCL simplifies the task of integrating FPGAs into future heterogeneous platforms. An application developed in OpenCL can better guide synthesis tool by explicitly exposing parallelism. Availability of OpenCL-HLS tools for FPGAs has raised many new challenges which need to be well understood and addressed. In the following, we explain these challenges and our motivation for this thesis.

1.1 Motivation

Despite their significant potential, OpenCL-HLS tools introduce a set of new design challenges for both parallel application developers and synthesis tool developers. The challenges mainly stem from the fundamental architectural differences between GPUs and FPGAs. GPUs are throughput-oriented machines relying on concurrent execution of massively parallel threads on many cores (spatial parallelism). In contrast, an FPGA’s efficiency stems from a customized data-path, operation-level parallelism and also the ability to exploit deep pipelining (temporal parallelism). Previous studies have focused on OpenCL tuning for GPUs since these devices have dominated the heterogeneous computing market. Now that FPGAs have become potential targets, FPGA programmers need to assess the impact of source-level and synthesis-time decisions on the
CHAPTER 1. INTRODUCTION

generated architecture. In many cases, programmers have to revisit their source-level design decisions to enable the synthesis tools to generate an efficient data-path for the FPGA. The decisions include choices impacting the type of parallelism (spatial and temporal), the granularity of parallelism (OpenCL work-items), the thread grouping (OpenCL workgroup size), the synchronization semantics across concurrent kernels, and the semantic of host-to-device and device-to-host communication.

Overall, OpenCL support for FPGAs is in its early stages. There has been little prior work that considers the challenges and potential of the OpenCL for FPGAs in any depth. The general trend has been to compare a hand-crafted RTL implementation with an OpenCL-programmed GPU execution [20, 21, 3, 60, 27]. Some recent studies have reported the performance of OpenCL applications targeting FPGAs, using commercially available OpenCL-HLS tools [13, 8, 50, 23, 48, 41, 58]. However, there is a general lack of understanding of the impact of OpenCL decisions can have on the generated architecture and its execution efficiency on FPGAs. There is a demand for new knowledge that can guide both OpenCL programmers and FPGA vendors to fully utilize the potential of OpenCL. New research is required to study and analyze the impact of OpenCL source-level constructs on the generated data-path, and the execution efficiency on FPGAs.

This thesis explores and addresses OpenCL-HLS challenges. To this end, we use three different approaches. In the first approach, called the source optimization approach, we evaluate the impact of OpenCL source-level decisions on the generated data-path and the FPGA execution. Our aim is to analyze the correlation between OpenCL parallelism semantics and parallel execution on FPGA devices to guide OpenCL programmers how best to develop optimized code (FPGA-aware OpenCL codes). We study the impact of different grains (fine and coarse-grained), and types of parallelism (spatial and temporal) exposed by OpenCL on the generated data-path by Altera OpenCL SDK. We also study the efficiency of the OpenCL Pipe semantic on the FPGA execution.

In the second approach, called the synthesis optimization approach, we introduce new techniques to better synthesize the OpenCL kernels for FPGA devices. Our aim is to guide OpenCL-HLS tool developers (e.g. Altera and Xilinx) to design a more efficient data-path for a given OpenCL kernel. We focus our study on irregular kernels where the current approaches by Altera and Xilinx produce some inefficiencies.

In the third approach, called the architectural approach, we propose a novel device, called an FP-GPU (field programmable GPU), a new class of architecture that utilizes the benefits of both GPU and FPGA architectures. The FP-GPU is a GPU-like architecture, adopting the same memory system and compute unit organization. However, instead of assuming general-purpose
CHAPTER 1. INTRODUCTION

All compute units are implemented with programmable logic resources to implement the OpenCL application. To use FP-GPU, the OpenCL program needs to be compiled to RTL, and the RTL code is synthesized and used to program the compute units. The FP-GPU utilizes a traditional GPU memory hierarchy, and leverages a thread switching mechanism similar to a GPU to hide the memory latency. The major difference is that the FP-GPU creates an application-specific data path that can outperform general-purpose GPU compute units.

1.2 contributions

The goal of this dissertation is to develop a novel design methodology and computing fabric that can exploit thread-level parallelism when mapped to reconfigurable architectures. Here, we outline the contributions of this dissertation.

- We have evaluated the potential benefits of leveraging the OpenCL Pipe semantic to accelerate OpenCL applications. We analyze the impact of multiple design factors and application optimizations to improve the performance offered by OpenCL Pipes.

- Focusing on the Meanshift Object Tracking algorithm as a highly challenging compute-intensive vision kernel, we have evaluated various grains of parallelism, from fine to coarse, on both a GPU and a FPGA.

- We analyzed the correlation between OpenCL parallelism semantics and parallel execution on FPGAs. We evaluated the impact of different types of parallelism (spatial and temporal) exposed by OpenCL on the generated data-path by OpenCL-HLS tool.

- We have proposed a novel solution, called Hardware Thread Reordering (HTR), to boost the throughput of the FPGAs when executing irregular kernels processing non-deterministic and runtime-dependent control flow.

- We have proposed a novel architecture, called a Field Programmable GPU (FP-GPU), to execute OpenCL programs more efficiently.

- We implemented the proposed FP-GPU architecture, and compared it with an AMD Southern Islands GPU, evaluating the merits of this new approach in terms of the performance and area.
CHAPTER 1. INTRODUCTION

1.3 Thesis Outline

The outline of this thesis is as follows. Chapter 2 reviews the background needed for this study. It reviews the OpenCL execution model on GPUs and FPGAs and highlights the differences between the two. Chapter 3, explores various source-level decisions such as grain and type of parallelism and their impact on FPGAs. It also explores the impact of OpenCL Pipe semantic as a promising feature to optimize the OpenCL execution on FPGAs. Chapter 4, proposes some techniques to enhance OpenCL synthesis to fit better to the new architecture. Chapter 5 presents our proposed FP-GPU architecture and compares it with a Southern Islands GPU. Finally, in Chapter 6 we present our conclusions, and discuss directions for future work.
Chapter 2

Background

The Open Computing Language (OpenCL) is a heterogeneous programming framework to develop applications that execute across various devices from different vendors \[26,22\]. OpenCL provides a promising semantic to capture the parallel execution of a massive number of threads, especially when all threads perform a fixed routine over a large volume of data. OpenCL supports a wide range of levels of parallelism and efficiently maps to heterogeneous systems containing CPUs, GPUs, FPGAs, and other types of accelerators.

The OpenCL platform model (see Figure 2.1) contains a processor, called *host*, coordinating the execution of the program, as well as one or more accelerators, called *devices*, capable of executing OpenCL C code (called *kernel*). The *host* is usually a x86 CPU, and the devices can be a combination of CPUs, GPUs, and FPGAs. The *host* code executes the serial portions of the program. The host is also responsible for setting up the devices and managing host-to-device and device-to-host communications. The *kernel* code is the parallel portion of the program, which executes on the devices.

Figure 2.2 represents the OpenCL execution model. The unit of parallelism in OpenCL is called a *work-item*. All OpenCL *work-items* execute the same kernel over different data. The total number of *work-items* executing the kernel code is defined by the programmer in the *host* code and is called an *NDRange*. The *NDRange* is an N-dimensional index space of *work-items*, where N is one, two, or three. As shown in Figure 2.2, the *NDRange* is divided into *work-groups*, each of which contains multiple *work-items*. The *NDRange* size (or *global size*), and the *work-group* size (called *local size*), is defined in the host code by the programmer. A block of *work-items* executing on the device simultaneously is called a *wave-front*. The *wave-front* size is architecture dependent and is defined by the device vendor.
In principle, OpenCL aims to provide a universal programming interface across many heterogeneous devices. However, OpenCL initially was developed and ported to GPU platforms to accelerate data-parallel computations. In the past decade, many papers have studied how best to optimize OpenCL applications on the GPU devices [57, 39, 51]. With the recent developments on OpenCL-HLS, these applications can be easily mapped to the FPGA devices with minimal modifications. However, to achieve a comparable performance, the OpenCL programs need to be optimized based on the target platform. To this end, the programmer should be aware of execution differences between a GPU and a FPGA. In the following, we review the OpenCL execution models on a GPU versus a FPGA. The aim is to highlight the key differences that can impact throughput when developing OpenCL kernels for FPGAs.

### 2.1 OpenCL Execution on GPUs

GPUs are many-core devices that can provide high throughput using a massive number of threads (exploiting spatial parallelism). GPUs are able to hide memory latency by switching threads whenever a thread encounters a stall. Figure 2.3 highlights the internal architecture of the AMD Radeon HD 7970 GPU [38]. The basic computational building block of a GPU architecture is the *compute-unit*. The AMD Radeon HD 7970 GPU contains a set of 32 independent compute-units (CU). Each CU is a combination of 4 SIMD (Single Instruction Multiple Data) units for vector processing. Using 16 SIMD lanes, each SIMD unit simultaneously executes a single instruction.
across 16 work-items. In addition to SIMD lanes (vector ALUs), each SIMD unit contains some other private resources, such as instruction buffering, and registers. To achieve area and power efficiency, the other resources in a compute-unit are shared among all SIMD units, such as the instruction fetch unit, decode and schedule unit, as well as the data caches.
CHAPTER 2. BACKGROUND

The dispatcher is the module that schedules workloads on CUs. The dispatcher maps the work-groups to CUs based on a specific scheduling policy. However, the execution unit on each CU is the collection of 64 work-items called a wavefront. The SIMD units within the CUs execute one wavefront at a time. Each SIMD unit has an instruction buffer for 10 wavefronts. Therefore, the whole CU can have 40 wavefronts in flight. The AMD 7970 GPU with 32 CUs can thus execute 1280 wavefronts or 81920 work-items.

To maximize the throughput on a GPU device, the OpenCL programmer should launch as many threads as possible to increase the GPU occupancy. The more threads that are mapped to a CU, the higher the chances of hiding memory latency and achieving higher throughput. The programmer needs to take the wavefront size into account to minimize the thread divergence within the wavefronts. This also helps to achieve a better throughput on the GPU. Next, we review the OpenCL execution on FPGAs.

![AMD Radeon HD 7970 GPU with 32 Compute-Units.](image)

Figure 2.3: AMD Radeon HD 7970 GPU with 32 Compute-Units.

2.2 OpenCL Execution on FPGAs

While GPUs offer massively parallel fixed ALUs, the reconfigurable nature of an FPGA allows construction of a customized data-path. A customized data-path can optimize thread execution by removing instruction-fetch, streamlining the execution. To increase the throughput, the generated data-path can be deeply pipelined. Deep pipelining enables FPGAs to utilize the temporal parallelism across many hardware threads while sharing the same data-path.
CHAPTER 2. BACKGROUND

Previous studies have proposed OpenCL-HLS tools to execute OpenCL programs on FPGAs \cite{1,2,33,45}. In our experiments we use ALtera OpenCL SDK, a widely used commercial OpenCL-HLS tool \cite{1}. Figure 2.4 represents the flow of the Altera OpenCL compilation framework. The input is a host program written in C, as well as a set of kernels written in OpenCL-C language. The host program is compiled using a C/C++ compiler, and is executed on the CPU. The kernels are compiled into a data-path to be executed on the FPGA. To compile the OpenCL kernels, the Altera OpenCL SDK starts with a C parser to generate an intermediate representation (LLVM IR) for each kernel \cite{36}. Next, the LLVM IR is optimized for the target FPGA device. The optimized LLVM IR is then translated into a Control-Data Flow Graph (CDFG). Another optimization pass is performed on the CDFG to improve the performance and area. Finally, the RTL generation step produces Verilog code for the given kernel. The Altera OpenCL compilation flow is presented in \cite{17} in more detail.

![Figure 2.4: Altera OpenCL compilation framework.](image)

In principle, the OpenCL-HLS tool can expose various types of parallelism for a given OpenCL kernel. Temporal parallelism can be exposed by creating *Compute Unit* (CU) with a deep pipelined data-path for a kernel. Also, the created CU can be replicated multiple times to expose spatial Parallelism. Figure 2.5 shows a generic synthesized architecture for OpenCL kernels on
CHAPTER 2. BACKGROUND

The architecture contains multiple CUs with a shared memory interface and a shared dispatcher. The shared dispatcher assigns OpenCL work-groups across multiple CUs. Each CU internally offers a customized pipelined data-path. When there is no stall in the pipeline, one work-item (thread) enters the pipeline per clock cycle, and one work-item completes its execution and exits the pipeline. The pipelined data-path is designed to execute the work-items in an in-order fashion throughout the pipeline stages. This results in very high data-path utilization, and thus, high program throughput when executing regular OpenCL kernels. The in-order execution, however, might degrade performance in irregular OpenCL kernels with complex flow control. To achieve even higher throughput, the data-path inside each CU can be replicated. The replicated data-path is able to commit multiple work-items per cycle [7, 6].

In contrast to GPUs, OpenCL execution on FPGAs is directly impacted by the synthesized architecture and the resulting data-path. In fact, achieving high throughput on FPGAs requires optimizations at two different levels. First, the programmer is responsible for adjusting the OpenCL code to match the execution semantics of FPGAs effectively. For example, varying the thread granularity may reduce the number of stalls in a pipeline, and improve the performance by increasing the data-path occupancy. For example, increasing the number of CUs might improve the performance in compute-bound kernels. However, the same strategy might degrade the performance in memory-bound kernels, where the CUs might compete for the shared memory interface. The programming
CHAPTER 2. BACKGROUND

decisions and strategies, and their effects on the generated data-path, and the resulting execution efficiency of OpenCL codes run on FPGA devices are explored in Chapter 3.

Second, the synthesis tool developer is responsible for generating an efficient data-path for the given OpenCL kernel. The synthesis optimization decisions will impact the OpenCL execution on FPGAs dramatically. In Chapter 4, we study the Hardware Thread Reordering as a synthesis optimization method to improve the execution of irregular OpenCL kernels.
Chapter 3

Source Optimization Approach

The ability to run OpenCL across many heterogeneous nodes (FPGAs, GPUs, CPUs) opens up significant design choices, as well creates new design challenges for system designers and application programmers. In principle, OpenCL offers a universal description of an application, independent from the target architecture. But get the best performance, some customization should take place at the source code level that considers the actual target platform. This challenge is more pronounced when we consider platforms that include FPGAs.

Despite the significant potential, OpenCL for FPGAs introduces a set of new design challenges for both parallel application developers and OpenCL synthesis tool developers. These challenges stem mainly from the fundamental architectural differences between GPUs and FPGAs. GPUs are throughput-oriented machines relying on concurrent execution of massively parallel threads on many cores (spatial parallelism). In contrast, an FPGA’s efficiency depends upon a customized data-path, operation-level parallelism and also the ability to exploit deep pipelining (temporal parallelism). Previous studies have focused on OpenCL tuning for GPUs, since these devices have dominated the heterogeneous computing market. Now that FPGAs have become potential targets, FPGA programmers have started to assess the impact of source-level decisions on the generated architecture. In many cases, programmers have to revisit their source-level design decisions to enable the synthesis tools to generate an efficient data-path for the FPGA. The decisions include choices impacting the type of parallelism (spatial and temporal), the granularity of parallelism (OpenCL work-items), thread grouping (OpenCL workgroup size), synchronization semantics across concurrent kernels, and semantics of host-to-device and device-to-host communication.

This chapter analyzes the impact of source-level decisions, applied in OpenCL, on the
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

FPGA’s execution efficiency. Our aim is to analyze the correlation between OpenCL parallelism semantics and parallel execution on FPGA devices to guide OpenCL programmers to develop FPGA-optimized code. First, we evaluate the impact of the OpenCL Pipe semantic, and explore how we can leverage it for FPGA compute efficiency. Next, we study various levels of parallelism granularities, and compare an FPGA with a GPU device to find the most suitable grains of parallelism on each device. Finally, We evaluate the impact of different types of parallelism (spatial and temporal) exposed by OpenCL on the generated data-path for FPGA.

3.1 Related Work

Optimizations implemented at the OpenCL programming layer have not been studied in depth for FPGA devices. Most optimization studies have focused solely on GPU tuning, since these devices have dominated the heterogeneous computing market to date. The general trend has been to compare a hand-crafted FPGA implementation with an OpenCL-programmed GPU execution to evaluate the performance and power efficiency [20, 21, 16, 10, 34]. Furthermore, previous work mainly focuses on embarrassingly parallel applications, ignoring the broader class of irregular applications that possess lower degrees of parallelism, though have plenty of potential for acceleration with the right device.

With the release of OpenCL-HLS tools, recent work has demonstrated the potential of OpenCL for FPGAs [13, 8, 50, 23]. Chen et al. [13] present an OpenCL implementation of fractal compression, an encoding algorithm based on iterated function system (IFS). They compare an FPGA optimized code with CPU and GPU optimized code. They also evaluate the Altera’s SDK for OpenCL by comparing the OpenCL implementation with a hand-coded RTL implementation. Andrade et al. [8] propose an OpenCL implementation of the Fast Fourier Transform Sum-Product Algorithm (FFT-SPA) decoder used in Error-Correcting Codes (ECCs) algorithm. Settle [50] uses Altera’s SDK for OpenCL to implement the Smith Waterman algorithm for DNA, RNA, or protein sequencing in bioinformatics. In this implementation, pipe channels were utilized to communicate between adjacent diagonal and vertical cells. The result showed that a FPGA can significantly outperform a CPU or GPU in terms of both performance and power efficiency. Gautier et al. [23] evaluated the performance, area, and programmability trade-offs of the Altera OpenCL tool based on two prominent algorithms in 3D reconstruction. However, these approaches primarily focus on the performance optimization of OpenCL on FPGAs, and do not show the correlation between OpenCL code and the generated data-path.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Overall, there has not been any in-depth analysis of how OpenCL source-level constructs map to FPGAs. We argue that new research needs to explore the effect of source-level design decisions across a wide range of architectures. The results of this study can help guide an OpenCL developer to better leverage the targeted accelerator. At the same time, there is a need to tackle OpenCL implementations of complex algorithms that possess irregular execution patterns and less obvious parallelism. A representative class of applications include advanced vision algorithms, which are compute-intense kernels mixed levels of parallelism and regularity across parallel threads.

3.2 Case Studies

To carry out the study, we have developed parallel OpenCL codes for three compute-intense applications in computer vision and big data analytic markets. MeanShift Object Tracking (MSOT) and Object Detection Vision Flow (ODVF) are two irregular kernels from the vision market. ODVF consists of four different kernels. It is a good case to study different methods of exposing temporal parallelism across kernels. MSOT is a appropriate algorithm to explore thread-granularity. Different levels of granularity, from coarse to fine, can be exposed in MSOT algorithm. Our third case-study, Apriori Frequent Itemset Mining (AFIM), is an algorithm used for frequent itemset mining on transactional databases. The AFIM kernel calculates the support of the itemsets on a very large amount of data that can be processed in either parallel or pipelined fashion. This makes AFIM another good application to compare the impact of spatial and temporal parallelism on FPGAs. Next, we provide detailed information about each application, followed by an explanation of our experimental setup, including the baseline OpenCL implementation for each application.

3.2.1 MeanShift Object Tracking (MSOT)

The MSOT algorithm was originally proposed by Comaniciu et al. [15] and later became widely used for object tracking due to its high quality and robustness. Figure 3.1 highlights the major steps of the MeanShift algorithm. Breaking it down top-down, the algorithm is divided into two steps: 1) initialization and 2) adaptive tracking. During the initialization step (frame 0 only), a color histogram is calculated per object in the scene (line 3 and line 4). The histogram values are used as the reference model for tracking objects through the remaining sequence of frames. In the adaptive tracking phase (frames 1 to N), MSOT iteratively identifies the new location of the object with respect to the reference histogram. MSOT calculates the current histogram (line 10) and uses
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Figure 3.1: MSOT algorithm

```
Algorithm 1 MSOT

Require: \( F = \{f_0, \ldots, f_N\} \)
1: \text{procedure OBJECT TRACK}
2: \textbf{for} all objects in the scene \textbf{do} \hspace{1cm} \triangleright \text{Initialization}
3: \hspace{1cm} b \leftarrow \text{calculate the bins of } f_0
4: \hspace{1cm} q \leftarrow \text{calculate the target color histogram on } f_0
5: \textbf{end for}
6: \textbf{for} \( i = 1 \text{ to } N \) \textbf{do} \hspace{1cm} \triangleright \text{Adaptive tracking}
7: \hspace{1cm} \textbf{for} all objects in the scene \textbf{do}
8: \hspace{2cm} b \leftarrow \text{calculate the bins of } f_i
9: \hspace{2cm} \textbf{while} \( k < \text{Threshold} \) \textbf{do}
10: \hspace{3cm} p \leftarrow \text{target color histogram on } f_i
11: \hspace{3cm} d \leftarrow \text{distance between } p \text{ and } q
12: \hspace{3cm} (X, Y) \leftarrow \text{calculate shift vector}
13: \hspace{3cm} \text{Update the target position}
14: \hspace{2cm} \textbf{end while}
15: \hspace{1cm} \text{Draw the target window on the frame } f_i
16: \textbf{end for}
17: \textbf{end for}
18: \textbf{end procedure}
```

the Bhattacharyya distance measure to determine the similarity between the current and reference histograms (line 11). Next, the shift-vectors are calculated based on the Bhattacharyya distance, and the object moves one step toward its new location (line 12 and line 13 of the algorithm). Overall, the higher the iteration threshold (i.e., higher Threshold), the higher the similarity matching, and thus the higher the quality. A thorough description of the original MSOT algorithm has been described previously [15].

3.2.2 ODVF

Figure 3.2 shows the flow of our second case study, Object Detection Vision Flow (ODVF), which has been widely used for object detection and tracking [59, 47]. The application consists of 4 kernels operating on pixel streams as follows.

**Pixel smoothing (SMT):** This kernel is a 2D filter implementing a Gaussian pixel density smoothing function [25]. The kernel adjusts the value of each pixel based on the values of neighboring pixels.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

**Mixture of Gaussians (MOG):** MOG is a commonly used machine learning algorithm for subtracting the foreground pixels from the background scene [52]. MOG employs multiple Gaussian distributions to capture the multi-modal background values per pixel. The output of the MOG kernel is a Foreground (FG) mask.

**Erosion (ERO):** Erosion applies a 2D vision filter to calculate the minimum value from the neighbors of a pixel. Erosion operates on the FG mask, removing random FG pixels in the foreground scene.

**Dilation (DIL):** Dilation also applies a 2D vision filter, calculating the maximum value from the neighborhood of each pixel. Dilation is used to fill the inside of an object body in the FG mask.

As highlighted in Figure 3.2, each kernel operates on the output stream of the previous kernel. The streaming data is passed to the next kernel in the pipeline. Smoothing, Erosion, and Dilation are all 2D vision filters. The dimension of the window varies from $3 \times 3$ to $7 \times 7$ or more – a higher resolution frame requires a larger window size. In contrast, MOG operates on independent pixels, and does not consider the pixel’s neighborhood. Although the ODVF kernels can be easily implemented in a pipelined fashion, passing data between kernels is challenging.

3.2.3 AFIM

Our third case study, Apriori [4], is one of the best-known Frequent Itemset Mining (FIM) algorithms. FIM algorithms are used to find the most frequently-occurring itemsets in large-scale transactional databases. For each itemset, the number of transactions containing the itemset divided by the total number of transactions, called support ratio, is used to measure the frequency of the itemset. Starting with $1$-item candidates, AFIM iteratively generates $k+1$-items candidates by merging $k$-items frequent itemsets. This step is called candidate generation. Then, AFIM calculates the support ratio for the generated $k+1$-items candidates (support counting step). The AFIM prunes the
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

$k+1$-items candidates if their support ratio is less than a given minimum support ratio threshold. The remaining itemsets will be used to generate candidates for the next iteration. This procedure continues until no new candidate can be built in the candidate generation step.

3.3 OpenCL Pipes

One key feature of OpenCL 2.0 is the Pipe execution semantic. The Pipe semantic can effectively capture the execution behavior of streaming applications, which comprise multiple kernels working in parallel over changing streams of data. Utilizing the OpenCL Pipe semantic could potentially produce great improvements in application throughput by fully overlapping execution and data exchange between OpenCL kernels. Leveraging Pipes opens up additional opportunities for efficient management of streaming data access, as well as reducing synchronization overhead.

Despite the significant potential, architectural support for OpenCL Pipes is in its nascent stages. GPUs and FPGAs are the two major platforms aiming to support OpenCL Pipes. GPU vendors are still in the process of identifying the architectural challenges and performance benefits when executing pipelined kernels. Overall, the state-of-the-art GPU architectures are not able to fully utilize the execution benefits of pipelined kernels. The on-chip local memories are bound to individual Compute Units (CUs) and are only shared within work-items in a single kernel. As a result, the entire stream of data accesses demanded by multiple pipelined kernels is forwarded to off-chip memory, thereby minimizing the potential for overlapped execution, while increasing the latency and the power consumption.

Compared to GPUs, FPGAs have a number of features that allow them to support the OpenCL Pipe execution semantics. Altera recently announced support for OpenCL 2.0 features, including the Pipe semantic. The FPGA’s reconfigurability simplifies pipeline realization and opens up the door to improve throughput of pipelined kernels. However, OpenCL support for FPGAs, and in particular the Pipe feature, are in their early stages. There has been little prior work that considers the challenges and potential of the Pipes. This motivates us to explore the potential benefits of the OpenCL Pipe semantics for FPGAs. There is little guidance for OpenCL programmers and FPGA vendors to aid the development of pipelined kernels and perform synthesis. Decisions such as granularity and the rate of streaming data sent cross stages of pipelined kernels, as well as placement of the Pipe memories, have not been explored in detail.

This section considers the impact of the OpenCL Pipe semantic, and how we can leverage it for FPGA compute efficiency. We focus our attention on streaming applications. We study
ODVF, which combines four vision kernels (smoothing, MoG background subtraction, erosion, and dilation). Our work compares overall throughput when executing parallel kernels, comparing non-pipelined (i.e., sequential) and pipelined execution. We demonstrate that in order to utilize the potential benefits of pipelined execution, multiple design alternatives need to be explored and optimized to achieve efficient execution. The main bottleneck is present in the memory interface, especially when kernels issue parallel memory accesses concurrently. Through a proper resizing of a kernel’s granularity, as well as an adjustment of the rate and volume of streaming data transfer, pipelined execution achieves far higher throughput. Furthermore, we propose a novel mechanism to effectively capture the behavior of 2-dimensional (2D) vision algorithms in an OpenCL abstraction. The proposed mechanism offers an OpenCL wrapper to efficiently overlap streaming data transfer and vision processing in 2D vision processing; maximizing the benefits of kernel-level pipeline execution on vision applications.

3.3.1 Background

The object detection vision flow described in Figure 3.2 is an example of a tightly-coupled application with multiple compute kernels. These tightly-coupled applications invariably demand a high degree of communication. In such workloads, even though the enqueued OpenCL kernels utilize the same memory space, they will still need to be stopped and restarted often to support synchronization and data exchange. Inter-kernel communication between multiple kernels is challenging to implement, since all communication primitives between OpenCL kernels are built using atomic operations within workgroups of an NDRange.

The OpenCL Pipe provides for well-defined communication and synchronization when concurrent kernels execute in a consumer-producer fashion. An OpenCL Pipe is a typed memory object which maintains data in a first-in-first-out (FIFO) manner. The Pipe object is created and initialized by the host and stores data in form of packets. Access to the Pipe is restricted to the kernels executing on the device (FPGA/GPU) and cannot be updated by the host. Memory transactions on the Pipe object are carried out using OpenCL built-in functions such as read.pipe and write.pipe. Multiple Pipes which have different access permissions can be accessed in the same kernel.

Figure 3.3 illustrates a case where a Pipe object is used for communication between two kernels. The producer kernel writes the tile-id of the data in the Pipe. The tile-id is retrieved by the consumer kernel and is used as an offset into the intermediate data buffer to obtain input data. The Pipe can also be used to pass a reference to the data instead of the entire data object. The producer
and consumer kernels execute concurrently on the device. The state of the data in the Pipe object is maintained until the Pipe object is released by the host. Changes to the state of the Pipe are visible to all kernels accessing the Pipe.

To support the Pipe semantic on a FPGA, Altera has recently released a compiler and tool to build and run OpenCL applications; supporting the OpenCL 1.0 API and the Pipe semantic feature introduced in OpenCL 2.0. This feature is called a *channel* in the Altera OpenCL tool. An AOCL channel is a FIFO buffer which allows kernels to communicate directly with each other, independent of the host processor. The read and write operations in AOCL channels can be either blocking or non-blocking [6]. The blocking read and write operations may cause stalls in the compute pipeline. The stalls occur either when the producer tries to write data into the channel while the channel is full, or when the consumer tries to read from an empty channel. These scenarios occur in the case of having unbalanced producer/consumer kernels. The channel *depth* attribute helps the programmer deal with these situations. The programmer can increase the depth of the channel to guard against getting full when the consumer is slower than the producer.

### 3.3.2 Kernel Pipelining Methods

This section explores pipeline design tradeoffs, including a number of key source-level design decisions that effect the FPGA data-path and execution efficiency. Figure 3.4 shows both sequential and concurrent kernel execution scenarios. In the sequential scenario (highlighted in Figure 3.4a), first the host processor (CPU) writes the input frame into the global memory for the FPGA to use. Then the CPU launches the first kernel (SMT). The SMT kernel performs the pixel smoothing operation on the input frame. Theoretically, when the first pixel of the input frame is
calculated by the SMT kernel, the next kernel (MoG) can start its operation on that pixel. However, since there is no communication mechanism between kernels in this case, the MoG kernel has to wait until the SMT kernel completes its processing. At that moment, the host processor launches the next kernel. This sequence continues until all four kernels are finished. Then, the host CPU reads the result frame and writes the next frame into the global memory for processing.

The sequential scenario (highlighted in Figure 3.4a) has two main inefficiencies. The first is the latency incurred due to individual kernel execution. Second, there are multiple accesses issued to global memory. Each kernel reads in the input data from global memory, and writes back the result into global memory for the other kernels to use. In the concurrent scenario (Figure 3.4b), the CPU writes the input frame into global memory, then launches all kernels at the same time. The SMT kernel reads in the input frame, pixel-by-pixel, and performs the pixel smoothing operation. After computing the new value for each pixel, the SMT kernel writes the result into a Pipe (or channel) for the MoG kernel to use. Then the SMT kernel can start the computation for the next pixel, while the MoG kernel reads in the calculated pixel from the Pipe and begins processing on that pixel. A similar pattern of communication takes place between the MoG and ERO kernels, and also between the ERO and DIL kernels. The DIL kernel, which represents the final stage in the pipeline, writes the result back to the global memory. This pipelined execution continues until all pixels of the input frame have passed through all stages. Then the CPU reads the result and writes the next input frame to the global memory. Figure 3.4b shows how kernel execution is overlapped in this case. Also, the number of off-chip global memory accesses has decreased dramatically compared
to the sequential execution.

Synchronizing the kernels and running them in a pipelined fashion can be a challenging task. This is particularly true for a 2D kernel which needs the value of the pixel, as well as the neighboring pixels. As an example, when the MoG kernel performs background subtraction on the first pixel of the frame \( (p_{0,0}) \), it writes the result into the Pipe for the ERO kernel. The ERO kernel reads that pixel from the Pipe and starts the erosion operation. However, the ERO kernel also needs pixels \( p_{0,1}, p_{1,0}, \) and \( p_{1,1} \) which are the neighbors of pixel \( p_{0,0} \). In this case, we need to provide all neighboring pixels to calculate \( p_{0,0} \) before the ERO kernel can start its processing. In other words, each work-item in the 2D kernel, which is in charge of computing one pixel position of the input frame, reads 9 pixels as the input (assuming a 3\( \times \)3 filter window), while the same work-item outputs only a single pixel value. The disparity in size between the input and the output for the 2D kernels makes the synchronization mechanism challenging. In this section, we introduce synchronization methods for 1D and 2D kernels using OpenCL Pipe.

### 3.3.2.1 Synchronizing Data Transfer using Pipes

To provide synchronization, we use local memory to manage input data for the 2D kernels. The local memory is an on-chip memory shared between all work-items within a work-group. Since the local memory size is limited, we divide the input frame into several blocks. Each block is assigned to a work-group to complete the 2D kernel. The work-groups are executed in a sequential manner. Each work-group fetches a block of the frame from local memory, and performs the computation. The computation consists of two steps. In the first step, each work-item within the work-group reads the pixel value from the pipe, and stores that value in local memory. In the second step, each work-item has access to all pixels in the block, including the neighboring pixels. In this step, each work-item reads all needed pixels from the local memory and performs the actual computation on its pixel position.

Using synchronized data transfer (see Figure 3.5a), the kernels are synchronized in such a way that they can be executed concurrently in a pipelined fashion. We can avoid global memory accesses in the middle stages of the pipeline. However, this method has some drawbacks. First, to guarantee that the first step is completed by all work-items before starting the second step, we need to use barriers between the two steps. Using barriers imposes resource utilization overhead to implement the barrier mechanism. It also imposes a delay due to pipeline stalls. The value for all of the pixels within the block need to be calculated by the producer kernel before the consumer kernel
starts its actual computation (the second step). In other words, the kernels are being executed in a block-level pipeline instead of in a pixel-level pipelined fashion.

Another disadvantage of the current coarse-grained data transfer method is the need to divide frames into blocks based on local memory size limitations. Since the pixels on the block boundaries cannot access all neighboring pixels, the block division imposes a quality loss at the boundaries. Increasing the block size, and therefore decreasing the number of blocks, improves the quality. However, the block size is limited by the size of local memory. Also, increasing the block size increases the number of pipeline stalls, and therefore decreases overall performance.

### 3.3.2.2 A Protocol for Managing Data Transfers using Pipes

Instead of using Pipes for data transfer across kernels, we choose to use global memory. Each work-item in a kernel reads the required pixels from the global memory, performs the computation, and writes the result back into the global memory. We use Pipes as the mechanism to synchronize the kernels. The kernels communicate with each other through the pipes by sending control signals. As an example, consider the MoG and ERO kernels (Figure 3.6). To perform the erosion operation on pixel $p_{1,1}$, the ERO kernel needs pixels $p_{0,0}$, $p_{0,1}$, $p_{0,2}$, $p_{1,0}$, $p_{1,1}$, $p_{1,2}$, $p_{2,0}$, $p_{2,1}$, and $p_{2,2}$. Using control signals, the MoG kernel sends a signal to the ERO kernel when it calculates pixel $p_{2,2}$ and writes it into the global memory. Therefore, the ERO kernel does not have to wait for all pixels to be calculated by the MoG kernel and can start the computation as soon as the necessary neighboring pixels are ready.

Our *control signal transfer* method (Figure 3.5b) provides a finer-grained synchronization mechanism to execute kernels concurrently. Concurrency is maintained at the level of a pixel. There
are no barriers, and so we avoid pipeline stalls and resource wastage. We also maintain vision quality, which can tend to deteriorate due to the use of block division. However, the disadvantage of this method is that the streaming data is transferred through off-chip global memory. Since all kernels are executing concurrently, all kernels need to access the global memory at the same time. This memory contention can impact the performance in memory-bound kernels.

### 3.3.3 2D Communication Wrapper

Next, we describe our proposed OpenCL wrapper to accelerate streaming data communication in 2D vision processing. A Pipe can be used for data transfers between kernels, similar to the synchronized data transfer method. This helps to reduce the number of global memory accesses dramatically. There is no need to divide the frame into blocks, and utilize the local memory to provide data for 2D kernels. Therefore, this method does not suffer any quality loss, and we avoid using barriers to synchronize work-items. This method provides a pixel-level pipeline structure,
similar to the control signal transfer method. It combines the advantages of both synchronized data transfer and control signal transfer methods to increase the throughput of the vision flow.

This method takes advantage of a new transformation called the Dimension Transform Module, which is designed to provide pixel data for 2D kernels (see Figure 3.7). This module has one Pipe as its input to receive input pixels. It has 9 Pipes (assuming a $3 \times 3$ filter size) to provide all needed pixels to the consumer, the next 2D kernel. The output Pipes are labeled as top-left, top-center, top-right, middle-left, middle-center, middle-right, bottom-left, bottom-center, bottom-right. The dimension transform module (see ) reads pixel data from the input Pipe, and writes the pixel in as many output Pipes as needed. Based on the pixel position, the dimension transform module decides if the pixel is the top-left neighbor of another pixel. If it is, the module writes the pixel into the top-left Pipe. The module does the same procedure for the other output Pipes as well. If the input pixel is not a boundary pixel, then it is a neighbor for 9 pixels. Therefore, the module writes to all 9 output Pipes. As an example, the pixel $p_{0,0}$, the top left corner pixel of the frame, is the top-left neighbor of pixel $p_{1,1}$. It also is the top-center, middle-left, and middle-center neighbor of pixels $p_{1,0}$, $p_{0,1}$, and $p_{0,0}$, respectively. Therefore, the dimension transform module writes this pixel into the top-left, top-center, middle-left, and middle-center output Pipes. The consumer kernel is placed after the dimension transform kernel. The consumer kernel (see ) reads the required pixel values from the appropriate Pipes. For example, the pixel $p_{0,0}$, reads the neighboring pixels from middle-center, middle-right, bottom-center, and bottom-right Pipes. The dimension transform module enables 2D kernels to access the data through the Pipes without using local memory. Figure 3.8 shows the kernels used in this method.

3.3.4 Experimental Results

We have implemented the vision flow application using the 3 different methods introduced in previous sections. Our implementation is based on the OpenCL 1.0 standard, the version currently
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Algorithm 1 Dimension Transform Module

1: _kernel void DTM()
2:  tid ← get_global.id()
3:  pixel ← inFrame[tid]
4:  switch tid
5:     case top_left_corner
6:         write_channel(top_left, pixel)
7:         write_channel(top_center, pixel)
8:         write_channel(middle_left, pixel)
9:         write_channel(middle_center, pixel)
10:    case top_boundary
11:        write_channel(top_left, pixel)
12:        write_channel(top_center, pixel)
13:        write_channel(top_right, pixel)
14:        write_channel(middle_left, pixel)
15:        write_channel(middle_center, pixel)
16:        write_channel(middle_right, pixel)
17:        ...  
18:     case non_boundary
19:         write_to_all_channels(pixel)

Algorithm 2 2D kernel

1: _kernel void SMT()
2:  tid ← get_global.id()
3:  switch tid
4:     case top_left_corner
5:         read_channel(middle_center)
6:         read_channel(middle_right)
7:         read_channel(bottom_center)
8:         read_channel(bottom_right)
9:     case top_boundary
10:        read_channel(middle_left)
11:        read_channel(middle_center)
12:        read_channel(middle_right)
13:        read_channel(bottom_left)
14:        read_channel(bottom_center)
15:        read_channel(bottom_right)
16:        ...  
17:     case non_boundary
18:        read_from_all_channels()
19:        perform_computation()

Figure 3.9: Dimension Transfer Module and 2d Kernel algorithms

Table 3.1: Details of the implemented designs and associated features.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ</td>
<td>sequential execution</td>
</tr>
<tr>
<td>PPE</td>
<td>partially pipelined execution</td>
</tr>
<tr>
<td>SDT</td>
<td>synchronizing data transfer method</td>
</tr>
<tr>
<td>CST</td>
<td>control signal transfer method</td>
</tr>
<tr>
<td>DTM</td>
<td>dimension transform module method</td>
</tr>
</tbody>
</table>
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Table 3.2: System characteristics used in this study.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Xeon CPU E5410</td>
</tr>
<tr>
<td>Host clock</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>FPGA Family</td>
<td>Stratix-V</td>
</tr>
<tr>
<td>FPGA Device</td>
<td>5SGXE7N2F45C2</td>
</tr>
<tr>
<td>ALMs</td>
<td>234,720</td>
</tr>
<tr>
<td>Registers</td>
<td>939K</td>
</tr>
<tr>
<td>Block memory bits</td>
<td>52,428,800</td>
</tr>
<tr>
<td>DSP Blocks</td>
<td>256</td>
</tr>
</tbody>
</table>

supported by the Altera tool. Table 3.1 describes 5 different implementations of the vision flow application and the feature used in each implementation. The first implementation (SEQ) is the sequential kernel execution. There are no Pipes between the kernels.

The kernels are designed using NDRanges versus OpenCL Tasks. This allows individual work-items within the NDRange kernel to communicate with work-items in another NDRange-based kernel through an OpenCL Pipe. Each kernel reads the input frame from global memory, performs its computation, and stores the result back to the global memory for the next kernel. This implementation is considered as the baseline. In the second implementation, the MoG kernel (the only 1D kernel in our vision flow), is connected to the producer SMT kernel through a Pipe. This implementation is partially pipelined (PPE). The host processor launches the SMT and MoG kernels, which run concurrently, while the other two kernels are executed sequentially. This implementation evaluates the impact of overlapping 1D kernels only in our sample application. The other 3 implementations have the same Pipe as before in the MoG kernel. They also overlap the 2D kernel execution, as well as leverage the 3 new methods developed. The third implementation (SDT) uses the synchronized data transfer method, while the fourth (CST) and fifth (DTM) implementations use the finer-grained control signal transfer module method and dimension transform module method, respectively.

We have targeted the Altera Stratix-V FPGA as the accelerator architecture. Table 3.2 shows the system parameters in more details. We have also used the Altera SDK for OpenCL v14.0 for compiling and synthesizing the OpenCL code. The experiments are carried out on a sequence of 120 full HD (1080 x 1920) frames of a soccer field.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Figure 3.10: Speed-up and performance.

3.3.4.1 Performance

We use the overall throughput (FPS) of the kernel executions as the performance metric. We also use the Altera SDK for OpenCL (AOCL) profiler to evaluate the performance of various implementations in greater detail. The AOCL profiler uses performance counters to collect kernel performance data, and reports the memory read and write accesses, stalls, and global memory bandwidth efficiency. Stalls refer to the percentage of time that a memory access causes a pipeline stall. The global memory bandwidth efficiency also refers to the percentage of total bytes fetched from global memory that the kernel program uses. Figure 3.10 shows the impact of using Pipes on the performance. Figure 3.11 also represents the number of accesses to different types of memory, as well as the global memory access efficiency. The maximum global memory bandwidth is 25.6 GB/s.

The sequential execution scenario (our baseline) can process 21 frames/second. Using only one Pipe between the SMT and MoG kernels (PPE) increases the performance to 24 frames/second. Increasing the number of Pipes between the kernels, and decreasing the number of global memory accesses, increases the global memory access efficiency in the SDT and DTM implementations (see Figure 3.11). In these cases, we see 2.7X and 2.8X speed-up for SDT and DTM, respectively. We can achieve up to 57 FPS in DTM, which is approaching real-time processing speeds. In the CST implementation, Pipes have been used for synchronization, but not for data transfer. Therefore, the number of global memory accesses is still as high as the SEQ and PPE implementations. The total number of accesses has increased because of the overhead of Pipe accesses. However, the four kernels are executed concurrently in this case, and we see 2X speed-up.
(40 FPS) in the CST implementation. Since the kernels are executed at the same time, the global memory efficiency is lower due to added contention.

### 3.3.4.2 Resource Utilization

The resources available on the Altera Stratix-V FPGA board are presented in Table 3.2. The Adaptive Logic Module (ALM) refers to the basic building block of the Altera FPGA. The ALM can support up to eight inputs and eight outputs. It also contains two combinational logic cells, two or four register logic cells, two dedicated full-adders, a carry chain, a register chain, and a 64-bit LUT mask. The Digital Signal Processing (DSP) block is a feature to support higher bit precision in high-performance DSP applications. The DSP block contains input shift registers to implement digital filtering applications. The DSP can also implement up to eight $9 \times 9$ multipliers, six $12 \times 12$ multipliers, four $18 \times 18$ multipliers, or two $36 \times 36$ multipliers.

Figure 3.12 compares various implementations in terms of resource utilization. Increasing the block memory bit usage in the SDT, CST, and DTM implementations increases the amount of on-chip local memory used. As a result, these designs use local memory more than off-chip global memory. Using the Dimension Transform Module, we decreased the registers usage by 9%. Also the local memory usage is lowered by 5% and 2% when compared to SDT and CST, respectively. Our results show that the DTM implementation uses memory more efficiently than the other implementations.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

3.4 Parallelism Granularity

This section evaluates the impact of parallelism granularity on FPGAs and GPUs. We demonstrate that FPGAs significantly benefit from fine-level parallelism, while GPUs need a combination of coarse- and fine-grained parallelism. Furthermore, a homogeneous FPGA-only solution leads to skyrocketing speedup due to constructing a customized data-path for both parallel and serial portions of algorithm. For this study, we focus on MSOT as a highly challenging compute-intense vision kernel. We propose a new vertical classification for selecting the grain of parallelism for MSOT algorithm. We start with a serial implementation of the MSOT algorithm as our baseline. Next, we evaluate various levels of parallelism on FPGA and GPU.

3.4.1 Serial Implementation

To evaluate the performance of the serial MSOT on heterogeneous platforms, we start by developing a serial (single-threaded) version of the code in ANSI C running on an Intel Core i7-3820 CPU. The input is a sequence of 120 frames of a soccer field, with 10 objects (players) being tracked. We also consider the maximum quality by selecting an iteration threshold of 60. Table 3.3 reports the execution efficiency of the serial code in term of Frames per Second (FPS) as we increase the number of objects in the scene from 1 to 10 objects. The quality of FPS significantly degrades as the number of tracked objects increases (e.g., 30 FPS for 1 object and only 2.5 FPS for 10 objects). The single-threaded (i.e., CPU version) execution is non-scalable. One possible solution is to reduce the value of Threshold, which results in a significant quality loss. Vision
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

markets are always demanding higher quality and improved performance together. As a result, the trend is toward leveraging accelerator architectures, including FPGAs and GPUs, that can exploit the parallelism present in vision tracking algorithms.

Table 3.3: Serial execution of MSOT

<table>
<thead>
<tr>
<th># of Targets</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance (FPS)</td>
<td>29</td>
<td>13</td>
<td>6.7</td>
<td>4.1</td>
<td>3.1</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Some previous studies have considered accelerating MSOT using GPUs [37, 64] and FPGAs [5, 42]. What these approaches have in common is a lack of insight into the possible performance opportunities available through tuning the implementation at a source level, and instead focus on optimizations working at very fine-grained resolution. Furthermore, they mainly focused on first-order implementation possibilities, achieving very limited speed-up (e.g., Li et al. [37] reported a 3.3X speed-up). In contrast, we look for opportunities using analysis of the algorithms involved, and by finding the right grain of parallelism. We study the effect of source-level parallelism choices on both FPGAs and GPUs acceleration.

3.4.2 Parallel Implementation

In Section 3.4.1 we showed that a serial (CPU-based) execution of MSOT algorithm is slow when tracking multiple objects at high resolution. Developing a parallel implementation of MSOT is not straightforward. Compared to many embarrassingly-parallel vision filters (e.g., Canny edge detection, convolution filtering), MSOT would appear to have much less inherent parallelism when working at a coarse granularity. In particular, the main factor hindering the parallelism potential is the inherent serial nature of MSOT. The algorithm computes a histogram of the current position, the calculating distance and gradually moving to a new position. This serialization factor makes develop a parallel MSOT very challenging.

After exploring a spectrum of parallel implementations, we have developed insight into the best path to accelerate MSOT using parallelism. We have identified how to leverage parallelism at multiple levels of granularity. Figure 3.13 highlights the different levels, ranging from coarse to fine, and includes object-level, neighborhood-level, window-level and instruction-level parallelism. Next, we will describe each level further, starting at coarse-grained parallelism and ending at fine-grained strategies.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Object-Level Parallelism (OLP): The first and coarsest level of parallelism present in the Mean-shift algorithm is object-level parallelism. The procedure for tracking the individual objects in the scene are completely independent. This allows several objects in a sequence of images to be tracked concurrently.

Neighborhood-Level Parallelism (NLP): OLP offers very limited parallelism – the number of OpenCL threads are bound by the number of objects in the scene, leading to a significant underutilization on the target architecture. One possible way to increase parallelism is to speculatively compute feature positions across the neighbors of the current position being computed.

The basic idea is to calculate the histograms and shift vectors not only for the current position of the objects, but also across a number of neighbors (which we will refer to as the search distance) in parallel. By speculating on these values, we are able to utilize a much higher number of parallel OpenCL threads (work-items). Speculatively computing feature positions can potentially lead to higher speedup. However, in general, this approach can exploit the parallelism present in the target architecture. On the downside, speculative execution in NPL introduces a serialization later in the execution. At the end of the neighborhood histogram calculation, a serial thread needs to run to identify the neighbor that matches the value at the current position with the value that has been estimated by the shift vector.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

**Window-Level Parallelism (WLP):** The histogram calculation involves all of the pixels of each object. One possible way to utilize a larger number of threads is to split pixels covering an object into smaller windows or segments (object segmentation), and then calculate the histogram across the segments in parallel. Window-level parallelism exposes far more parallelism and can potentially lead to higher speedups. On the downside, similar to NLP, there will be significant serialization delay when we need to gather the individual results and calculate the final histogram across all parallel threads.

**Instruction-Level Parallelism (ILP):** Even inside a window, there exists significant fine-grained instruction level parallelism between the pixels. It is very difficult to expose ILP working at the OpenCL source code level. ILP extraction will solely depend on the capability of the underlying architecture, as well as the target compiler.

### 3.4.3 Experimental Results

Next, we explore and evaluate the effects of source-level decisions on the execution efficiency of GPUs and FPGAs. The exploration focus is based on the parallelism approaches identified in Section 3.4.2.

#### 3.4.3.1 Execution Setup

All accelerator codes considered in our study are based on the OpenCL 1.0 standard (the version supported by the Altera tools). We have targeted two state-of-the-art accelerator architectures: the NVIDIA Tesla K20 GPU and the Altera Stratix-V FPGA. We have also utilized the NVIDIA CUDA Compiler (NVCC) and the Altera SDK for OpenCL v14.0 [1] for compiling the OpenCL codes on the GPU and FPGA platforms, respectively. To evaluate parallelism approaches on the GPU and the FPGA, we have used two different systems (listed in Table 3.4). Note, we are not trying to directly compare these systems against one another. For both architectures, we consider the Object-Level Parallelism (OLP) as the baseline. The experiments are carried out on a sequence of 120 full HD (1080×1920) frames of a soccer field, tracking 10 objects simultaneously. Based on our quality explorations, the bin size and the threshold are set to 4096 and 50, respectively, to achieve high quality.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Table 3.4: System characteristics

<table>
<thead>
<tr>
<th></th>
<th>System I</th>
<th>System II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>Core i7-3820</td>
<td>Xeon CPU E5410</td>
</tr>
<tr>
<td>Host clock</td>
<td>3.60 GHz</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>Device</td>
<td>Tesla k-20 GPU</td>
<td>Stratix-V FPGA</td>
</tr>
<tr>
<td>Device resource</td>
<td>2496 processor cores</td>
<td>622000 LEs</td>
</tr>
<tr>
<td>Device clock</td>
<td>706 MHz</td>
<td>100 MHz</td>
</tr>
<tr>
<td>Device memory size</td>
<td>5 GB</td>
<td>~8 GB</td>
</tr>
</tbody>
</table>

3.4.3.2 Heterogeneous Approaches

Utilizing the various levels of parallelism can generate serial computation overhead for the MSOT algorithm (i.e., there is no free lunch). Therefore, the OpenCL implementation of the MSOT is composed of both parallel and serial execution sections. In our heterogeneous implementation, the serial portion of the algorithm is executed on the CPU, while the parallel portion is executed on either the GPU or the FPGA.

**NLP Evaluation:** NLP is the coarsest level of parallelism that can be applied to our baseline implementation. Table 3.5 shows the impact of using NLP for the MSOT. When the search distance is 1, 9 neighbors are calculated in parallel for each object. Since the MSOT tracks 10 objects in our experiments, the total number of threads is 90. Similarly, the total number of threads is 250, 490, and 810 when the search distance is 2, 3, and 4, respectively. Increasing the search distance on the GPU platform increases the parallelism in the algorithm, and therefore, the resource utilization increases dramatically. However, the serialization factor added by NLP also increases. On the GPU, the best performance is achieved when the search distance is 2 (1.9X speedup). Increasing the search distance to more than 2 overloaded GPU resources and the speedup drops.

The Altera OpenCL SDK uses the concept of *pipelined parallelism* to map the OpenCL kernel code to the FPGA. It builds a deeply pipelined compute unit for the kernel. The compiler replicates the compute unit based on the available resources on the FPGA to expose parallelism. To gain the benefit of NLP optimization (which is a coarse-grained parallelism approach), the platform needs to have multiple compute units. However, since the MSOT kernel design is very large, we exhaust resources on the FPGA for the Altera OpenCL SDK to replicate the compute unit. Therefore, the FPGA base system, unlike the GPU base system, did not gain any benefit from the NLP approach.

**WLP Evaluation:** The WLP optimization can provide benefits to both platforms. Increasing the number of segments increases the parallelism, but also the serialization overhead. At
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Table 3.5: NLP speedup on a GPU and FPGA.

<table>
<thead>
<tr>
<th>Search Distance</th>
<th># of neighbors</th>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>1.58X</td>
<td>1.005X</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>1.91X</td>
<td>1.007X</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>1.77X</td>
<td>1.006X</td>
</tr>
<tr>
<td>4</td>
<td>81</td>
<td>1.63X</td>
<td>1.006X</td>
</tr>
</tbody>
</table>

Figure 3.14: WLP speed-up on GPU and FPGA

Figure 3.15: Speedup of the hybrid approach on a GPU
some point, the overhead of the reduction process will dominate any performance improvements provided by Window-Level Parallelism. Looking at Figure 3.14, the best distribution choice on System I is 128 (a 3X speedup). The total number of threads is 2560 (256 × 10) in this case. The same number is for System II is 32 (320 threads). The WLP optimization achieves a 4X speed-up on the FPGA platform in this case.

Similar to NLP, the Altera OpenCL SDK builds only one compute unit for the MSOT kernel. However, the compute unit that was generated is capable of running many fine-grained threads in parallel. Based on Table 3.5 and Figure 3.14, the FPGA platform is better suited to exploit fine-grained parallelism versus working at a coarser grain.

**Hybrid Evaluation:** Since both NLP and WLP provide performance improvements on the GPU platform, we wanted to explore a Hybrid implementation where we combine NLP and WLP to take advantage of both approaches. We varied the Search Distance (D) from 0 to 4, and the segmentation level from 1 to 1024. The combination of coarse-grained and fine-grained parallelism achieves a more efficient use of the GPU resources. However, the best source-level decisions are different from using NLP or WLP individually. In the Hybrid approach, the best performance (i.e., 6X speedup) is achieved by choosing a search distance of 4, and using 16 segments per object (Figure 3.15). The total number of threads in this case is 12960 (81 × 16 × 10).

### 3.4.3.3 Homogeneous Approaches

A GPU is a massively parallel device that can outperform a CPU when executing regular, data-parallel, applications. However, the weakness of the GPU is executing serial computations. On the other hand, a FPGA can handle both parallel and serial computations. This potential can lead programmers to develop better designs for the FPGA and achieve significant performance improvements. In this section, we evaluate the FPGA performance when running both parallel and serial portions of the MSOT algorithm. The OpenCL kernel is designed in such a way, that lots of threads are executed in parallel to perform the parallel section of the algorithm. Then, the first thread executes the remaining serial portion. The key benefit of using this approach is that we can reduce the overhead of transferring data between the host and the device. However, the drawback is that the kernel executable is large and uses more FPGA resources in comparison to the size of only the parallel kernel. Figure 3.16 shows that there is a huge benefit in using this homogeneous approach on the FPGA. We can achieve up to a 21X speed-up, which is much higher than previous approaches. The best performance improvement is seen when the number of segments per object is
3.5 Parallelism Type

This section analyzes the impact of source-level decisions, applied in OpenCL, on the FPGA’s execution efficiency. Our aim is to analyze the correlation between OpenCL parallelism semantics and parallel execution on FPGA devices to guide OpenCL programmers to develop optimized code. We focus on the impact of different types of parallelism (spatial and temporal) exposed by OpenCL on the generated data-path. In terms of spatial parallelism, we explore source code decisions used to create multiple data-paths for concurrent thread execution at various grains of parallelism. In terms of temporal parallelism, we zoom in on the source-level decisions necessary to optimize a pipelined execution model across many hardware threads. Pipelined execution helps to hide the memory access latency across hardware threads, resulting in a significant speed-up.

3.5.1 Spatial Parallelism Semantic

In this section, we study the correlation between the OpenCL source-level constructs to expose spatial parallelism and the synthesized data-path in the resulting FPGA architecture. To begin, Figure 3.17 illustrates a source-level construct commonly found in OpenCL kernels, and the corresponding synthesized data-path for an FPGA (as the result of OpenCL-HLS). To better understand the OpenCL kernel, we have split the kernel into three major parts: 1) memory read, 2) compute and 3) memory write. The resulting design contains only one CU, a data-path reflecting the OpenCL kernel. The generated data-path is deeply pipelined. Therefore, the spatial parallelism
in the source level translates to temporal thread-level parallelism across the pipeline stages. The
data-path issues and commits one thread each clock cycle, assuming perfect conditions (i.e., zero memory latency).

In the unoptimized implementation, throughput is bound to one thread per clock cycle. To achieve higher throughput, the programmer needs to guide OpenCL-HLS to synthesize an architecture that exploits spatial parallelism. Spatial parallelism can be exposed at various levels of granularity. The possible classes of spatial parallelism that can be exposed in OpenCL include: Compute Unite (CU) replication (CU replication), data-path replication (DP replication), and partial/selective data-path replicated (P-DP replication). Next, we consider each of these forms of spatial parallelism.
### 3.5.1.1 CU Replication

Working at a coarse level that performs *CU replication*, an entire CU is replicated, including the entire data-path, *id iterator*, and load/store units. The *dispatcher* splits the workload between multiple CUs, such that each CU performs the kernel function on a group of threads. Figure 3.18 shows the OpenCL pseudocode that can expose *CU replication*, and its corresponding synthesized data-path. A programmer can choose the number of synthesized CUs as an attribute in the OpenCL source code (`__attribute__((num_compute_units(2)))`). OpenCL-HLS will synthesize the corresponding CUs with respect to the availability of resources in the target FPGA device. Figure 3.18b presents replicated CUs for the same OpenCL kernel; both CUs execute the same data-path. In this case, each CU performs the kernel function on half the number of threads. In the best case, this results in a 2X speed-up as compared to unoptimized OpenCL code. This speed-up, however, comes at the cost of utilizing 2X the number of FPGA resources.

The programming decision to apply *CU replication* is fairly straightforward. The programmer has control over OpenCL-HLS to replicate an entire CU with minimum programming effort. Only one attribute is added to the OpenCL source code. However, this is not necessarily an efficient approach for complex kernels. When applying *CU replication*, the entire CU, including *id iterators*, load, and store units are replicated. As a result, *CU replication* is not often feasible for complex kernels with large code size, due to the FPGA’s limited compute and memory resources. Beyond the limitations placed on FPGA resources, the most important drawback of using *CU repli-
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

cation is the increased memory pressure on off-chip memory. In memory-bound kernels, increasing off-chip memory accesses degrades the performance due to the contention between CUs for the limited memory bandwidth on the device.

3.5.1.2 DP Replication

The next finer-grained optimization is data-path replication (DP replication). DP replication involves replicating the entire data-path inside the CU without replicating the id iterator and the load/store units. All other components of the CU remain unchanged. By replicating the data-path, the CU is able to execute multiple threads at the same time. Semantically, DP replication parallelism can be considered similar to the Single Instruction Multiple Threads (SIMT) model leverage in GPUs. In DP replication, each CU has multiple ALUs to execute the same instruction across multiple threads over multiple data. The replicated data-paths share same control signals.

Figure 3.19 presents the OpenCL pseudocode to expose DP replication, and its corresponding synthesized data-path. The programmers can choose the number of synthesized data-paths as an attribute in the OpenCL source code (\texttt{attribute((num\_simd\_work\_items(2)))}). Figure 3.19b illustrates DP replication method in one CU with two replicated data-paths. The CU is able to issue two threads per clock cycle, depending on the availability of data. Similar to the CU replication, the DP replication ideally can double the throughput. Compared to CU replication, DP replication is more efficient in terms of resource utilization, as it only replicates the data-path without the need for replicating id iterator and load/store units.
Similar to \textit{CU replication}, \textit{DP replication} requires minimal programming effort. Compared to \textit{CU replication}, \textit{DP replication} can achieve higher throughput (due to potentially generating less memory pressure) and with lower resource utilization (removing the overhead of synthesizing a new CU). The downside of \textit{DP replication} is the lock-step execution between the replicated data-paths, which will introduce additional execution stalls due to the lack of data. Since replicated data-paths share same control signals, they need to execute in synchronous lock-step mode. With lock-step execution, the data for both threads needs to be available, otherwise, both threads will be stalled. This limits the performance improvement in some memory-bound kernels. The requirement of lock-step execution also limits \textit{DP replication} to be applied to simple kernels with no data-dependent or conditional branches. \textit{DP replication} is not very useful for optimizing complex kernels containing conditional branches.

3.5.1.3 P-DP Replication

Working at finer grain, the data-path can be partially/selectively replicated (we refer to this as \textit{P-DP replication}). In \textit{P-DP replication}, the CU still issues one thread per clock cycle. However, we should be able to increase throughput by exposing sub-kernel level parallelism. Replicating the
entire CU depends on the available resources in FPGA, as well as the complexity of the OpenCL kernels. For kernels with complex control, creating the first CU uses most of the resources on the FPGA, and therefore there are not enough resources to replicate the entire CU. At the same time, DP replication for a complex kernel containing divergent threads is not possible. This makes P-DP replication a suitable choice when it is possible to parallelize compute-intensive portions of OpenCL kernels to achieve a higher throughput.

Figure 3.20 illustrates one example of P-DP replication. It splits a large OpenCL kernel into smaller functions, and replicates the compute-intensive portions of the data-path. In Figure 3.20b, a loop is replicated four times. Exposing P-DP replication in an OpenCL kernel is very challenging. While CU replication and DP replication can easily be exposed by using pragmas, exploiting P-DP replication needs significant source-level modifications. For example, programmers can use `num_compute_units` and `num_simd_work_items` pragmas to expose CU replication, and DP replication, respectively. However, to leverage P-DP replication, programmers need to split large OpenCL kernels into smaller kernels and replicate them manually. Also transferring data be-
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

tween kernels is another challenge. From this perspective, P-DP replication is similar to using the Dynamic Parallelism semantic available in CUDA and OpenCL, where a parent kernel launches several child kernels, independent of the host processor. The pseudocode that performs a child kernel launch is illustrated in Figure 3.20a. Dynamic parallelism is not supported by the current OpenCL-HLS tools. In the following, we discuss the OpenCL source code decisions that result in CU and data-path replication in the underlying architecture. The P-DP replication method will be explored in the next section, where we study the temporal parallelism present across kernels.

3.5.1.4 Spatial Parallelism Summary

Overall, with minimal programmer effort modifying source code, we can expose spatial parallelism in OpenCL kernels. However, the performance improvement of spatial parallelism is limited. The limitations are primarily due to increase memory pressure. Applying CU replication, the parallel CUs compete over the shared memory bandwidth. This sharing can result in long memory stalls and significantly limits the performance benefits in CU replication. Applying DP replication, given that the threads are executing in lock-step mode, the memory stall in one thread is propagated across all parallel threads, which limits potential performance benefits. Furthermore, both techniques introduce considerable overhead in terms of FPGA resource utilization. In the next section, we consider the effects of temporal parallelism found in OpenCL kernels on the execution efficiency of the resulting FPGA implementation. Temporal parallelism can potentially hide memory stalls, allowing the programmer to leverage P-DP replication.

3.5.2 Temporal Parallelism Semantic

In this section, we study the effectiveness of OpenCL constructs to expose temporal parallelism in OpenCL kernels. The benefits of spatial parallelism are limited due to the increase in memory stalls. Exploiting spatial parallelism is limited to simple kernels with regular execution patterns that contain no thread divergence. Temporal parallelism in OpenCL kernels enables programmers to effectively hide memory stalls during FPGA execution.

Applying the newly introduced Pipe semantic in OpenCL (released in OpenCL 2.0), it is possible to express temporal parallelism at an OpenCL source code level. The Pipe semantic offers an efficient way to launch multiple kernels that have data dependencies, allowing them to execute concurrently in a pipelined fashion (producer/consumer model). In the following, we study the impact of kernel-level and sub-kernel level temporal parallelism.
3.5.2.1 Kernel-Level Temporal Parallelism

In a non-Pipe execution model, kernels (producers and consumers) execute sequentially with data communication through off-chip memory. By exposing temporal parallelism using OpenCL Pipes, the kernels can be executed concurrently in a pipelined fashion.

Figure 3.21 illustrates the effect of temporal parallelism on multi-kernel applications. In sequential execution, (see Figure 3.21a) the host processor (CPU) launches the producer kernel and waits until it completes its processing. Later, it writes the result into global memory. Then the CPU launches the consumer kernel. The consumer kernel reads the produced data from global memory, performs its operation, and writes the final result into global memory for the CPU to use. Using pipelined execution (see Figure 3.21b), the CPU launches both the producer and consumer kernels at the same time. The first thread of the producer kernel completes its processing and writes a result into the Pipe for the consumer kernel. At this time, the first thread of the consumer kernel begins its processing. While the first thread of the consumer kernel is executing, the producer kernel starts executing a second thread. This sequence continues until all threads are finished. Then, the CPU reads the final result.
3.5.2.2 Sub-Kernel Temporal Parallelism

When an application encounters a high number of memory stalls, performance will suffer. In a deeply pipelined data-path, the memory stalls are directly exposed to the execution. As a result, if one thread is waiting for the memory, all following threads will be stalled until the waiting thread receives its data. To address memory stalls in OpenCL kernels compiled run on FPGAs, we propose exploiting sub-kernel temporal parallelism express at the OpenCL level of abstraction. Sub-kernel temporal parallelism results in a data-path that is able to hide a number of memory stalls.

To utilize temporal parallelism as a way to hide memory stalls, we separate the memory access portions that involve loading/storing data from the computation. This generates multiple kernels. While some of them are only responsible for memory accesses (loads/stores), others only perform computation. The kernels are connected via OpenCL Pipes. The kernels execute concurrently, but in an asynchronous fashion, while they communicate data through the Pipes. This allows separation of data access operations from the communication logic. As a result, the memory stalls occur in the memory access kernel can be hidden from the computation paths. In a flat, non-pipelined, implementation, if a stall occurs in the read stage, the whole pipeline stall until the data

Figure 3.22: OpenCL kernel and synthesized data-path sub-kernel temporal parallelism.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

loaded from memory.

Figure 3.22 presents the OpenCL pseudocode to expose sub-kernel temporal parallelism and its corresponding synthesized data-path. The generated data-path (Figure 3.22b) shows the split and concurrent pipelined execution between memory loads/stores and computation. The memory accesses are issued in parallel with the computation kernel, exchanging the data through pipes. As long as the Pipes are not empty, the sub-kernels execute concurrently across multiple threads and memory stalls are hidden.

3.5.2.3 Sub-Kernel Temporal Parallelism with P-DP replication

The second benefit of exposing temporal parallelism in large OpenCL kernels, is to leverage P-DP replication, which was described in Section 3.5.1. Splitting a large kernel into smaller sub-kernels opens the opportunity to replicate compute-intensive portions of the data-path.

Figure 3.23 presents the OpenCL pseudocode that exposes sub-kernel temporal parallelism and its corresponding synthesized data-path with P-DP replication. In the synthesized data-path, the first sub-kernel reads data from global memory and the last sub-kernel writes the results back into global memory. The middle sub-kernel performs the actual computation in the spatially-parallel model.

3.5.2.4 Temporal Parallelism Summary

Overall, the Pipe semantic in OpenCL offers the programmer the ability to overlap execution of multiple kernels. Using overlapped execution, the number of off-chip memory accesses is reduced significantly. When developing OpenCL programs for FPGA devices, the Pipe semantic can be effectively used to support concurrent execution of multiple independent kernels. In addition, the Pipe semantic can be utilized to hide memory stalls on FPGA devices, resulting in a higher throughput. Furthermore, it provides the opportunity to expose partial data-path parallelism within a kernel (P-DP replication). The overhead associated with this optimization is the pressure on on-chip memory required to realize the OpenCL Pipe semantic on the FPGA.

3.5.3 Experimental Evaluation

This section presents our experimental results and evaluation. At first, we introduce applications select for this study, as well as their baseline (FPGA-unaware) OpenCL implementations. We also present experimental results and evaluation for both temporal and spatial parallelism.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

__kernel void READ() {
    // load input from global memory
    read()
    // write input into the pipes
    write_channel_1()
    write_channel_2()
}

__kernel void PIPE_1() {
    // load input from the pipe
    read_channel_1()
    // perform the computation on even threads
    compute_1()
    // write result into the pipe
    write_channel_2()
}

__kernel void PIPE_2() {
    // load input from the pipe
    read_channel_2()
    // perform the computation on odd threads
    compute_2()
    // write result into the pipe
    write_channel_2()
}

__kernel void WRITE() {
    // load result from the pipes
    read_channel_1()
    read_channel_2()
    // store result into global memory
    write()
}

(a) pseudocode

Figure 3.23: OpenCL kernel and synthesized data-path exploiting sub-kernel temporal parallelism with P-DP replication.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

To carry out this study, we developed parallel OpenCL codes for three compute-intense applications from computer vision and big data analytic markets. MeanShift Object Tracking (MSOT) and Object Detection Vision Flow (ODVF) are two irregular kernels from the vision market. We also select Apriori Frequent Itemset Mining (AFIM) from the big data analytic market. These three applications are explained in Section 3.2 in detail.

3.5.3.1 Experimental Setup and Baseline Implementations

For our experiential evaluation, we targeted an Altera Stratix-V FPGA device. We implemented the applications in an extended version of OpenCL v1.0 that supports the Pipe semantic available in the Altera OpenCL-HLS tool-chain [1]. For synthesis and runtime profiling, we utilized the Altera SDK for OpenCL v14.0 [1]. Table 3.4 provides details of our experimental setup.

<table>
<thead>
<tr>
<th>Host</th>
<th>Xeon CPU E5410</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host clock</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>Device</td>
<td>Stratix-V FPGA</td>
</tr>
<tr>
<td>Device resource</td>
<td>622000 LEs</td>
</tr>
<tr>
<td>Device clock</td>
<td>100 MHz</td>
</tr>
<tr>
<td>Device memory size</td>
<td>~8 GB</td>
</tr>
</tbody>
</table>

Figure 3.24 provides an overview of our baseline implementations for the studied applications. For MSOT (illustrated in Figure 3.24a), we developed a single coarse-grained kernel that can track multiple objects in a frame concurrently. Each thread (i.e., OpenCL work-item) tracks one object in the scene. The baseline implementation of ODVF (see Figure 3.24b) consists of four different kernels working at a pixel-level granularity. Both MSOT and ODVF have been evaluated using a sequence of 120 full HD (1080×1920) frames of a soccer field with ten objects (soccer players) on the field, simultaneously. For AFIM, we applied the same parallel implementation as presented by Zhang [63]. Each thread reads two large k-item bitsets, finds the joint k+1-items bitset candidate, and computes the support ratio for the joint bitset. The AFIM kernel processes 16K candidates (bitsets) in each round of its execution, allowing up to 131,072 transactions in a database.

3.5.3.2 Spatial Parallelism Evaluation

Next, we explore and evaluate the benefits of spatial parallelism on our selected applications. To illustrate the potential benefits of DP replication, we focus on the AFIM application. The benefits of DP replication are hard to demonstrate in MSOT and ODVF, as they have large
Figure 3.24: Baseline Implementations
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

kernels containing divergent threads in their control flow. Instead, we use the MSOT and ODVF applications to explore and evaluate the benefits of CU replication.

To explore the potential impact of DP replication on performance, we implemented the AFIM kernel with 2, 4, and 8 replicated data-paths (2 DPs, 4 DPs, and 8 DPs implementations). As mentioned earlier, using the num_simd_work_items pragma, the synthesis tool can replicate the entire data-path to support execution of multiple threads, but then the CU has to read input data for multiple threads per clock cycle. This increases the number of memory accesses per clock cycle. Figure 3.25 shows the impact of DP replication on performance and resource utilization for the AFIM application. Increasing CU width increases the number of accesses, and therefore increases memory bandwidth utilization (see Figure 3.25b). At the same time, it also increases the number of stalls due to memory bottlenecks. Figure 3.25a shows that 8 DPs increases the number of memory stalls significantly, which significantly degrades overall performance. The memory bandwidth utilization also drops in the 8 DPs implementation. The maximum speed-up (2.8X speed-up) is achieved using the 4 DPs implementation. Figure 3.25c presents the corresponding resource overhead for each implementation. As can be observed in the figure, as we increase the width of DP replication, overall resource utilization increases. For the 8 DPs implementation for example, we use 13% more logic on the FPGA than the Baseline implementation. It also uses 12%, 11%, and 5% more Registers, Block Memory Bits, and DSP Blocks, respectively.

In the next experiment, we explore the impact of CU replication on the ODVF and MSOT applications. Of the four kernels in the ODVF code, MOG is the most compute-intensive kernel. We experiment with CU replication for the MOG kernel using the num_compute_units pragma. num_compute_units guides the synthesis tool to create multiple CUs for the target kernel. Similarly, we specify CU replication for the MSOT kernel using the num_compute_units pragma.

Figure 3.26 compares the Baseline implementations with a design with 2 CUs (i.e., replicating the CU twice) for ODVF and MSOT. We evaluate both performance and resource utilization. Although the bandwidth utilization is increased slightly, CU replication in both cases degrades the performance due to memory contention and a significant increase in the number of stalls (see Figure 3.26a and Figure 3.26b). For example, the 2 CUs implementation of ODVF has 20% more stalls than the Baseline. We also observe a significant increase in the resource utilization (see Figure 3.26c). On average, the 2 CUs implementation of ODVF uses 17% more resources than the Baseline implementation. Also, the 2 CUs implementation of the MSOT application uses 15% more resources utilization than the Baseline.
Figure 3.25: DP replication impact on the AFIM application.
Figure 3.26: CU replication impact on the ODVF and MSOT applications.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

3.5.3.3 Temporal Parallelism Evaluation

To expose temporal parallelism, we experiment with the OpenCL Pipe semantic. To demonstrate the benefits of utilizing temporal parallelism when developing OpenCL codes for FPGA devices, we categorize our case studies into multi-kernel and single-kernel applications. We explore the temporal parallelism across ODVF, a multi-kernel application, as well as AFIM and MSOT, two single-kernel applications.

To evaluate the benefits of temporal parallelism, we first focus on ODVF, which is a multi-kernel application. Then, we demonstrate the benefits of Pipe in hiding memory stalls for the AFIM and MSOT kernels. Further, we explore the benefits of kernel splitting to enable partial data-path replication (P-DP replication) at OpenCL source level.
Figure 3.27 compares our pipelined implementations with the baselines. We observe a significant reduction in the number of memory stalls across all benchmarks. Furthermore, the global off-chip memory accesses are replaced with on-chip Pipe reads and writes, which further reduces latency. This also improves bandwidth utilization (see Figure 3.27b). We see a 2.4X and 2.7X speed-up in the MSOT and ODVF pipelined implementations, respectively (see Figure 3.27a). In AFIM, the pipelined implementation does not improve performance versus the baseline implementation since the computation kernel reads input for one thread in each clock cycle, and the actual throughput is still limited to one thread per clock cycle.

Figure 3.27c compares various implementations in terms of resource utilization. Overall, the overhead of temporal parallelism is fairly low. The synthesis tool uses Block memory Bits to implement OpenCL Pipes. Therefore, we see an increase in Block memory bits utilization in all case studies. Exposing temporal parallelism in ODVF and MSOT, however, simplifies the datapath. In both applications, the pipelined implementation uses less Registers and DSP Blocks.
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

As we discussed earlier, by splitting large OpenCL kernels into smaller kernels, the programmer can expose spatial parallelism more effectively when using the P-DP replication method. The effect of P-DP replication on AFIM is illustrated in Figure 3.28. In contrast with our spatial-parallel implementations (which experienced limited speed-up (Figure 3.25a), we see an up to 95X speed-up in our P-DP implementations (see Figure 3.28a). Increasing the spatial parallelism factor in these implementations increases both BW utilization and reduces stalls, since the read kernel reads data for multiple threads. However, the memory read accesses are coalesced and the stalls in P-DP implementations are fewer than in the DP implementations (see Figure 3.28b and Figure 3.25b).

Figure 3.28c shows that the P-DP implementations use more Block Memory Bits than the DP implementations. However, in terms of other resource utilization, P-DP implementations are more efficient than DP implementations. For example, an 8 P-DP implementation uses 22% of the Logic, 19% of the Registers, and 0% of the DSP Blocks, while an 8 DP implementation (see Figure 3.25c) uses 33%, 26%, and 6% of Logic, Registers, and DSP Blocks, respectively.

Similar to AFIM, P-DP replication in MSOT increases BW utilization as well as stalls (see Figure 3.29b). In case of MSOT application however, the stalls increase is dominant. Therefore, the speed-up is limited to 3.4X in 2 P-DPs implementation (Figure 3.29). The performance is decreased after this point since the number of Pipes is increased and the memory reader kernels cannot provide enough data for all kernels. Figure 3.29c compares various P-DP implementations of MSOT in terms of resource utilization. The block memory bits usage is increased significantly in pipelined implementations in order to create OpenCL Pipes across the kernels. However, having smaller kernels and removing the overhead of barriers in the pipelined implementations, the utilization of other resources is decreased slightly. For example, 2 P-DPs implementation uses 30% of Logic, 24% of Registers, and 15% of DSP Blocks, while 2 CUs implementation (see Figure 3.26c) uses 52%, 49%, and 24% of Logic, Registers, and DSP Blocks respectively.

3.5.4 Discussion

OpenCL on FPGAs can move these devices from serving the role of prototyping, and become a heavily used processing component in future heterogeneous platforms. Parallel programmers who are familiar with OpenCL semantics should be able to compile their applications to run on an FPGA. This enables programmers to develop a customized data-path for compute-intensive kernels without getting involved in implementation details. At the same time, device-dependent
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

Figure 3.29: The impact of P-DP replication for the MSOT application.

(a) Performance

(b) Stalls/Mem BW Utilization

(c) Resource Utilization
(FPGA-dependent) optimizations need to be considered by the OpenCL programmer in order to fully utilize the benefits of a customized data-path on FPGA devices.

Our analysis demonstrates that memory stalls are the primary barrier to achieving high throughput of OpenCL kernels on an FPGA. In contrast to GPUs, which hide memory stalls by overlapping the processing and memory accesses using concurrent execution of many parallel threads, FPGA are less able to hide these memory stalls. This significantly limits the benefits of spatial parallelism on FPGAs. Spatial parallelism, working at Compute Unite (CU) granularity, introduces inter-CU memory contention, resulting in considerable memory stalls exposed to each CU’s data-path. Thus, exploiting spatial parallelism within a CU by replicating the kernel data-path (DP replication) has very limited benefits. As the replicated data-path executes in lock-step mode (due to sharing common control signals), a memory stall in one data-path stalls all other parallel data-paths.

Our analysis also reveals that by utilizing temporal parallelism available in OpenCL, memory stalls can be hidden from the execution. The FPGA synthesis tools already leverage deep pipelining to achieve high throughput. The temporal parallelism available in OpenCL separates data-accesses (memory loads/stores) from computation. The memory access kernels execute concurrently with compute-only kernels, but in an asynchronous fashion, communicating data through OpenCL Pipes. Our results demonstrate that temporal parallelism can partially hide memory stalls during execution. Furthermore, by hiding the memory stalls, temporal parallelism opens up new opportunities to take advantage of spatial parallelism on FPGAs. Using hybrid temporal+spatial parallelism, we can achieve much higher throughput. For example, in the AFIM application, DP-replication with 4 parallel data-paths achieves a peak throughput (2.8 times over the baseline implementation). By combining the benefits of temporal parallelism, DP-replication generates 128 data-paths and results in a 95X speedup over the baseline implementation.

Overall, research on OpenCL computing on FPGAs is in its infancy. We believe that FPGAs can deliver even higher efficiency for applications written developed in OpenCL. To achieve higher throughput, we will need advances in two major areas: 1) OpenCL programming support and (2) the OpenCL synthesis tools.

OpenCL has been designed to support GPU execution (focused on exploiting spatial parallelism). To deliver higher efficiency for FPGAs, we need to rethink OpenCL semantics. The OpenCL programming paradigm needs be expanded to support FPGA-specific optimizations. New semantics are required to bit map OpenCL codes to FPGAs with forcing the programmer to worry about implementation details. We have already observed some promising benefits of using the Pipe semantic across a number of parallel kernels. However, new semantics are required to expose
CHAPTER 3. SOURCE OPTIMIZATION APPROACH

hybrid spatial-temporal parallelism in OpenCL programs. Providing new hybrid spatial-temporal semantics can also allow synthesis tools to generate much more efficient data-paths. In this paper we have applied manual modifications to the code to enable optimization. One missing aspect in OpenCL is the lack of memory coalescing. Memory coalescing on FPGA devices is a function of type, granularity and degree of parallelism of the kernel.

3.6 Summary

This Chapter explored the challenges and opportunities provided in the OpenCL language when targeting FPGA devices. We primarily explored the potential benefits of using OpenCL Pipes on an Altera FPGA. We proposed three different methods to synchronize concurrent OpenCL kernels. To drive our study, we evaluated an object detection vision application. As compared to a sequential kernel baseline, we achieved a 2.8X speed-up when using the proposed dimension transform module. This speed-up translates to 57 frames per second.

We also explored parallelism granularity on GPUs and FPGAs. We showed how to exploit different classes of parallelism on a GPU and FPGA platform. We reported on the performance of the Mean-Shift object tracking algorithm on each platform. Our experiments showed up to a 4X speed-up on an FPGA-based platform when using WLP approach, and up to a 6X speed-up on a GPU-based platform when using both NLP and WLP approaches. Also, if we execute both parallel and serial sections of the algorithm on an FPGA, this can produce a 21X speed-up.

Finally, We focused on the correlation between OpenCL’s ability to express parallelism and execution model of an FPGA. The aim was to provide early insight into the potential of OpenCL when targeting FPGA devices, as well to provide guidance to OpenCL programmers and OpenCL synthesis tool developers on the benefits of spatial and temporal parallelism. We explored programming decisions that result in a more efficient data-path, increasing thread-level parallelism, while hiding memory stalls. We evaluated 3 challenging applications and found that FPGA-aware OpenCL codes can achieve much higher speed-up as compare baseline implementations targeted for GPUs. To achieve the best performance, the OpenCL code needs to leverage temporal parallelism to hide the memory access latency. The results of this research can also help the FPGA synthesis community to produce more efficient data-paths for OpenCL programs.
Chapter 4

Synthesis Optimization Approach

OpenCL provides a promising semantic to capture the parallel execution of massive number of threads. The primary aim of OpenCL is to provide a universal programming interface across many heterogeneous devices (e.g. CPUs, GPUs, FPGAs, and special accelerators). While OpenCL guarantees functional portability, the achieved performance depends on the target architecture.

Every architecture has its own strengths and weaknesses when running OpenCL applications. GPUs for example, are many core devices achieving a very high throughput by concurrent execution of massive number of threads on many cores. GPUs can hide memory latency by switching the threads when they are waiting for data. However, the general purpose CUs, makes the GPU architecture inefficient in comparison with application-specific CUs in FPGAs and special accelerators.

In contrast to GPU architectures with massively parallel fixed ALUs, FPGAs re-configurability allows construction of CUs containing customized data-path for the OpenCL threads. Due to the limited bandwidth and logic resources, FPGA major benefit stems from pipelining. The generated data-path receives OpenCL threads in-order and executes them in a pipelined fashion. Although the data-path is deeply pipelined, the memory stalls in one thread, blocks other threads execution. In this chapter, we propose a method, called Hardware Thread Reordering, to evaluate the effectiveness of thread switching as a synthesis optimization technique on FPGAs.

4.1 Related Work

Previous studies have considered multithreaded execution on FPGAs. Some have focused on executing multiple-kernels on FPGAs [31], while some others have studied executing multiple
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

threads in a single kernel[44, 53, 28, 54, 55]. The CHAT compiler [28] generates multithreaded data-paths for dynamic workloads. In this compiler, a Thread Management Unit (TMU) dispatches the threads to multiple Processing Elements (PEs). The TMU dynamically balances threads across multiple PEs by switching to ready-to-run threads.

The CHAT compiler exposes spatial parallelism by replicating the PEs. However, CHAT ignores temporal parallelism or pipelining as a method to exploit parallelism on FPGAs. Nuvitadhi et al. [44] proposed a synthesis technique to generate a multithreaded pipelined data-path from a high-level unpipelined data-path specification. They used transactional specifications (T-spec) to capture an abstract data-path, and T-piper to analyze and resolve hazards and generate the RTL implementation of the pipelined design [43].

ElasticFlow [54] is a synthesis approach for pipelining kernels with dynamic irregular loop nests. ElasticFlow proposes an array of loop processing units (LPUs) and dynamically distributes inner loop iterations to run on LPUs. While ElasticFlow targeted inner loops for pipelining, Turkington et al. [55] proposed an outer loop pipelining approach. They extended the Single Dimension Software Pipelining (SSP) [49] approach to better suit the generation of schedules for FPGAs. However, all of these studies considered in-order threads or loop-based execution. The in-order thread execution approach has also been used in commercial OpenCL-Verilog compilers by Altera [1] and Xilinx [2]. Out-of-order thread execution, that is present in a number of important applications, has not been considered as a path to achieve much better efficiency of the multithreaded data-paths.

A context switching mechanism has been proposed by Tan et al. [53] that supports out-of-order execution in the pipelined data-paths. However, a deeper study of related aspects of out-of-order execution, such as stall management and thread scheduling, has not been pursued. This paper presents a hardware thread reordering approach to enhance the efficiency of multithreaded data-paths for irregular OpenCL kernels.

4.2 Hardware Thread Reordering

Since the introduction of massively parallel programming models, such as OpenCL, one important research question has been the efficiency of FPGAs to support these programming models. Recent studies have shown that a deeply pipelined data-path can achieve a very high throughput for OpenCL kernels [17, 40, 50]. The high throughput, in particular, is pronounced for regular kernels with deterministic execution patterns (no runtime conditional branches). In such a scenario, the
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

OpenCL threads share the same data-path and execute in an in-order fashion throughout the pipeline stages (thread-level temporal parallelism). This results in very high data-path utilization, and thus, high program throughput.

Recently, there has been renewed interest running complex machine-learning and deep-learning algorithms on FPGAs. These algorithms often contain non-deterministic control flow with varying execution patterns across the threads. With in-order thread execution, only one thread is allowed to execute the non-deterministic part of the generated data-path. As a result, other threads have to wait for the current thread to finish its execution. For irregular kernels, the in-order thread execution significantly reduces the amount of temporal parallelism available across threads, significantly impacting data-path utilization, and thus, limiting application throughput. There has been limited prior work on thread-level parallelism on FPGAs. New research is required to enhance the utilization of the FPGA’s data-path when targeting massively parallel applications. Such research can help FPGAs to deliver much higher throughput for irregular OpenCL kernels.

This chapter proposes a novel approach called Hardware Thread Reordering (HTR) to enhance an FPGA’s efficiency when targeting irregular massively-parallel kernels processing non-deterministic runtime control flows. The aim of HTR is to achieve significantly higher throughput by increasing the data-path utilization. Its key insight is relaxing the in-order thread execution by enabling the threads reordering at basic-block granularity. In a nutshell, HTR proposes to extend synthesized basic-blocks with independent/dedicated control signals and context switching registers. To further enhance data-path utilization, we also propose a set of optimization techniques to manage competition over the shared resources to further reduce the number of unnecessary stalls across reordered threads. To demonstrate the efficiency of our proposed approach, we use three parallel irregular kernels from standard benchmark suites. For all the benchmarks, we compare the effectiveness of our HTR-enhanced data-path against a baseline (in-order) data-path.

4.2.1 Background and Motivation

OpenCL offers a suitable programming model to capture compute-intensive kernels with massive thread-level parallelism. FPGAs, in principle, can achieve very high throughput by providing a customized data-path for OpenCL kernels. To further enhance FPGA throughput and increase thread-level parallelism, the generated data-path is often deeply pipelined with unrolled loops. The OpenCL threads share the same data-path and execute in an in-order fashion throughout the pipeline stages (temporal parallelism).
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

__kernel void SPMV( __global int *row, __global int *val, __global int *col, __global int *vec, __global int *out, const int dim)
{
    int id = get_global_id(0)
    if (id < dim) {
        int tmp = 0
        for (c = row[id]; c < row[id+1]; ++c) {
            tmp += val[c] * vec[col[c]]
        }
        out[id] = tmp
    }
}

Figure 4.1: SPMV OpenCL kernel

An FPGA’s throughput can be significantly impacted when faced with running irregular kernels that contain data-dependent branches. The primary challenge is due to the significant reduction in data-path utilization for the runtime-dependent non-deterministic regions (with data-dependent branches) of the data-path. With in-order execution, thread-level pipelining stalls during non-deterministic regions occur given that the next thread has to wait for the current thread to finish. Furthermore, with variable memory latency, deep pipelining is inefficient and imposes a huge overhead due to the large number of delay buffers that would need to be added to hide the memory latency. With in-order thread execution, loop unrolling and loop pipelining will not be applicable for run-time dependent dynamic loops.

Figure 4.1 presents the code of the sparse matrix vector multiplication (SPMV) kernel captured in OpenCL. SPMV is an example of an irregular kernel with run-time dependent control flow, containing thread dependent conditional IF statements, as well as LOOPs with variable runtime dependent iterations. In the FPGA synthesis flow, the high-level language is compiled to the LLVM intermediate representation. The LLVM instructions will be scheduled into clock cycles by the HLS tool. The instructions that are scheduled into the same clock cycle will be mapped to a pipeline stage. Figure 4.2 represents the LLVM IR and the control flow of the SPMV kernel. Overall, the kernel contains five basic-blocks with three runtime dependent branches. Figure 4.2 also shows the mapping between LLVM instructions and Pipeline stages. Notice that each load instruction is mapped to two pipeline stages. In the first stage, request, the load request is sent to the memory module. In the next stage, load, the data is received from the memory. The memory latency in this example is assumed to be 2 clock cycles. The generated pipeline data-path is illustrated in
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

Figure 4.3: The data-path and the number of pipelined stages are based on the real synthesis reported by the LegUp toolchain; for simplicity, we do not present the internal computational logic per each basic-block.

Figure 4.4 illustrates a pipeline timing diagram of the in-order execution of the SPMV data-path. As shown in the figure, as thread_0 executes the runtime dependent loop (stage 5 to stage 9), Thread_1 and all following threads are stalled in their current pipeline stages. After two iterations
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

Figure 4.3: Generated data-path for SPMV kernel

of thread_0 in the non-deterministic region, thread_1 will enter the loop section, and similarly, all following threads will be stalled until thread_1 finishes its runtime-dependent loop region.

Overall, the example of SPMV reveals the inefficiency of in-order thread execution for irregular kernels with runtime dependent branches. As highlighted by Figure 4.4, the data-path is often under-utilized. In the following Section, we demonstrate the principles of hardware thread reordering to increase the data-path utilization to achieve a higher throughput.

4.2.2 Hardware Thread Reordering

As illustrated in Section 4.2.1, in-order thread execution is inefficient when the control flow graph of a kernel contains data/thread-dependent branches and dynamic loops. To remove this source of inefficiency and enhance the FPGA’s utilization, this section proposes Hardware Thread Reordering (HTR) for out-of-order execution of threads over a shared datapath. The thread reordering in principle is done at a basic-block granularity, which can create a non-deterministic execution order for the pipelined threads. To support thread reordering, HTR enhances the generated data-

Figure 4.4: Pipeline timing diagram of the SPMV datapath.
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

path in two aspects. For the first aspect, \textit{Hardware Thread Switching} introduces additional logic and memory elements necessary to support thread switching. For the second aspect, \textit{Hardware Thread Arbitration} is added to support thread arbitration across the reordered hardware threads to better manage shared compute resources.

4.2.2.1 Hardware Thread Switching

In order to support out-of-order hardware thread execution over a multithreaded pipelined data-path, we propose two major extensions to maintain a thread’s live variables and computational status across pipeline stages.

First, every pipeline stage needs to hold the contexts of its currently executing thread. We define the context of a hardware thread as the live variables required to perform the associated thread computation in the current, as well as all following, pipeline stages. This may include some of the input variables, and the intermediate variables which were produced by predecessor stages and will be used in future stages. For example, stage 5 of the SPMV data-path has six live variables. Variables \(c\), \(tmp\), \(6\), and \(7\), which are initialized in this stage as well as variables \(id\), and \(up\) which are initialized in predecessor stages and being used by the following stages in the pipeline. Figure 4.5 shows the context variables of each pipeline stage in the SPMV data-path. A context register file in the extended pipeline stage is added to store all live variables.

![Figure 4.5: Context variables per pipeline stages of SPMV](image)

Second, it is necessary to hold the status of the every pipeline stage. Each stage of the pipeline should perform its computation whenever its input data is valid. To achieve this, we propose to add single bit active mask to all pipeline stages. For example, for the SPMV data-path, stage 1 receives thread 0 and performs its computation on thread 0, while all other stages are inactive. The next cycle, stage 1 passes thread 0 and its live variables to stage 2, and receives thread 1. Both stage
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

<table>
<thead>
<tr>
<th>Stage</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>2</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>3</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>4</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>5</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>6</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>7</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>8</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>9</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>10</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>11</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
<tr>
<td>12</td>
<td>T0</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T5</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
<td>T9</td>
<td>T9</td>
</tr>
</tbody>
</table>

Figure 4.6: Out-of-Order execution in SPMV kernel

1 and 2 are active and the other stages are inactive during this cycle.

Figure 4.7 presents an abstract visualization of our proposed approach. If the stage is active, the context register file loads the values from the previous stage to perform its computation. Otherwise, the context register file holds its current value. The active mask receives its value from

Figure 4.7: Extended Pipeline stage for HTR approach
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

Figure 4.8: Generated HTR-enhanced datapath for SPMV kernel

the active mask in the preceding stage. If in clock $t$, stage $i$ is active, in clock $t+1$ stage $i+1$ would be active.

Figure 4.9: Extended pipeline stage with stall signal.

4.2.2.2 Hardware Thread Arbitration

The HTR-enhanced pipeline stages described in the previous section enable the option to execute concurrent re-ordered threads across pipeline stages. Ideally, during each clock cycle, one thread enters the pipeline (in any order), and one completes its execution and exits the pipeline. An example of out-of-order execution of the SPMV kernel is illustrated in Figure 4.6. However, with out-of-order thread execution, the reordered threads may compete for shared resources. The contention may occur over computational resource. For example, two parallel threads may compete over the merged pipeline stage after the branches. Contention can also occur over shared memory when the number of concurrent parallel memory accesses is more than the available number of memory ports.
In our SPMV data-path, stages 5, 10, and 12 are merged stages. Each of these stages can receive input from two preceding stages. If both preceding stages are active in cycle $t$, only one of them can proceed to the next stage in cycle $t+1$. The other has to wait until the conflict is resolved. Also, stages 2, 5, and 7 are memory load stages and connected to the global memory ports. Connecting this data-path to a single port memory module means that only one of these stages can be active at a time. The other two have to wait until the memory port is available.

In the following, we introduce Basic-Block Level Data-path Control and Dedicated Memory Management to control, arbitrate and manage shared computation and memory resources across concurrent re-ordered threads.

### 4.2.2.3 Basic-Block Level Data-path Control

We propose a dedicated control logic unit to manage the arbitration over the computation resources. When there is a conflict/contention between threads in the pipeline, the control logic unit schedules threads, allowing one to execute and stalling the other threads until the conflict is resolved. Figure 4.9 illustrates the extended pipeline stage with the control (stall) signal. In this case, the pipeline stage performs its computation if the stage is active and the stage is not stalled by the control unit. If the stage is stalled, it will retain its current context register and active mask values until the stall is cleared.

The thread scheduling and stall management can be performed working at different granularities. At a coarse-grained level, the whole data-path has a single stall signal. If a conflict happens in the pipeline, all the stages are stalled until the conflict is resolved, no matter if the stage is involved in the conflict or not. In this approach, a stall in one stage impacts all other stages in the pipeline. Working at a fine-grained level, each stage has its own stall signal, and the control logic unit controls stages independently. This method results in more efficient pipeline execution since it stalls only the stages involved in a conflict. The other stages still perform their execution. However, this method involves a complex scheduling scheme in the control logic. This complex control logic can become a bottleneck in the design and degrade the clock frequency in the synthesized hardware. Our control unit uses a middle approach. It stalls the stages at a basic-block granularity.

In our proposed method, all pipeline stages of a basic-block will be controlled by a single stall signal. In other words, in-order thread execution is performed within each basic-block. However, different basic-blocks have independent stall signals. This allows the data-path to stall only the basic-blocks with conflicts, and execute the others. This method is more efficient than a data-
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

path with a single stall signal, while the control logic unit remains simple. Figure 4.8 shows the basic-blocks, and includes the stall signals. Note that the stall in one basic-block may still impact the processing of the preceding basic-blocks.

The control unit also decides which basic-block has to be stalled when two basic-blocks have a conflict. In our approach, a round-robin policy is used for memory accesses. When two stages in two different basic-blocks compete to access the single ported memory module, the control logic schedules the accesses in a round-robin fashion. In addition, the control logic unit gives higher priority to the thread which is inside the loop. For example, in the SPMV kernel, if stage 4 in basic-block “For.Entry” and stage 9 in basic-block ”For.Body” both need to precede to stage 5, the control logic stalls basic-block “For.Entry” to service basic-block ”For.Body” which has higher priority.

4.2.2.4 Dedicated Memory Management

Contention over memory increases the number of stalls in multi-threaded data-paths. For example, using a dual-ported memory module limits the number of stages that can concurrently access the memory to two active stages at a time. We propose a dedicated memory management module to manage and arbitrate the concurrent memory requests to the memory. In this way, the number of memory stalls exposed to data-path reduces. Memory management module buffers the concurrent requests and schedules them for the memory access.

Figure 4.10 illustrates a memory request handler used in our implementations. It uses FIFOs to buffer the requests submitted from the concurrent stages (up to 4 memory stages in this example). A request handler decides which request will be submitted to global memory if more than one is available. The data returned from the global memory is also stored in output FIFOs to be utilized by pipeline stages.

Adding the memory management module reduces the number of stalls significantly. The control unit has to take the FIFO’s status into account when it decides to activate or stall a basic-block. If a memory request stage is active, and its request FIFO is full, the basic-block has to stall. Similarly, when a memory load stage is active and its data FIFO is empty, the basic-block has to stall.
4.2.3 Optimizations Methods

HTR-enhanced data-path offers out-of-order thread execution to increase data-path utilization and thus improve the performance. Overall, with thread reordering the number of stalls across the threads reduces compared to in-order thread execution. However, there are still a large number of stalls incurred due to the stalls propagation across the basic-blocks and memory accesses. This section explores some optimization methods to reduce the number of data-path stalls and further improve the utilization of HTR-enhanced data-path.

4.2.3.1 Basic-block Stalls Isolation

In the stall management mechanism described in Section 4.2.2.3, stalling in one basic-block may propagate to the preceding basic-blocks. For example, in the SPMV pipeline Figure 4.8, stall in "For.Body" basic-block will propagate to the "For.Entry" basic-block. This means that threads executing basic-block "For.Entry" will be unnecessary stalled due to an stall in "For.Body" basic-block. This increases the number of stalls and degrades performance.

To avoid stall propagation, we propose adding FIFOs across the basic-blocks. In the
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

previous example, the “For.Entry” basic-block puts thread \( i \) and its variables into the FIFO without being stalled. When the stall in the “For.Body” basic-block is resolved, it reads thread \( i \) from the FIFO and executes it. These FIFOs improve the throughput by reducing the stalls, but with the added cost of imposing area overhead for implementing the data-path. By optimizing the FIFO depth, we can reduce this overhead. Finding the optimal depth of the FIFOs has been left for future study.

4.2.3.2 Memory Stalls Isolation

Current FPGA synthesis flows translate every memory access instruction in the LLVM IR into two different stages. The first stage sends the load request to the memory module. The second stage receives and consumes the data from the memory module when the data is available. We call these two stages request and load. Consider a scenario when the load stage in a basic-block is active but the data has not been received from the memory module. The control unit stalls the whole basic-block, including the request stage. This means the control logic prevents the request stage from submitting a new request, impacting effective bandwidth utilization and therefore application throughput.

To resolve this issue, we isolate the request and load. The approach is based on splitting a basic-block that contains a load instruction into two sub-blocks, one containing the request stage, and the other containing the load stage. Then the request stage still can submit a new request, even if the load stage is stalled.

4.2.4 Implementation Method

This section describes our implementation method of Hardware Thread Reordering in a pipelined data-path. We used LegUp compiler [11], to generate the pipelined implementation of the kernels. Then we manually modified the generated Verilog code to add the support for HTR technique. Although the HTR implementation is manual at this point, however the process is algorithmic and can be added to the high-level synthesis tools such as LegUp. Automating this process is left for the future work.

The HTR implementation process is represented in Figure 4.11. In the first step, we use the LegUp compiler to generate a baseline data-path for a given C code. The C code represents the actual kernel function of a OpenCL program which will be executed by each thread. Next, we perform live-variable analysis to determine the live variables in each pipeline stage. In the
baseline implementation, there is a single register associated with each variable. However, in the HTR implementation each register will be replicated for every pipeline stage that the variable is alive. This step is called register replication. Next step is to add the support of multi-threaded pipeline stages. In this step, we modify each pipeline stage as presented in Figure 4.9 by adding the active and stall signals. Next, we add the stall management unit (see Figure 4.8) to stall the pipeline stages when there is a conflict. Finally, to reduce the number of stalls, we add FIFOs between the basic-blocks. The FIFO width is determined by the number of live variables moving from one basic-block to another. The final output is a multi-threaded pipelined data-path with out-of-order thread execution support.

4.2.5 Experimental Results

4.2.5.1 RTL Simulation Setup

To evaluate the efficiency of the HTR approach, we use three irregular kernels from standard benchmark suites, sparse matrix vector multiplication (SPMV), K-means clustering (KM) and image convolution (CONV). In particular, we focused on the irregular benchmarks with runtime dependent conditional branches. The SPMV and KM kernels contain dynamic loops, while the CONV kernel contains various irregular branches. We used LegUp compiler [11], to generate the pipelined implementation of the kernels. The LegUp implementations are considered as the baseline datapath (in-order thread execution). To construct the HTR-enhanced datapath, we expand the
baseline pipeline with the additional components introduced by HTR components; e.g. context registers, reordering control and memory management modules. In addition, the optimization methods explained in Section 4.2.3 have been added to the HTR-enhanced datapath. The global memory delay for all implementations is fixed, 10 clock cycles. All the implementations are synthesized by Altera Quartus for CycloneIV FPGA.

4.2.5.2 Throughput Comparison

Figure 4.12 compares the throughput of HTR-enhanced datapath against the baseline implementation (in-order datapath). On average, the HTR-enhanced datapath achieves 6.7X higher throughput compared to the baseline across all three benchmarks. The highest speed-up (11.2X) is achieved in CONV, as CONV does not contain data-dependent loops. The significant speedup primarily achieves due to pipelined execution of non-deterministic runtime dependent sections of datapath. While in baseline implementation (the result of LegUp tool), the computation sections with thread-id and data-dependent branches are not pipelined, the HTR is able to generate a pipelined datapath with reordered thread execution for entire design. With reordered thread execution, the

![Figure 4.12: Speed-up](image-url)
memory bandwidth utilization increases. Furthermore, it reduces the number of datapath stalls, resulting in as a higher datapath utilization.

### 4.2.5.3 Datapath Stalls and Utilization

To provide more insight about the source of significant throughput improvement in HTR-enhanced datapath, Figure 4.13 shows the memory bandwidth utilization across the benchmarks. We observe significant increase in BW utilization in HTR-enhanced datapath compared to baseline datapath (with near 100% utilization for KM and CONV benchmarks, and 90% on average). Figure 4.14 illustrates the number of datapath stalls and their corresponding sources (computation and memory). Figure 4.14 presents that on average 95% of the stalls are due to memory requests, and 5% is due to compute resource conflicts in merge stages. Overall, the 99% memory bandwidth utilization and 95% stalls in memory requests demonstrate that increasing the global memory bandwidth (e.g. doubling the memory modules), the HTR approach can achieve even higher throughput.

The SPMV kernel contains dynamic loop. Therefore, the LegUp tool is not able to pipeline this kernel. The HTR implementation however, pipelines the thread execution and achieves 5.2X speed-up over the baseline. The HTR improves the bandwidth utilization by 40%. However, the HTR implementation is not able to fully utilize the bandwidth, and the bandwidth utilization is stuck at 50%. The reason is the two consecutive memory requests in the SPMV kernel where the first load requests the address for the second load. In this case, the consecutive requests cannot be pipelined. As it shown in figure Figure 4.14 all of the stalls in this implementation is due to memory requests.

In KM benchmark, the memory bandwidth utilization of the baseline implementation is much higher (4X) than the other benchmarks, as the KM is inherently a memory-bound benchmark. Although the HTR implementation is able to increase the bandwidth utilization to 100%, the speed-up is limited to 3.5X. However, increasing the memory bandwidth can potentially increase the throughput in the HTR implementation.

### 4.2.5.4 Resource Overhead

Overall, the HTR execution occupies pipeline stages more efficiently and improves the throughput significantly. This however, comes with the cost of more resource utilization on the FPGA. Figure 4.15 and Figure 4.16 compare the HTR implementation with the baseline implementation in our three benchmarks in terms of resource utilization. On average, we see 1.9X increase
in logic resource utilization, and 1.3X increase in register utilization. This overhead is mainly due
to register replication and FIFOs used in HTR implementation. Optimizing the depth of the FIFOs can reduce this overhead significantly. Also, some LLVM optimization passes can be used to reduce the number of variables passing across basic-blocks as well as the size of the variables. These optimizations are left for future studies.
CHAPTER 4. SYNTHESIS OPTIMIZATION APPROACH

4.3 Summary

This Chapter proposed a novel Hardware Thread Reordering (HTR) approach to enhance the throughput of OpenCL kernel execution on FPGAs. The HTR approach works at a basic-block level granularity, generating control signals to perform out-of-order thread execution in irregular kernels possessing non-deterministic runtime-dependent control flow. We demonstrated the efficiency of our HTR approach on three irregular kernels: SPMV, KM, and CONV kernels. HTR can achieve up to 11X speed-up with less than 2X increase in resource utilization.
Chapter 5

Architectural Optimization Approach

In Chapters 3 and 4, we evaluated GPUs and FPGAs as two major class of architectures, commonly used in parallel computing systems. We discussed the strengths and weaknesses of each architecture when running OpenCL applications. Table 5.1 compares GPU and FPGA devices, summarizing their key characteristics. Overall, the main strength of a GPU device is its ability to run millions of threads on a massive number of cores. Given this scale of spatial parallel thread execution, a GPU is able to hide memory latencies by switching thread blocks whenever a thread is waiting for data from memory. On the downside, however, the fixed general-purpose compute-unit of a GPU device is not as efficient as customized FPGA. Also, switching threads at a block granularity can actually degrade a GPU’s performance in thread divergent kernels.

The efficiency of FPGAs on the other hand, stems from pipelined execution of threads. The customized compute unit of an FPGA device can be more efficient than a GPU device with general-purpose compute unit. As Table 5.1 suggests, the main disadvantage of an FPGA device is its in-order thread execution. With in-order thread execution, an FPGA cannot hide long memory latencies. Therefore, every memory access can potentially create a pipeline stall in a FPGA device.

Table 5.1: GPU and FPGA characteristics comparison

<table>
<thead>
<tr>
<th>GPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Massive number of cores</td>
<td>Massive number of programmable blocks</td>
</tr>
<tr>
<td>Spatial parallelism</td>
<td>Both spatial and temporal parallelism</td>
</tr>
<tr>
<td>Fixed CUs</td>
<td>Customized CUs</td>
</tr>
<tr>
<td>Small Pipeline</td>
<td>Deep pipeline</td>
</tr>
<tr>
<td>Out-of-order thread execution</td>
<td>In-order thread execution</td>
</tr>
<tr>
<td>Hides memory latency with thread switching</td>
<td>Pipeline stalls on each memory access</td>
</tr>
</tbody>
</table>
In this chapter, we propose an new architecture, which we call a *Field Programmable GPU* (FP-GPU), that combines the strengths of both a GPU and a FPGA, that can execute OpenCL applications more efficiently. FP-GPU is a GPU-like architecture, maintaining the same CU structure and memory configuration. Also, each CU has its own L1 cache. However, instead of producing general purpose ALUs, each CU contains programmable logic resources to implement a specific OpenCL application. The OpenCL program will be compiled to an RTL data-path, and the RTL code will be used to program the CUs. The generated data-path will also be replicated to mimic SIMD behavior in GPUs. To hide memory latencies, the data-path has a thread controller that can switch threads when they are waiting for data. However, FP-GPU thread switching however works at a finer granularity than is supported in today’s GPUs.

To evaluate the FP-GPU architecture, we compare the performance and area of FP-GPU with an AMD Southern Islands GPU. To do so, we use the Multi2sim [56] and MIAOW [9] simulators. Section 5.2 reviews Multi2sim, an open-source GPU simulator, and MIAOW, an open-source GPU RTL implementation. Section 5.3 describes the FP-GPU architecture in more detail, and Section 5.4 discusses our experimental results. Section 5.5 discusses the advantages of the FP-GPU and some of the current limitations. Finally, Section 5.6 summarizes this chapter.

### 5.1 Background

To evaluate our proposed FP-GPU architecture and compare it with a general-purpose GPU (GP-GPU) architecture, we utilized two different open-source GPU simulators. Multi2sim [56] is used to provide performance analysis, and MIAOW [9] is used for our area comparison. This section reviews these two GPU simulators.

#### 5.1.1 Multi2sim

Multi2sim [56] is a free, open-source and cycle-accurate, simulation framework for CPU-GPU heterogeneous architectures. Multi2sim supports superscalar, multithreaded, and multicore CPUs, as well as multiple GPUs (AMD’s Southern Islands, and NVIDIA’s Kepler) architectures. The development model of Multi2sim is based on four independent software modules, as shown in Figure 5.1.

The first stage of Multi2sim’s simulation model is the Disassembler, which parses the executable binary file containing machine instructions. It decodes the instructions into Multi2sim’s
Figure 5.1: Four independent phases of multi2sim’s simulation paradigm.

internal representation that allows for interpretation of the instruction fields.

In the second stage, called the **Emulator** (functional simulator), the execution of the GPU program is modeled. The Emulator guarantees that the execution of the kernel produces the exact same result as its execution on the native device. To achieve this goal, the Emulator receives the instructions from the Disassembler, and dynamically updates the state of the program, instruction by instruction, until the program completes. To keep track of the program state, the Emulator updates the virtual memory image and the architected register file after consuming every single instruction.

The **Timing simulator** models hardware structures and keeps track of their execution time. This stage of Multi2sim’s simulation model provides a cycle-accurate simulation of the architecture by modeling pipeline stages, pipe registers, instruction queues, functional units, cache memories and others. The Timing simulator provides detailed hardware state and performance statistics, including execution time and cycles. It also generates a detailed simulation trace that can be used in later simulation steps.

The last stage, called the **Visual Tool**, is a graphical visualization framework. It consumes a compressed text-based trace file generated by the Timing simulator to provide the user with a cycle-based interactive debugging capability. Using the Visual tool, the user can observe memory access, instructions in flight, processor pipeline state, etc. The Visual tool can help the user to find performance bottlenecks in the program.

Multi2sim also provides a very flexible configuration of the memory hierarchy. The user can pass the configuration of memory hierarchy as a plain-text file to the simulator. The memory hierarchy can have any number of cache levels, with any number of caches in each level. Cache sizes and cache lines also can be specified in the configuration file, as well as the replacement policy.
and cache latency.

In this thesis, we use Multi2sim for two purposes. We use the Timing simulator to compare the performance of an AMD SI GPU with our proposed FP-GPU architecture. We also use Multi2sim’s memory hierarchy in our implementation for the FP-GPU. The FP-GPU implementation is described in Section 5.3.

5.1.2 MIAOW

MIAOW (Many-core Integrated Accelerator Of the Wisconsin) [9] is an open-source RTL implementation of the AMD Southern Islands GPGPU ISA. MIAOW supports a subset of the Southern Islands ISA (95 out of 400 instructions). In this thesis, we compare the RTL implementation of our proposed FP-GPU with the MIAOW GPU in terms of resource utilization (area). We compare the application-specific compute-unit of our FP-GPU design, while running a number of applications, with the general-purpose compute-unit of MIAOW GPU, shown in Figure 5.2.

Figure 5.2: MIAOW compute unit block diagram and its submodules.

5.2 FP-GPU High Level Architecture

Next, we describe the details of our Field-Programmable GPU architecture (see Figure 5.3). FP-GPU has the same CU configuration and memory hierarchy as a Southern Islands GPU architecture. It also contains an ultra-threaded dispatcher to distribute workgroups across the CUs. Similar to the SI GPU architecture, the FP-GPU has LDS memory and an L1 cache within
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

![Figure 5.3: FP-GPU high level architecture]

_each CU. These blocks, shaded green in Figure 5.3, are unchanged blocks borrowed from the AMD SI GPU.

The main difference between our FP-GPU design and the existing SI GPU is the design of functional units in the CUs. The entire functional unit of the SI CU, including the ALUs (scalar and vector) and instruction fetch and decode units, are replaced with a reconfigurable fabric in the FP-GPU. The reconfigurable fabric (shaded brown in Figure 5.3) can be programmed to run any given application, making the application-specific CU of the FP-GPU much more efficient than the general-purpose CU of the SI GPU. The efficiency of the FP-GPU compute-unit however, depends on how the OpenCL kernel is compiled into RTL. To improve the performance of FP-GPU, we apply the methods described in previous sections. When implementing an OpenCL kernel in RTL, we use pipelining to exploit Temporal parallelism. As we learned earlier in this thesis, Temporal parallelism is much easier to exploit than Spatial parallelism in FPGAs. We also use the Hardware Thread Reordering (HTR) method proposed in Chapter 4 to reduce the number of stalls during kernel execution. In section 5.3, we describe the RTL implementation of OpenCL kernels in more detail, and provide a sample implementation using the Binary Search kernel.

Our FP-GPU CU also contains a fixed microarchitecture (shaded blue in Figure 5.3),
which connects the customized reconfigurable portions of the CU to the fixed general-purpose elements of the design. The FP-GPU microarchitecture contains a Thread Dispatcher that receives the work-group size, and produces a thread id for the data-path. The Thread Dispatcher is connected to the data-path via a FIFO, producing a new thread id each clock cycle, even if the data-path is stalled. The data-path receives the thread ids and executes them in a pipelined fashion. When a thread is finished, the data-path sends its id to the Finish Detector module. The Finish Detector keeps track of completed threads and sets a flag when all threads have finished.

The FP-GPU CU also contains a Load/Store Unit (LSU) that connects the data-path to the memory hierarchy. The LSU is also responsible for thread switching. It receives memory requests from the data-path and sends requests to the memory hierarchy. Whenever data is ready for a pending request, the LSU sends the data to the data-path. The implementation of the LSU is described in more details in Section 5.3.

5.3 FP-GPU CU Implementation

Next, we present implementation details of the FP-GPU. We use a Binary Search kernel as an example to describe pipeline execution and thread switching support in the FP-GPU. Figure 5.4 shows the OpenCL code for the Binary Search kernel. The Binary Search kernel searches a sorted array to find a given number. The kernel will be launched in several iterations until the number is found. The array will be launched in several iterations until the number is found. The array will be launched in several iterations until the number is found. The array is subdivided into partitions, where each partition is searched by a work-item. During each iteration, every work-item compares the given number with the lower bound and upper bound of its partition. If the number is between the lower bound and upper bound, the work-item writes the lower bound and the upper bound of its partition to the output. This partition will be used as the input array in the next kernel execution.

The pipelined data-path generated for Binary Search kernel is represented in Figure 5.5. In this data-path, the Scalar Load Unit is responsible for requesting the global variables (global-LowerBound, findMe, and partitionSize) shared between all work-items. These variables will be requested and loaded before the dispatcher starts sending the thread ids. The pipelined data-path in Figure 5.5 consists of 5 stages. Stage 1 of the pipeline receives a thread id (i.e. thread i) in each clock cycle, calculates lowerBound for thread i, and sends a request for sortedArray[lowerBound] to LSU.

Once the data (lowerBoundElement) is ready for a thread (i.e. thread j), it will be sent to stage 2 as the input. In this stage, the lowerBoundElement will be compared with findMe. If
kernel void binarySearch(
    __global int *outputArray,
    __global int *sortedArray,
    __global int findMe,
    __global int globalLowerBound,
    __global int partitionSize)
{
    int tid = get_global_id(0);
    int lowerBound = globalLowerBound + partitionSize * tid;
    int upperBound = lowerBound + partitionSize - 1;

    int lowerBoundElement = sortedArray[lowerBound];
    int upperBoundElement = sortedArray[upperBound];

    if ( (lowerBoundElement > findMe) || (upperBoundElement < findMe) ) {
        return;
    } else {
        outputArray[0] = lowerBound;
        outputArray[1] = upperBound;
        outputArray[2] = 1;
    }
}

Figure 5.4: Binary Search OpenCL kernel
lowerBoundElement is greater than findMe, the execution of thread $j$ is finished. In this case, thread $j$ will be sent to Finish Detector. Otherwise, thread $j$ will be given to stage 3. Stage 3 calculates the upperBound and sends a load request for $\text{sortedArray}[\text{upperBound}]$. When there is more than one request from the data-path (i.e. both stage 1 and stage 3 are active), the arbiter arbitrates between the requests. The arbitration policy can be either round-robin or priority based, depending on the OpenCL application. Stage 4 performs the same operations as stage 2, and compares the $\text{sortedArray}[\text{upperBound}]$ with findMe. Finally, stage 5 sends the write requests to the Load/Store unit.

The Load/Store unit connects the data-path to the memory hierarchy (see Figure 5.6). In
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

Each clock cycle, the LSU receives a request from thread $i$, along with the context of thread $i$. It stores the context in the context table and transmits the passes the memory requests to the memory hierarchy. The LSU also stores the channel id (CID). When the data is ready for this request, the data will be transmitted on the same channel based on the stored CID. The LSU contains a Data Receiver block. The Data Receiver receives the data from the memory hierarchy and fills the proper row of the context table with the data that is received. Then, the LSU sets a ready flag in that row. The Data Transmitter block in the LSU keeps searching the context table to find ready data. When the Data Transmitter finds ready data, it retrieves the data (along with the data’s context), and sends it to the channel id associated with the data.

In our implementation, we utilized Multi2sim’s memory hierarchy implemented in C, rather than implementing the memory hierarchy in RTL. Instead, we focused on the implementation of the compute-unit, which is the main contribution of this portion of the thesis. In order to use the C implementation of the memory, we utilized the Verilog Procedural Interface (VPI). VPI is a C-programming interface for Verilog which provides consistent, object-oriented, access to the Verilog HDL. The connection of the LSU to the memory hierarchy using VPI is shown in Figure 5.6. LSU sends requests to a VPI module. The VPI module is a Verilog wrapper that initializes Multi2sim and accesses the memory hierarchy using C functions. In each Verilog clock cycle, the VPI module increments Multi2sim’s clock by one to keep the Verilog and Multi2sim synchronized. When a load request is finished, Multi2sim sends an event to the VPI module. At this moment, VPI module reads the data from a DRAM module and sends it to the LSU.

5.4 Evaluation

In this section, we evaluate our proposed FP-GPU architecture. We compare the FP-GPU with an AMD Southern Island GPU. In our comparison, we use the same memory hierarchy configuration for both architectures and compare one customized compute-unit of the FP-GPU with one general-purpose compute-unit of the SI GPU. We compare the two architectures in terms of the performance and area. In the following, we describe our experimental setup in more detail.

5.4.1 Experimental Setup

As discussed in previous sections, we implemented the FP-GPU compute-unit in Verilog for a number of benchmarks. To implement the data-path for each benchmark, we used the method
Figure 5.6: The Load/Store Unit
described in Chapter 4. We also implemented the FP-GPU micro-architecture including the LSU, Thread dispatcher, and Finish detector to connect the customized data-path to the memory hierarchy. We also used Multi2sim’s memory hierarchy, implemented in C. To connect the Verilog implementation and the C memory module, we utilized Verilog Procedure Interface (VPI) [18] to initialize Multi2sim and call Multi2sim’s C functions within our Verilog implementation. We used the same memory configuration as the SI GPU memory system in our simulations. Table 5.2 shows the cache hierarchy configuration used in our simulations.

To compare the area of the FP-GPU architecture with an AMD SI GPU, we synthesized the Verilog implementation for a Xilinx Virtex7 XC7VX485T FPGA device using Xilinx Vivado. Table 5.3 represents the specification of the Virtex7 FPGA device.

We implemented various OpenCL kernels to evaluate the FP-GPU compute-unit performance and area. We believe that a general-purpose GPU is a powerful architecture to execute kernels with simple control flow and no branch instructions. Since a general-purpose GPU executes the same instruction for a block of threads, performance drops whenever there is branch divergence in the thread block [62, 30, 61]. In the case of divergence in a thread block, a general-purpose GPU switches the whole block even if only one thread has to wait for memory. On the other hand, our pipelined customized data-path switches threads, using a finer granularity than GPU. In this case, the customized data-path switches only the thread which has to wait for memory with a thread that has its data ready. Therefore, it can handle thread divergence better than a GPU.

We chose Binary_search from the AMD SDK, BFS from rodinia benchmark [12], and SpMV (Sparse Matrix Vector Product) described in Chapter 4 as three kernels to explore the trade-offs of thread divergent applications. We also chose the vector_add and matrix_transpose kernels to evaluate our proposed architecture when executing a kernel with simple control flow and no thread divergence. The following subsections describe the performance and area comparison results.

Table 5.2: Cache hierarchy configuration

<table>
<thead>
<tr>
<th>Configuration</th>
<th>L-1</th>
<th>L-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>16 KB</td>
<td>128 KB</td>
</tr>
<tr>
<td>set-associativity</td>
<td>4-way</td>
<td>4-way</td>
</tr>
<tr>
<td>block size</td>
<td>64 B</td>
<td>64 B</td>
</tr>
<tr>
<td>Latency</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Policy</td>
<td>LRU</td>
<td>LRU</td>
</tr>
</tbody>
</table>
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

Table 5.3: Xilinx Virtex7 XC7VX485T FPGA device specification

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>75,900</td>
</tr>
<tr>
<td>Logic Cells</td>
<td>485,760</td>
</tr>
<tr>
<td>CLB Flip-Flops</td>
<td>607,200</td>
</tr>
<tr>
<td>Block RAM (KB)</td>
<td>37,080</td>
</tr>
</tbody>
</table>

Table 5.4: Number of instructions versus number of pipeline stages

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions is SI GPU</th>
<th>Pipeline stages in FP-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary_search</td>
<td>44</td>
<td>8</td>
</tr>
<tr>
<td>SpMV</td>
<td>57</td>
<td>8</td>
</tr>
<tr>
<td>BFS</td>
<td>88</td>
<td>11</td>
</tr>
<tr>
<td>Matrix_transpose</td>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>Vector_add</td>
<td>24</td>
<td>3</td>
</tr>
</tbody>
</table>

5.4.2 Performance Comparison

To evaluate the performance of the FP-GPU architecture, we compare the Verilog simulation of the FP-GPU (using Modelsim) with the simulation of an AMD SI GPU (using Multi2sim), for the five benchmarks. In all simulations, we simulate only one compute-unit. Also, in both cases, the compute-unit is connected to a memory hierarchy represented in Table 5.2. We use the total number of clock cycles as a performance metric to compare the two. In an AMD SI GPU, this metric represents the total number of cycles needed to fetch, decode, and execute the instructions of each kernel. In an FP-GPU, however, there are no explicit fetch and decode units since we do not have instructions to execute. In this case, the kernel is implemented in pipeline stages. Each thread passes through the pipeline and executes a part of the kernel function in each pipeline stage. From this point of view, we can compare the number of instructions (executed in SI GPU) with the number of pipeline stages (executed in FP-GPU) in each OpenCL kernel. As Table 5.4 suggests, the number of SI instructions is much higher than the number of FP-GPU pipeline stages for each benchmark. Also, the SI GPU needs to fetch and decode each instruction before executing it, while in our FP-GPU design, there is no such overhead. However, the benefits of the SI GPU over the FP-GPU are that the SI GPU executes each instruction on a batch of 64 threads (a wave-front), while each pipeline stage of the FP-GPU compute-unit contains only a single thread.

Figure 5.7 compares the performance of the FP-GPU with the SI GPU for five benchmarks. For the smaller work-group sizes FP-GPU outperforms SI GPU in all five benchmarks. When running 16 threads, FP-GPU is 4.1X, 2.6X, and 2.4X faster than SI GPU in Binary_search,
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

SpMV, and Vector_add kernels respectively. In this case, SI GPU is utilizing only 1 SIMD unit out of its 4 SIMD units. Therefore, the SI GPU is underutilized. As we increase the number of threads to 64, the SI compute-unit occupancy increases. For the vector_add and matrix_transpose kernels, which contains no thread divergence, increasing the number of threads to 64 does not increase the execution cycles. Increasing the number of threads in the FP-GPU, however, increases the execution cycles. In the FP-GPU, every thread needs to pass through the pipeline individually. As we increase the number of threads to more than 64, the execution time increases in both architectures. However, the increase in the FP-GPU is much higher than with the SI GPU, since the SI GPU executes the threads in wavefront granularity. Therefore, the FP-GPU speed-up drops as we increase the input size. The SI GPU is slightly faster than the FP-GPU when the input size is 1024.

For the Binary_search, BFS, and SpMV kernels, three kernels that contain thread divergence, the FP-GPU outperforms the SI GPU. In these cases, the SI GPU does not benefit from executing thread blocks, as it did with the vector_add kernel. In the case of the Binary_search kernel, only a single thread finds the findMe value in its own range and performs the actual computations. The other threads are just waiting for that thread to be executed. Then they can finalize their own computation. Increasing the input size in thread divergent kernels improves the SI GPU’s performance slightly. However, the FP-GPU is still faster than the SI GPU. On average, the FP-GPU is 3.9X and 2.2X faster than the SI GPU when executing Binary_search and SpMV kernels, respectively.

Figure 5.8 compares our FP-GPU with the SI GPU in terms of L-1 Cache misses. In all five benchmarks, the SI GPU has more cache misses than the FP-GPU. This is due to the fact that the SI GPU execute more instructions than the pipeline stages in the FP-GPU. Although the memory instructions will be executed on a thread block at the same time, each thread still needs its own data and accesses a different address. This will lower number of memory accesses, and so lowers the number of misses, which is another factor that improves the FP-GPU’s efficiency.

5.4.3 Area Comparison

To evaluate the area of the FP-GPU architecture, we synthesized the Verilog implementations of all five benchmarks. We used Xilinx’s Vivado 2017 and synthesized the designs for a Xilinx Virtex7 XC7VX485T FPGA device (see Table 5.3 for details). To compare the FP-GPU area with an AMD SI GPU, we utilized the numbers reported for the MIAOW GPU [9]. The numbers for MIAOW GPU however, is based on a SI compute-unit with only 1 SIMD unit and 1 SIMF unit,
Figure 5.7: FP-GPU and SI GPU performance comparison for five benchmarks
Figure 5.8: FP-GPU speed-up for five benchmarks
due to the resource limitation. To do a fair comparison, we multiplied the SIMD and SIMF resource utilization numbers by 4.

Figure 5.9 compares the FP-GPU with the SI GPU, in terms of resource utilization. For the simple vector_add kernel, the FP-GPU resource utilization is much lower than the SI GPU. For this kernel, the FP-GPU utilizes 69X fewer resources than the SI GPU. This means that, for the same area as a SI GPU, we can replicate the vector_add kernel 64 times to execute a batch of 64 threads instead of a single thread. This can improve the efficiency of FP-GPU significantly. This study, however, is left for future work.

For Binary_search, BFS, and SpMV, which are more complex kernels, our FP-GPU uses more resources. However, the resource utilization is still much less than the SI GPU. For Binary_Search and SpMV, the SI GPU uses 8.4X and 8.7X more resources than the FP-GPU compute-unit, respectively. This suggests that these two kernels can also be replicated 8 times to improve performance. However, the FP-GPU needs to handle the thread divergence when executing a batch of kernels in these cases. This is also left for future study.

Overall, FP-GPU is much more efficient than a SI GPU taking both performance and area into account. Figure 5.10 suggests that FP-GPU has much higher performance per area in all five benchmarks than a SI GPU.
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

Figure 5.10: FP-GPU performance per area improvement over SI GPU for five benchmarks
5.5 Discussion

In this section we discuss the advantages of FP-GPU over GPU as well as some of the current limitations of FP-GPU architecture which will be addressed in the future work. As mentioned in previous sections, FP-GPU contains a customized data-path which is more efficient than a general-purpose data-path in GPU architectures. The pipelined data-path in FP-GPU executes the kernel function in thread-level granularity as opposed to a GPU which executes the kernel in block-level (wavefront) granularity. This helps the FP-GPU to execute irregular kernels with thread divergence more efficiently. Thread-level execution also improves the cache hit rate in the kernels with high temporal and spatial locality. Overall, The FP-GPU outperforms the GPU significantly when executing irregular kernels.

For regular kernels with no thread divergence, a GPU with block-level execution is slightly faster than thread-level execution in FP-GPU. However, FP-GPU still has a better performance per area than GPU even for regular kernels. In order to improve the performance of FP-GPU in regular kernels, the customized data-path can be replicated. With a replicated data-path, FP-GPU can execute the kernel in a higher granularity (i.e. block-level). In other words, FP-GPU can be implemented to execute the kernel in Single Pipeline Multiple Threads (SPMT) fashion where each pipeline stage performs its function on a thread block. This behavior is similar to Single Instruction Multiple Threads (SIMT) execution in a GPU architecture. However, in memory-bound kernels, the performance is still limited by the memory bandwidth. The block-level execution in FP-GPU needs to be studied in more details in the future work.

The fully support of the OpenCL APIs is another area needs to be studied in the future to execute more benchmarks and expand the use of the FP-GPU architecture. Each OpenCL API needs to be implemented and evaluated. The implementation of some APIs seems to be simple. Barriers for example, can be implemented by adding a Finish Detector and a Thread Dispatcher as presented in Figure 5.11. Some other APIs such as OpenCL Pipes and Dynamic parallelism are more challenging. The full support of OpenCL is left for future studies.

5.6 Summary

In this Chapter, we proposed a novel architecture, called a Field Programmable GPU (FP-GPU). The FP-GPU is designed to combine the strengths of both a GPU and a FPGA. The result is an application-specific accelerator that can execute OpenCL applications more efficiently. FP-GPU
Figure 5.11: The OpenCL Barrier API and its implementation in FP-GPU

(a) Barrier API in a kernel  (b) Barrier implementation in FP-GPU
CHAPTER 5. ARCHITECTURAL OPTIMIZATION APPROACH

is a GPU-like architecture, adopting the same CU and memory configuration. FP-GPU’s compute-unit contains programmable logic resources to implement an OpenCL application. It also uses the GPU’s thread switching mechanism to hide the memory latency. However, the thread switching is implemented at a finer granularity than GPU, which switches the threads at a block granularity.

We demonstrated the efficiency of the proposed FP-GPU architecture on three OpenCL kernels: Binary_search, SpMV, and Vector_add kernels. Overall, the FP-GPU has a better performance than the SI GPU, while utilizing fewer hardware resources (less area). On average, our FP-GPU is 3.9X, 2.2X, and 1.8X faster than a SI GPU, while using 8.4X, 8.7X, and 69.7X fewer hardware resources when executing Binary_search, SpMV, and Vector_add kernels, respectively.
Chapter 6

Conclusions and Future Work

In this dissertation, we presented novel ideas on how to exploit thread-level parallelism on reconfigurable architectures. In Chapter 3, we presented our software approach to enhance OpenCL execution on FPGAs. In Chapter 4, we enhanced the OpenCL execution on a FPGA using a hardware approach. We proposed Hardware Thread Reordering method to support thread switching on FPGAs. In Chapter 5, we proposed an architecture, called the FP-GPU (Field Programmable GPU), that combines the strengths of both GPU and FPGA architectures to execute OpenCL applications more efficiently.

6.1 Contributions of this Thesis

Here, we summarize the contributions of this dissertation.

6.1.1 Source-level optimization

- We evaluated the effects of source-level decisions on the performance of OpenCL execution on FPGAs.
- We evaluated the potential benefits of leveraging the OpenCL Pipe semantic to accelerate OpenCL applications. We analyzed the impact of multiple design factors and application optimizations to improve the performance offered by OpenCL Pipes.
- Focusing on the Meanshift Object Tracking algorithm as a highly challenging compute-intensive vision kernel, we evaluated various grains of parallelism, from fine to coarse, on both a GPU and a FPGA.
CHAPTER 6. CONCLUSIONS AND FUTURE WORK

- We analyzed the correlation between OpenCL parallelism semantics and parallel execution on FPGAs. We evaluated the impact of different types of parallelism (spatial and temporal) exposed by OpenCL on the generated data-path by OpenCL-HLS tool.

- We showed the correlation between OpenCL programs and synthesized hardware on FPGAs.

- We showed the effectiveness of pipelining in FPGAs as opposed to spatial parallelism.

- This study guides OpenCL programmers to write FPGA-optimized code.

6.1.2 Synthesis optimization

- Based on our evaluations, we found out that the main disadvantage of OpenCL execution on FPGAs is the in-order thread execution. Especially in irregular kernels in-order thread execution degrades the performance significantly.

- We proposed a novel solution, called Hardware Thread Reordering (HTR), to boost the throughput of the FPGAs when executing irregular kernels processing non-deterministic and runtime-dependent control flow.

- We showed the effectiveness of out-of-order thread execution in irregular kernels by implementing three different irregular kernels.

- This study guides the synthesis tool developers to develop more efficient OpenCL to Verilog compilers.

6.1.3 Architectural optimization

- We showed the advantages and weaknesses of both GPU and FPGA architectures.

- We combined the benefits of both GPU and FPGA architecture proposing a novel architecture, called a Field Programmable GPU (FP-GPU), to execute OpenCL programs more efficiently.

- We combined the customized compute unit of FPGA with the GPUs memory hierarchy.

- We also used a fine-grained thread switching mechanism to boost the performance of our proposed architecture.
CHAPTER 6. CONCLUSIONS AND FUTURE WORK

- We implemented the proposed FP-GPU architecture, and compared it with a AMD Southern Islands GPU, evaluating the merits of this new approach in terms of the performance and area. We showed that FP-GPU outperforms GPU in terms of performance per area.

- This study guides the architecture designers to design more efficient hardware to execute OpenCL programs.

6.2 Directions for Future Work

There are many directions future work can take to build on the contributions of this thesis. In the source-level optimization techniques memory coalescing can be explored to reduce the number of memory accesses. Utilizing the memory bandwidth more efficiently can boost the performance since the memory bandwidth is the main bottleneck in FPGAs.

To continue the study on synthesis optimization one path is to develop an open-source OpenCL to RTL compiler to automate the data-path generation process. This can reduce the development time for the FP-GPU significantly. Also, delivering advanced tools will make it easier to study different techniques to create OpenCL data-paths for FP-GPU.

We believe that an application-specific data-path can outperform general-purpose GPUs in some cases. Specially, the performance of a GPU degrades when the application contains complex flow control with thread divergence. In these cases, a pipelined customized data-path can handle the thread divergence better than a GPU.

One path to continue the study on architectural optimization is to replicate the FP-GPU data-path to execute a batch of threads instead of only a single thread. This can improve the FP-GPU’s performance significantly in kernels with simple control flow.

Another path is to combine both general-purpose CUs and reconfigurable CUs in a single device. In this case, half of the CUs contain fixed ALUs, and the other half contain reconfigurable fabric. This device can utilize the benefits of both GPU and FP-GPU architectures.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


