EMPOWERING FPGAS FOR MASSIVELY PARALLEL APPLICATIONS

by

Suhas Ashok Shiddibhavi

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical and Computer Engineering

Charlotte

2018

Approved by:

Dr. Hamed Tabkhi

Dr. Ron Sass

Dr. Erik Saule

ABSTRACT

SUHAS ASHOK SHIDDIBHAVI. Empowering FPGAs For Massively Parallel Applications. (Under the direction of DR. HAMED TABKHI)

The availability of OpenCL High-Level Synthesis (OpenCL-HLS) has made FPGAs an attractive platform for power-efficient high-performance execution of massively parallel applications. FPGAs with their customizable data-path, deep pipelining abilities and enhanced power efficiency features are the most viable solutions for programming and integrating them with heterogeneous platforms. At the same time, OpenCL for FPGAs raises many challenges which require in-depth understanding to better utilize their enormous capabilities. While OpenCL has been mainly practiced for GPU devices, research is required to further study the efficiency of OpenCL written codes on FPGAs and develop a framework which can help categorize OpenCL parallelism potentials to the fullest. Aim of this work is to identify, analyze and categorize the semantic differences between the OpenCL parallelism and the execution model on FPGAs. As an end result we propose a generic taxonomy for classifying FPGAs based on available support from the OpenCL-HLS tool-chain. At the same time, new design challenges emerge for massive thread-level parallelism on FPGAs. One major execution bottleneck is the high number of memory stalls exposed to data-path which overshadows the benefits of data-path customization.

We introduce a novel approach for hiding the memory stalls on FPGAs when running massively parallel applications. The proposed approach is based on sub-kernel parallelism to decouple the actual computation from memory data access (memory read/write). This approach overlaps the computation of current threads with the memory access of future threads (memory pre-fetching at large scale). At the same time, this work proposes a LLVM-based static analyzer to detect the prefetchable data of OpenCL kernels with the capability to be integrated into commercial

OpenCL-HLS tools. This approach leverages the OpenCL pipe semantic to realize the sub-kernel parallelism. The experimental results of Rodinia benchmarks on Intel Stratix-V FPGA demonstrate significant performance and energy improvement over the baseline implementation using Intel OpenCL SDK.

The proposed sub-kernel parallelism achieves more than 2x speedup, with only 3% increase in resource utilization, and 7% increase in power consumption which reduces the overall energy consumption more than 40%.

To overcome the bottlenecks observed in the commercial OpenCL-HLS tool we propose an integrated tool chain for OpenCL-HLS. The new tool-chain is combination of already existing tool-chains for CPU, GPUs where LLVM acts as an intermediate machine level representation to translate from OpenCL to RTL. This open source tool chain is a proposed future extension of our work and we will be releasing it as an open source tool as a contribution of this thesis.

ACKNOWLEDGEMENTS

Here I would like to express my sincere gratitude to my adviser Dr. Hamed Tabkhi for the continuous support of my master's study and research, for his patience, motivation, enthusiasm and immense knowledge. His guidance helped me in all the time of research and writing this thesis. I would have not imagined having a better adviser and mentor for my master's thesis study.

Besides my advisor, I would like to thank rest of my thesis committee: Dr. Ron Sass and Dr. Erik Saule for their encouragement and insightful comments.

Finally, I would like to thank my parents, my sister, friends and tecsar team for cheering me up and stood by me through good times and bad.

DEDICATION

This thesis is dedicated to my family who has been constant source of support encouragement during the challenges of graduate school and life. I am thankful for having them in my life whose good examples have taught me to work hard for the things that I aspire to achieve.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALU	Arithmetic Logic Unit.
API	Applications Programming Interface.
CPU	Central Processing Unit.
CU	Compute Unit.
DP	Data-Path.
FPGA	Field Programmable Gate Array.
GPU	Graphical Processing Unit.
HLS	High-level synthesis.
HPC	High Performance Computing.
ISA	Instruction Set Architecture.
LLVM	Low Level Virtual Machine.
MCUMDP	Multiple Compute Unit Multiple Data-Path.
MCUSDP	Multiple Compute Unit Single Data-Path.
OpenCL	Open Compute Language.
PE	Process Elements.
PoCL	Portable OpenCL.
RTL	Resister Transistor Logic.
SCUMDP	Single Compute Unit Multiple Data-Path.
SCUSDP	Single Compute Unit Single Data-Path.

SDK	Software Development Kit.
SIMD	Single Instruction Multiple Data.
SIMT	Single Instruction Multiple Thread.

CHAPTER 1: INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are one of the major architectures that can significantly benefit from Open Computing Language (OpenCL) abstraction and unification. Open Computing Language (OpenCL) for FPGAs provides an opportunity to execute massively parallel applications on FPGAs with minimum programming effort. OpenCL High-Level Synthesis (OpenCL-HLS) enables parallel programmers to construct a customized data-path that can best match an application, without getting drowned by implementation details. At the same time, an application developed in OpenCL can better guide synthesis tools by explicitly exposing parallelism. The two major FPGA companies (Altera and Xilinx) have already released tool-chains for OpenCL-HLS on FPGAs [1, 2]. We also observe the integration of FPGAs in many platforms from Amazon cloud web service [3] to Microsoft Brainwave project (for real-time AI) [4] down to Xilinx Zynq platforms [5, 6] for real-time stream processing at the edge.

Despite significant potential, OpenCL for FPGAs introduces a set of new design challenges. The challenges mainly stem up from the fundamental architectural differences between CPUs/GPUs and FPGAs. CPUs and GPUs both are Instruction Set Architecture (ISA) based machines while FPGAs are reconfigurable platforms that lack ISA level of abstraction. FPGAs in short do not have the concept of program counter (PC) which is an essential component of CPUs/GPUs. FPGAs efficiency stems from a customized data-path, operation-level parallelism and also the ability to exploit deep pipelining or temporal parallelism. GPUs on the other hand rely on exploiting concurrent execution of massively parallel threads on many cores or spatial parallelism. Throughput-oriented platforms (e.g., GPUs) often rely on built-in schedulers to manage concurrent thread execution at massive scale (over many cores) thus hiding memory latency. With lack of run time scheduler, memory stalls on threads 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 the data. This creates significant data-path under-utilization for complex OpenCL kernels with high memory access demand. Under utilization of the memory bandwidth is another side effect of these memory stalls. Figure. 1.1 and Figure. 1.2 illustrates the memory stalls, and memory bandwidth utilization for the Hotspot and BFS applications from the Rodina benchmark suite [7], running on a Stratix-V FPGA [1]. Overall, 75% of memory accesses in Hotspot, and 40% in BFS, generate stalls in the pipeline. This leads to significantly low bandwidth utilization (9% in Hotspot and 7% in BFS). To enable efficient execution of massively parallel applications on FPGAs, reducing the memory bottleneck is paramount.

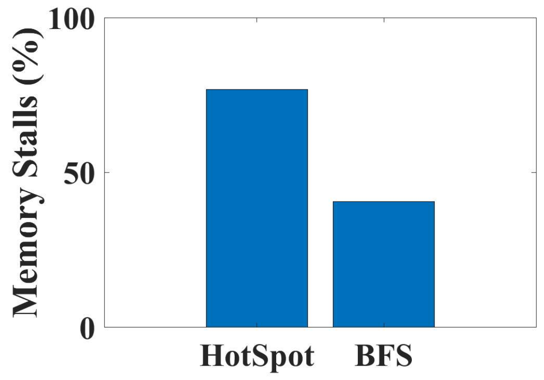


Figure 1.1: Memory stalls for Hotspot and BFS applications

1.1 Problem Statement

Since OpenCL was developed primarily for GPUs, their has been a lot of focus on OpenCL tuning for GPUs as these devices have dominated the heterogeneous computing market. In contrast 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 depth. Now that FPGAs have gained interest in both academia and the

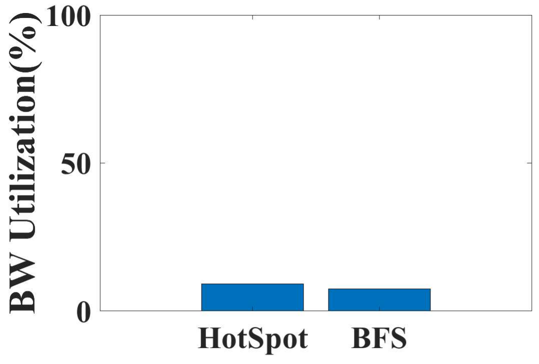


Figure 1.2: bandwidth utilization for Hotspot and BFS applications

industry, FPGA programmers and developers are responsible for understanding the impact of source-level and synthesize decisions on the generated architecture. This demands for an explicit understanding of the FPGA architecture when mapped to OpenCL execution model. Along with it a generic framework such as in CPUs/GPUs [8] is desirable so as to formalize the parallelism potentials of the FPGA.

OpenCL for FPGAs is still in its infancy. Current approaches primarily focus on optimizing computation-path when running OpenCL kernels on FPGAs [9, 10, 11]. In particular, we observe application-specific optimization (for deep learning and neural network applications) to achieve the maximum efficiency and minimize the memory stalls on FPGAs [12, 13, 14]. However, there is less focus on the major bottleneck of memory stalls. There is lack of a systematic solution to mitigate or entirely remove memory stalls when running massively parallel applications on FPGAs.

This work presents a scalable automated mechanism to decouple the memory access from the actual computation to hide the memory latency of massively parallel applications running on FPGA devices. In this regard, the work introduces the concept of sub-kernel parallelism to create concurrency between the computation and memory access among the OpenCL threads. It breaks down the entire kernel to memory access and computation sub-kernels communicating through OpenCL "Pipe" construct. The work also proposes a novel LLVM-based static memory access analysis for automatic detection of complex variable access patterns (beyond a constant stride) and managing the data dependencies across the sub-kernels. LLVM analysis detects and separates prefetchable data access (can be decoupled from the computation) and non-prefetchable data access (runtime dependent) patterns. Our experimental results over eight Rodinia kernels [7] running on Intel Stratix-V FPGA [1] demonstrates 2x speedup with 40% energy reduction compared to baseline implementation.

To the best of our knowledge, this work, as it stands, is the very first approach that proposes a generic synthesizable solution by introducing LLVM-based sub-kernel parallelism for decoupling access/execute and thus large-scale prefetching when running massive parallel applications on FPGAs. In a nutshell, the contributions of this work are:

1. Introducing sub-kernel temporal parallelism as a new level of parallelism for OpenCL applications running on FPGAs.
2. A systematic approach for decoupling memory access from computation when running massively parallel applications on FPGAs.
3. A Novel LLVM-based memory access analysis to extract the memory dependencies and detect the prefetchable data.

1.2 Contributions

The goal of this thesis is to develop a novel design methodology to explore the execution of massively parallel applications on reconfigurable devices. Here we outline the contributions of this thesis.

- To explore the impact of source-code decisions we propose a new taxonomy at OpenCL abstraction for FPGAs execution efficiency. Here we try to achieve maximum spatial parallelism potential to inherit the deeply pipelined architecture of FPGAs.

- To overcome the bottleneck of memory stalls exposed to data-path we proposed a novel solution, called sub-kernel parallelism for hiding the memory access latency on FPGA devices when running massively parallel applications.

- To better explore and understand the execution behavior of FPGA devices and overcome the bottlenecks of commercial HLS tools we propose an open source OpenCL-HLS tool, integration of existing tools to produce an OpenCL to RTL schematic which make use of LLVM compiler as intermediate machine level representation.

1.3 Thesis Outline

The outline of this thesis is as follows. Chapter 2 reviews the background needed for this study. It reviews OpenCL execution model on FPGAs. Chapter 3 briefly overviews related work in the field of OpenCL for FPGA devices. Chapter 4 presents our first contribution for taxonomy of spatial parallelism on FPGAs. Chapter 5 describes the second contribution of sub-kernel temporal parallelism architecture approach to hide memory access latency of massively parallel applications on FPGA devices. Chapter 6 chapter proposes open source OpenCL-HLS tool to explore the architectural behavior of FPGAs in real-time. finally Chapter 7 concludes thesis and brief details about future work.

CHAPTER 2: BACKGROUND

The Open Computing Language (OpenCL)Figure. 2.1 is a heterogeneous programming framework to develop applications that execute across various devices from different vendors [15, 16]. 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.

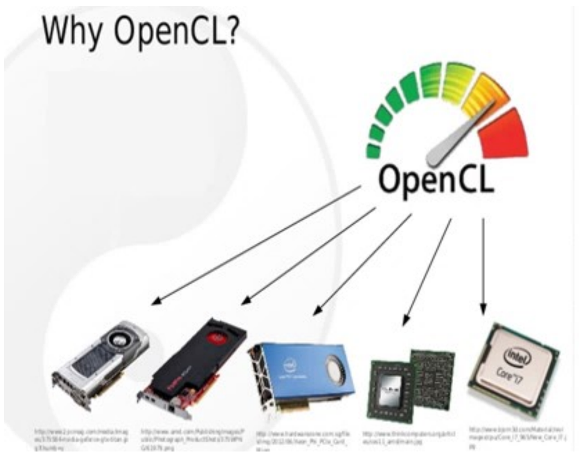


Figure 2.1: OpenCL programming models

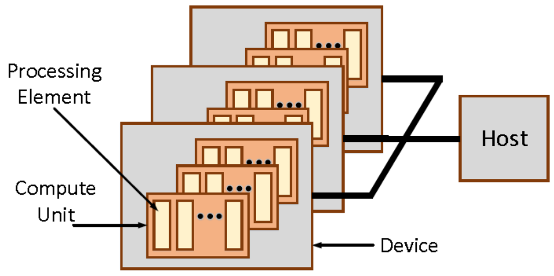


Figure 2.2: OpenCL Platform Model

The OpenCL platform model (see Figure. 2.2) 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.

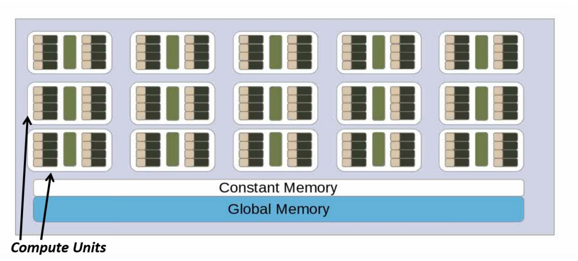


Figure 2.3: OpenCL Device Model

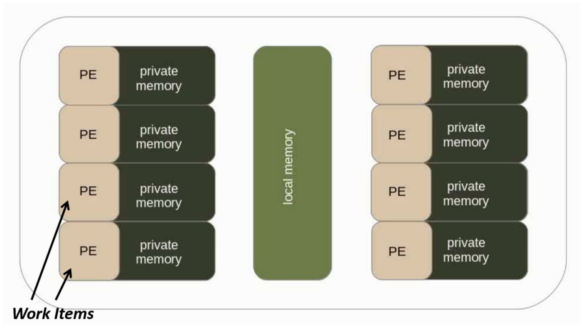


Figure 2.4: OpenCL Compute Unit model

Figure. 2.3 shows the internal architecture of OpenCL device. OpenCL device contains compute units, global memory and constant memory. Each compute unit internally contains Process Elements(PE) with its own private memory as show in Figure. 2.4. Local memory is shared across multiple process elements within a compute unit.

Figure. 2.5 represents the OpenCL execution model. The unit of parallelism in

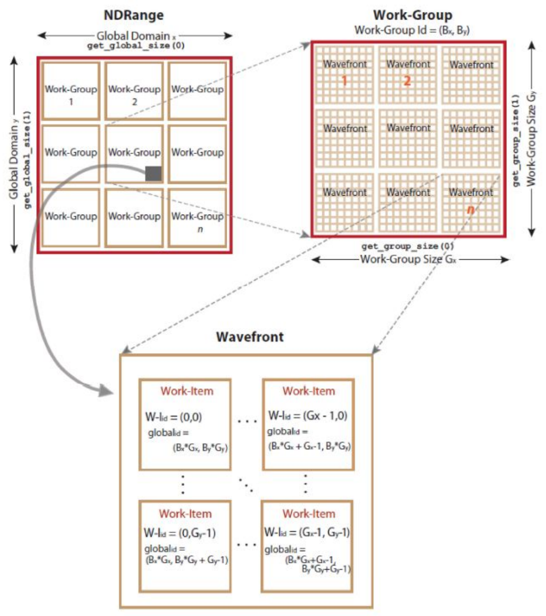


Figure 2.5: OpenCL Execution Model

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. 2.5, 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 [17, 18, 19]. 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 model of FPGA. In the following, we review the OpenCL execution model of FPGA. The aim is to highlight the impact when developing OpenCL kernels for FPGAs.

2.1 OpenCL Execution on FPGAs

Parallelism concepts on CPUs and GPUs have been extensively studied before [8, 20, 21]. While these techniques are able to capture instruction and data level parallelism, they were developed for ISA specific architectures. Reconfigurable platforms like FPGAs are ISA independent architectures and the same classifications cannot be justifiably applied on them. Recent improvements in OpenCL-HLS have opened up new opportunities for the FPGA programmers to work with massively parallel applications on FPGAs and utilize the capabilities of the FPGAs to their full potential. While there are quite a few techniques developed to aid the programmer in this field, most of them still lack intuition and a solid understanding for matching such programming capabilities on the FPGA architecture.

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. Threads execute in an in-order fashion over a deeply pipelined data-path. In the ideal situation, when there is no pipeline stall, one thread (work-item) enters the pipeline per clock cycle, and one thread completes its execution and exits the pipeline.

Previous studies have proposed OpenCL-HLS tools to execute OpenCL programs on FPGAs [1, 2, 22, 23]. In our experiments we use ALtera OpenCL SDK, a widely used commercial OpenCL-HLS tool [1]. OpenCL HLS primarily identifies data level and task level as the two core parallelism techniques[24]. We go a step further by formalizing these ideas in the context of FPGAs.

FPGAs Memory wall is one of the primary limitation of fine-level temporal parallelism on FPGAs. With no runtime single-cycle thread scheduling (as in GPUs), memory stalls are directly exposed to the execution path. FPGAs also lack sophisticated data caching and advanced hardware prefetching (as in CPUs) to reduce or hide memory access latency. Often, OpenCL-HLS tools add large numbers of delay buffers to partially hide the memory access latency.

One common solution to mitigate memory stalls is to use local memory (similar to scratchpad memory). With local memory allocation, OpenCL-HLS tools move the data from global memory (off-chip) to local memory (on-chip) prior to launching at workgroup granularity (groups of thread sharing same local memory, similar to thread block in CUDA programming model). Local memory can potentially reduce the large number of memory stalls. On the negative side, the memory copy time for moving the next workgroup data from global memory space to local memory space is directly exposed to execution.

2.2 Experimental Setup

Table 2.1: System characteristics used for study

Host	Intel(R) Core(TM) i7-7700K
Host clock	4.2 GHz
FPGA Family	Stratix-V
FPGA Device	5SGXMA7H2FE35C2
CLBs	234,720
Registers	939K
Block Memory bits	52,428,800
DSP Blocks	256

There are lots of benchmark suites available in market for heterogeneous devices and we use standard OpenCL kernels from Rodinia benchmarks suite[7]. All the implementations are synthesized on Stratix-V FPGA. Table 2.1 lists the parameters of our FPGA platform. We use Intel SDK for OpenCL [24] for compiling and synthesizing OpenCL code. We also use Intel SDK-OpenCL(AOCL) profiler to obtain the detailed execution results. AOCL profiler collects kernel performance data, bandwidth efficiency of global memory, stalls (Percentage time that memory access caused the stall in kernel execution).

CHAPTER 3: Related Work

A framework for exploring instruction and data level parallelism on ISA based architectures i.e, CPUs and GPUs has been proposed long back [8] by Michael J Flynn. Many other works [20, 21] have also been studied to provide an extension over Flynn's taxonomy for different multiprocessor architectures. In contrast to existing categories for ISA based machines, the execution model of FPGAs make them different which leads to a dire need for a classification of their own. OpenCL-HLS gives the programmer such flexibility by introducing various optimization techniques. In this work we formalize these ideas to propose a taxonomy that opens up new opportunities for the programmer to better utilize parallelism benefits on FPGAs.

Performance optimization using OpenCL framework for massively parallel applications on CPUs/GPUs has become very popular [25, 26, 9]. These approaches primarily focus on the application-specific performance optimization techniques [10, 11], or basically making a performance comparison between FPGAs and GPUs.

The approach of decoupling [27] has been well elaborated by Dr.Smith. Which is building blocks and has been explored and analyzed more on different variety of devices [28, 29, 30]. The approach of decoupling has been the major contribution for understanding the access latency of memory in various model of devices.

The data request in the devices for computation has been major bottleneck as the cost of data transfer from memory is very expensive. The immediate solution for this problem is making data available at the time of execution. And simple approach to furnish this data is called as prefetching [31, 32]. As the name suggests prior to the requirement understand the behaviour and keep the data available for device for computation. There has been a lot of research done. Based on the behavior prefetching can be acheived by hardware called as hardware prefetching [33, 34, 35] or software called as software prefetching [36, 37, 38]. There has been relevant experiments already done to merge both hardware and software prefetching to achieve maximum performance [39]. Also recently, there has been more focus towards the cache prefetching approaches [40, 41].

Availability of OpenCL-HLS for FPGAs poses many interesting research questions to OpenCL-HLS designers and system architects. The energy efficiency benefits of FPGA devices along with the ability to employ pipelined parallelism properties make the evaluation of OpenCL kernels on these devices very fascinating [10, 42]. Many researches [7, 43, 44] have been conducted on OpenCL programming capabilities to improve FPGAs efficiency. Further, [45, 46, 47] have worked on exploiting the parallelism on FPGAs with Opencl attributes as well as by suggesting new architectural modifications to improve performance. In particular, we observe a significant interest in accelerating neural networks and deep learning applications on FPGAs using OpenCL programming abstraction [12, 13, 14, 48, 49].

In the reconfigurable computing community, multi-thread execution on FPGAs is a very rich topic [50, 51, 52, 53, 54, 55, 56, 57, 58]. The primary focus is on context switching and partial reconfigurability across multiple applications. Few researchers have also studied multi-threaded execution on single kernels [59, 60]. However, there is a less focus on addressing the execution challenges of massively parallel applications on FPGAs when many threads are sharing same kernel (data-path) over many data (as in OpenCL).

Overall, despite many interesting researches in the field, OpenCL for FPGAs is still at its early stages. There is a lack of in-depth analysis and generalized solutions to enhance the OpenCL execution efficiency on FPGA devices. The major focus have been on creating an efficient application-specific data-path rather than removing the bottleneck of memory latency. Here, we propose a generalized design solution (borrowed from throughput-oriented design principles) to overlap memory access and computation at massive scale to hide memory latency and avoid memory stalls.

CHAPTER 4: Taxonomy

Primary motivation behind this work lies in identifying and classifying the OpenCL HLS programming techniques and mapping them into a framework which we believe could provide better programmability to FPGA programmers and help formalize OpenCL research in near future. To identify all possible spatial parallelism benefits and maximize parallelism potentials on FPGAs we propose a taxonomy that provides a clear classification based on the type of parallelism that can be employed on the FPGAs. Our proposed taxonomy is classified into four categories. Figure. 4.1 shows the grid depicting our taxonomy. OpenCL work-groups typically get mapped to compute unit while work-items get mapped to data-path. We show one through many compute units on the X-axis of the grid while variation of data-paths are shown on the Y-axis.

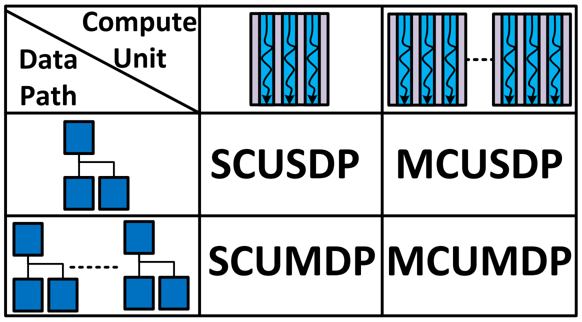


Figure 4.1: Taxonomy Grid

The proposed taxonomy is classified into four categories. Figure. 4.2 through Figure. 4.5 shows all possible classifications of OpenCL parallelism that we have proposed in our work.

4.1 Single Compute Unit Single Data-Path [SCUSDP]

The Single Compute Unit Single Data-Path [SCUSDP] is the default synthesis generated by the HLS tool. We introduce this as a starting point for comparison with other categories. The SCUSDP which has its own memory, load-store and control units. SCUSDP does not utilize any spatial parallelism capability when implemented on the FPGA although the data-path generated across the compute unit does enjoy temporal pipelining benefits.

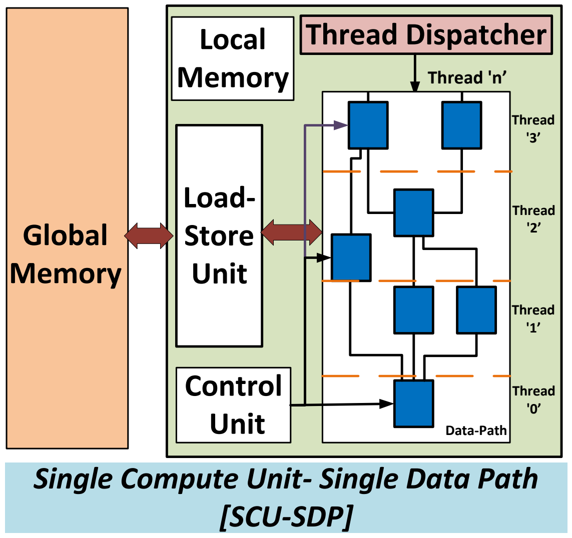


Figure 4.2: Single Compute Unit Single Data-Path

4.2 Single Compute Unit Multiple Data-Path [SCUMDP]

The Single Compute Unit Multiple Data-Path [SCUMDP] (Figure. 4.3) is a finer grained classification that involves replicating data-paths inside a single compute unit without replicating the thread dispather and the load/store units. By replicating the data-path, the CU is able to execute multiple threads at the same time. Semantically, SCUMDP can be considered similar to the Single Instruction Multiple Threads (SIMT) model employed in GPUs. This also makes it possible to implement spatial parallelism on top of the temporal parallelism. Moreover, since the same compute unit is involved each CU has multiple ALUs to execute the same instruction across multiple threads over multiple data while sharing the same control signals.

One possible downside of SCUMDP is the lock-step execution between the replicated data-paths, which in turn will introduce 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. This requires the data for all the threads to be available, otherwise, all the threads will be stalled. This limits the overall performance improvement.

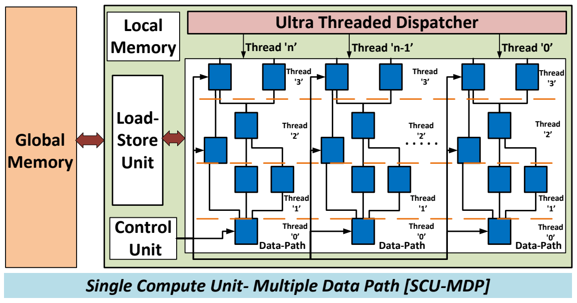


Figure 4.3: Single Compute Unit Multiple Data-Path

4.3 Multiple Compute Unit Single Data-Path [MCUSDP]

The Multiple Compute Unit Single Data-Path [MCUSDP] (Figure. 4.4) is a coarser level granularity of implementing spatial parallelism over temporal parallelism. This approach uses the idea of replicating an entire CU including the entire data-path, thread dispatcher, 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. Thus within each CU control signals work independently.

However MCUSDP is not often feasible for complex kernels with large code size due to limitation placed on FPGA resources. Compared to SCUMDP, MCUSDP is thus less efficient in terms of resource utilization. In addition to this, the most important drawback in MCUSDP 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.

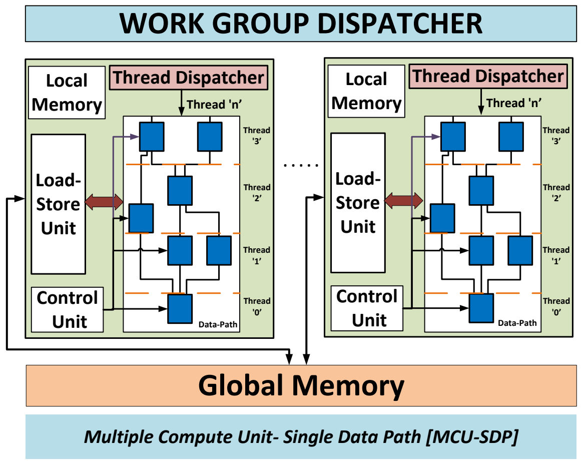


Figure 4.4: Multiple Compute Unit Single Data-Path

4.4 Multiple Compute Unit Multiple Data-Path [MCUMDP]

Employing multiple compute units leads to contention for global memory which in turn might lead to undesired memory access patterns affecting performance. This behavior however can be changed by vectorizing the application along with replicating CUs. This can be attributed to the fact that vectorizing a kernel gives an opportunity to the HLS tool to apply memory coalescing [61]. This concept is utilized in Multiple Compute Unit Multiple Data-Path [MCUMDP] (Figure. 4.5) which is a hybrid model developed from the previous two classifications. This technique not only makes use of pipe-lining potentials but also exploits massive parallelism across each compute unit. This is the maximum parallelism potential that can be exploited on the FPGAs. MCUMDP strives to offer a balance between the major bottlenecks of resource utilization and memory contention in MCUSDP along with stalls observed due to lock step execution in SCUMDP.

4.5 Evaluation

Spatial Parallelism We used eight standard applications from the Rodinia benchmark suite[62] for our experimental evaluation purposes. They namely are Nearest Neighbors, Srad_base(Srad extract application), Gaussian, B+Tree, Needleman

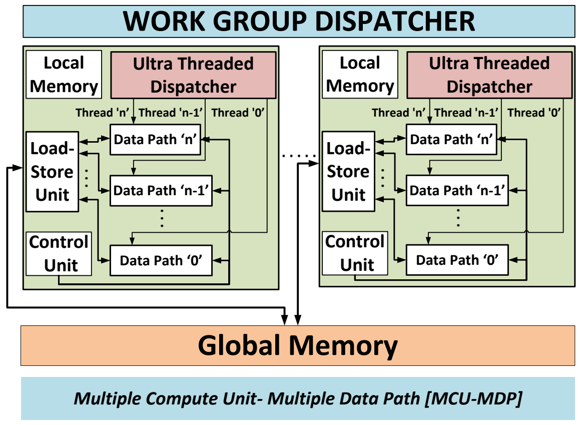


Figure 4.5: Multiple Compute Unit Multiple Data-Path

Wunsch(NW), Breadth First Search(BFS), Hotspot and Stream cluster. We have used the Intel SDK for OpenCL[24] based on OpenCL version 1.0 for compiling the OpenCL code. Our FPGA implementations are synthesized on the Stratix-V FPGA while we have used the AMD Firepro W7100 device for our GPU implementation. Table 2.1 shows the system parameters of our FPGA and GPU platform in more detail.

4.5.1 Taxonomy performance results

In our experiments we used the OpenCL attributes to set the number of compute units and data-paths along with sizing the work group dimensions for the purpose of our design. For the SCUSDP and the MCUSDP configurations we were able to synthesize all eight selected applications. On the other hand for SCUMDP and MCUMDP classifications we could synthesize only four applications namely, NN, Srad_base, Gaussian and B+Tree. The use of multiple data-path that is common to both these categories suffers from an inherent disadvantage of requirement of lock-step execution pattern that limits the data-path replication to be applied to simple kernels with no data dependent or conditional branches. The OpenCL-HLS tool fails to vectorize such applications.

Figure. 4.6 through Figure. 4.9 show the performance improvement(increase in

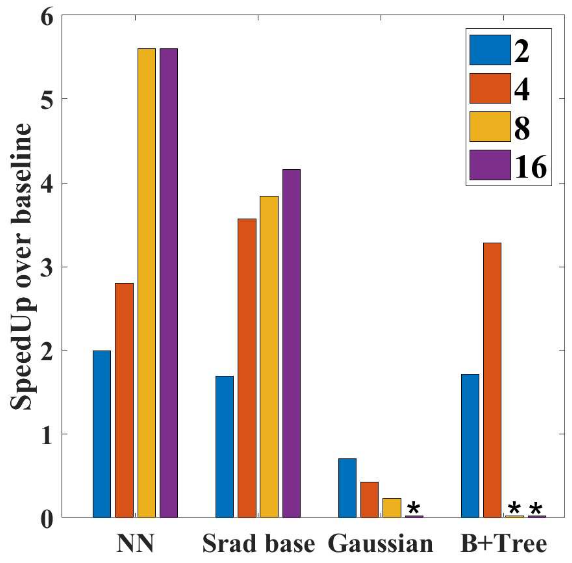


Figure 4.6: Speed Up of SCUMDP

times speedup over baseline) for each of the taxonomy categories for every single application. Legends of SCUMDP and MCUMDP indicate the number of data-paths from '2' through '16' while that of MCUSDP indicates number of compute units from '2' through '8'. The graphs for compute units and data-paths labels marked 'asterisk(*)' are the ones which could not be compilable due to FPGA resources reaching its maximum possible capacity.

Speed up due to SCUMDP can be attributed to two basic reasons:- 1. Vectorizing the kernel allows the creation of multiple data-paths that can execute in a single instruction multiple thread (SIMT)fashion. 2. It also avails an opportunity for the HLS tool to introduce memory coalescing to further increase efficiency. On the other hand performance either saturates or decreases owing to the increase in number of stalls with every additional data-path due to lock step execution model. Apart from this the application itself can effect the parallelism potential.

SCUMDP in Figure. 4.6 shows a maximum speed up of 5.6X over baseline for nearest neighbor application when employing '8' data-paths while saturating for '16' data-paths. The speed up for Srad_base and B+Tree show similar trends as well.

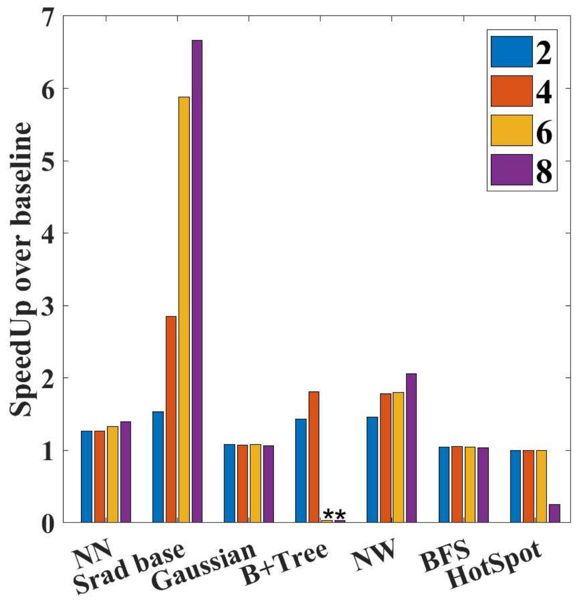


Figure 4.7: Speed Up of MCUSDP

The Gaussian application on the contrary shows reducing trends, this is due to the fact that low temporal locality hinders its ability to take full advantage of memory coalescing thereby reducing its overall performance despite employing data-path parallelism. Figure. 4.10 and Figure. 4.14 show a similar correlation between the results obtained. While bandwidth utilization tends to improve with increasing number of data-paths(maximum being at 10.2X for Srad base with 16 data-paths) the effect is more or less limited to increase in stalls due to lock step execution observed in SCUMDP.

MCUSDP in Figure. 4.7 shows a maximum speed up of 6.7X over baseline for srad_base application after replicating '8' compute units. The speed up however isn't that effective for all other applications maintaining an average of about 1.5X speed up. Overall more resource utilization in MCUSDP accounts for more memory contention leading to reduced performance for most of the benchmarks. Figure. 4.11 and Figure. 4.15 provide similar characteristics with increasing number of compute units however with more number of stall percentage compared to SCUMDP attributed to memory contention.

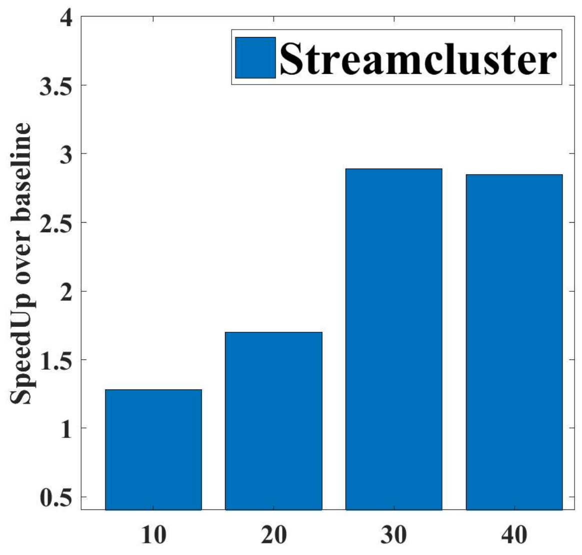


Figure 4.8: Speed Up of MCUSDP for Stream-Cluster

The stream cluster application for MCUSDP in Figure. 4.8 offered negligible improvements at low number of compute units while showing a considerable improvement only on increasing the number of CUs from 10 through 40 while achieving a maximum speed up of 2.89X over baseline before getting limited by resources.

Figure. 4.9 shows the performance improvement of MCUMDP. In this case we could experiment with two compute units and a maximum data-path of four. Anything over this was not compilable due to the major bottleneck of FPGA resources. However, the performance improvement observed is once again a trade-off between stalls (Figure. 4.17) due to memory contention(MCUSDP) and lockstep execution(SCUMDP). We attained a maximum speed up of 5.6x for the srad_base application using this approach.

4.5.2 FPGA vs GPU comparison results

The following two-fold approach was used to quantify and compare the performance of FPGA and GPU in terms of normalized performance/watt. 1. We used the PIN Tracer tool[63] to find the total number of instructions, memory accesses and branch accesses per application. We then calculated the total number of instructions

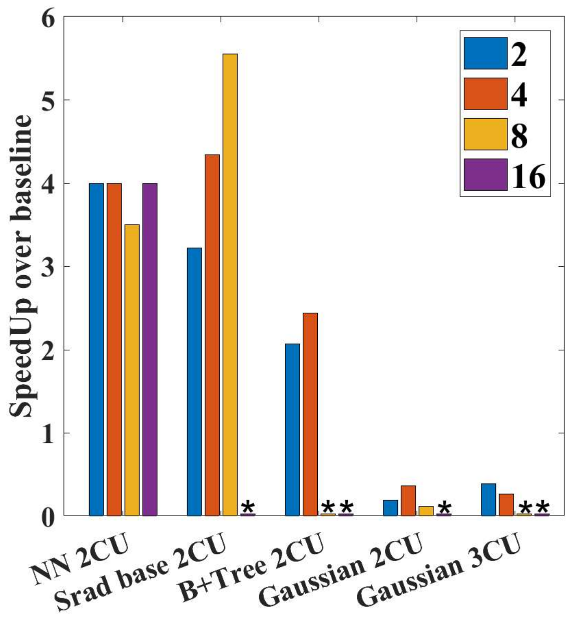


Figure 4.9: Speed Up of MCUMDP

utilized for computation related access as in Equation (4.1). Table 4.1 gives us a comprehensive idea of all the instruction accesses.

Table 4.1: Number of instructions per application

Application	**Instructions**		
	Memory Access	**Branch Access**	**Computation**
NN	971583	842826	2825645
Srad_extract	2534227	1613410	4149347
B+Tree	791962001	1431982510	2990223309
NW	37965825	12701717	63215476
BFS	131783847	99622929	135156055
HotSpot	9599430	5732485	9888922
Streamcluster	1086883070	561570230	2041351051

$$\begin{aligned}\textbf{No. of computation access instructions} &= \textbf{No. of total instructions} - \\ &(\textbf{No. of memory access instructions} + \textbf{No. of Branch access})\end{aligned} \tag{4.1}$$

2. Next, we used the CodeXL Power Profiler version 2.5 [64] to give us the GPU

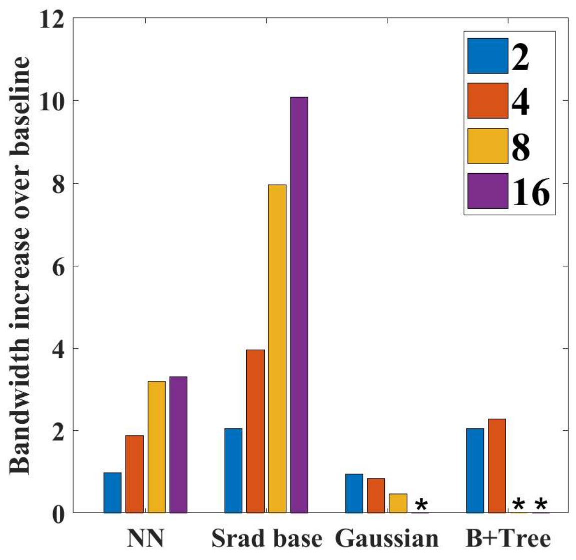


Figure 4.10: Bandwidth of SCUMDP

power consumption per application. For the FPGA power we used the Intel SDK-OpenCL(AOCL) profiler to collect kernel stalls(%) information and bandwidth(MBs) utilization. We then used the Stratix® IV and Stratix® V PowerPlay Early Power Estimator tool to find the total thermal power consumption(Watts) of the device. The Total thermal power of the device is calculated as a sum of the Static Power(P_{STATIC}), Dynamic Power(P_{DYNAMIC}) and I/O Power [61]. Static power is the leakage power dissipated from the chip and independent of user clocks. The I/O power is the DC bias power and transceiver DC bias power. The Dynamic power is calculated from internal nodes changing logic levels internal to the device in the form of equivalent lumped capacitance's, it is given as below:-

$$\begin{split} DynamicPower = V_{\text{CCINT}} \times \sum I_{\text{CCINT}}(&LE/ALM, RAM, \\ &DSP, PLL, Clocks, HSDI, Routing) \end{split} \tag{4.2}$$

where power is calculated from dynamic power dissipated across each of Adaptive Logic Modules(ALMs), RAM Blocks(RAM), DSP blocks(DSP), Phase Lock loops(PLLs),

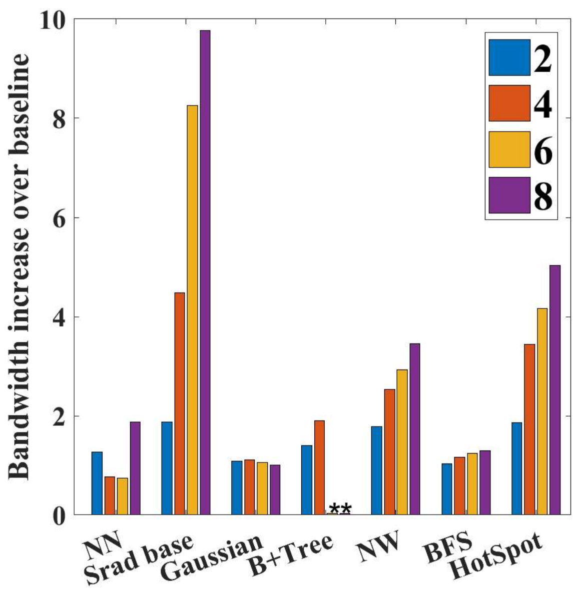


Figure 4.11: Bandwidth of MCUSDP

Clock, High Speed Differential I/Os(HSIO) and associated routing modules.

$$Performance/Watt = \frac{No.of computation access instructions/sec}{Power consumption(Watts)} \tag{4.3}$$

From these results we finally calculate the Performance/Watt for every individual application as shown in Equation (4.3). We believe that Performance/Watt is an ideal standard for comparing FPGAs vs GPUs on the same scale. We normalize the values obtained for curve fitting purposes. Figure. 4.18 shows a comparison between Baseline FPGA vs Best FPGA performance obtained after applying our taxonomy vs GPU. We observe that after using our taxonomy we get improved FPGA performance as against baseline FPGA for all the cases. However the FPGA performs fairly better than GPU only for three applications which have more number of regular accesses and are embarrassingly parallel. NN, Srad base and NW exhibit such properties. B+Tree, BFS, Hotspot and Streamcluster on the other hand have a large number of random accesses and suffer from stalls due to unavailability of data affecting the

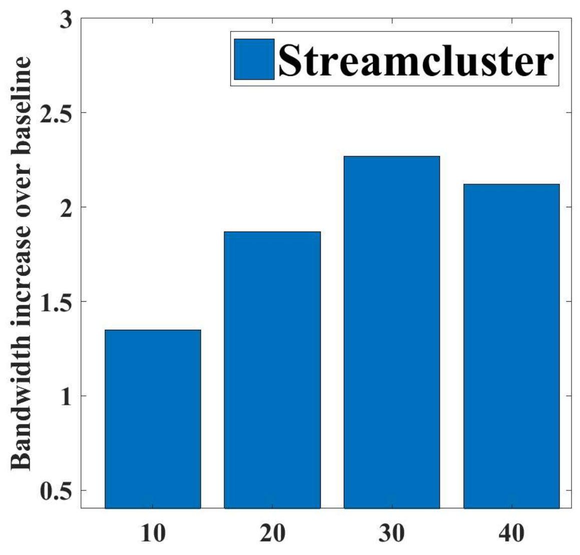


Figure 4.12: Bandwidth of MCUSDP for Stream-Cluster

entire FPGA pipeline. Such applications although do perform better after applying the taxonomy, they cannot compare to the level of performance of GPUs.

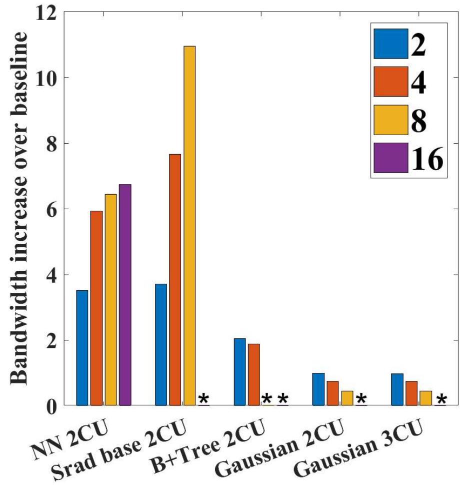


Figure 4.13: Bandwidth of MCUMDP

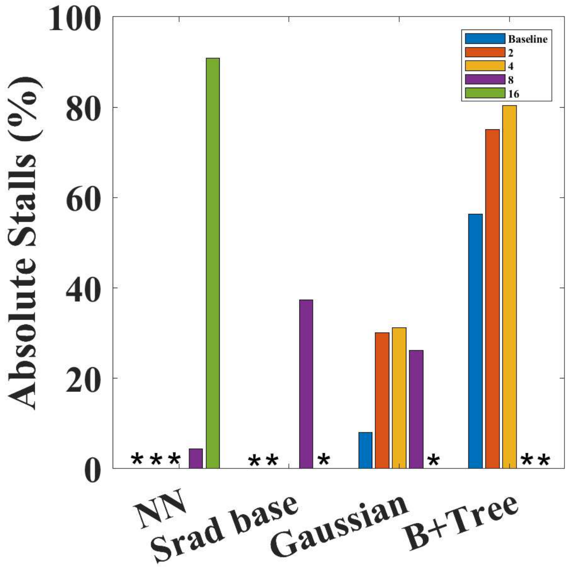


Figure 4.14: Stalls of SCUMDP

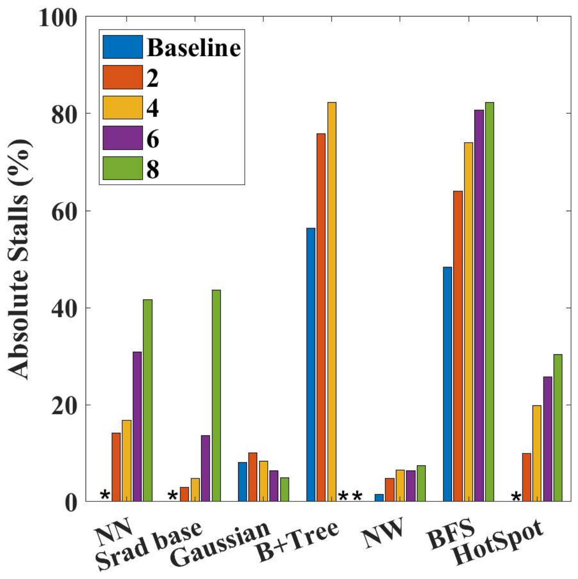


Figure 4.15: Stalls of MCUSDP

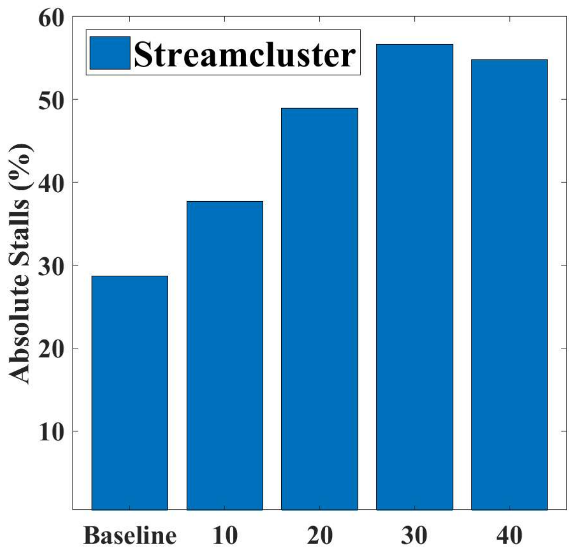


Figure 4.16: Stalls of MCUSDP for Stream-Cluster

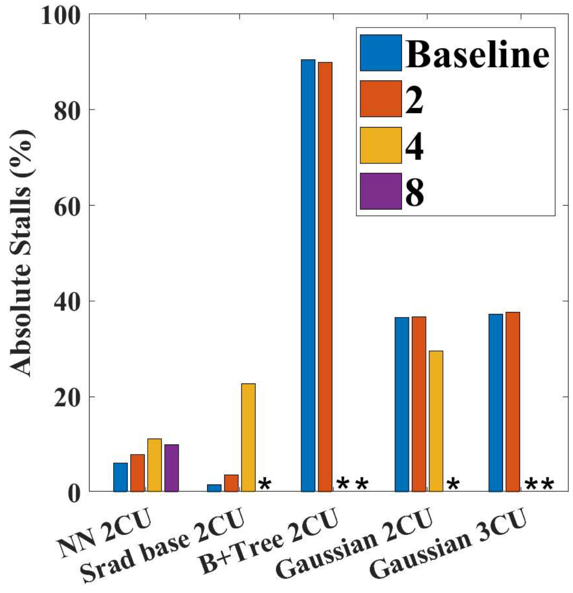


Figure 4.17: Stalls of MCUMDP

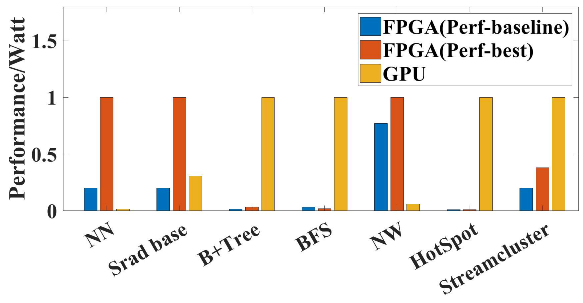


Figure 4.18: FPGA vs GPU Normalized Performance/Watts

CHAPTER 5: Sub-Kernel Temporal parallelism on FPGA Devices to Hide Memory Access Latency

5.1 SUB-KERNEL PARALLELISM

To address memory stalls in OpenCL kernels running on FPGAs, this paper suggests sub-kernel temporal parallelism to overlap memory access and thread execution at runtime.

we use the OpenCL Pipe semantic to realize sub-kernel temporal parallelism. The pipe semantic was introduced in OpenCL 2.0 and later integrated to the Altera (now Intel) OpenCL 1.0 environment. It creates an efficient data communications model (with built-in synchronization) for OpenCL kernels exchanging data in a producer-consumer fashion.

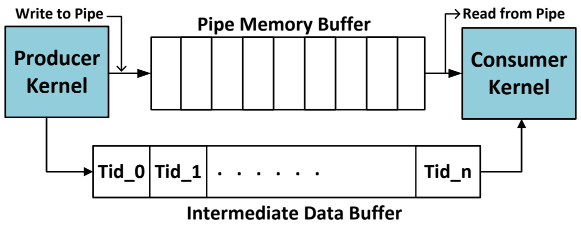


Figure 5.1: OpenCL Pipe semantic

Figure. 5.1 shows a basic pipe structure. It contains *pipe memory buffer* to store the inter-kernel communication data and an *intermediate buffer* to synchronize data communication with respect to the threads id. OpenCL-HLS often synthesis the pipe semantic in two ways: (1) *Pointer-based*, and (2) *Channel*. *Pointer-based* creates the actual Pipe construct in the global memory space (off-chip). In contrast, *Channel* creates an on-chip Pipe construct, assuming the availability of FPGA resources. *Channel* is an efficient way to keep the inter-kernel data communication on-chip, removing on-chip memory access between kernels communicating through the OpenCL pipe.

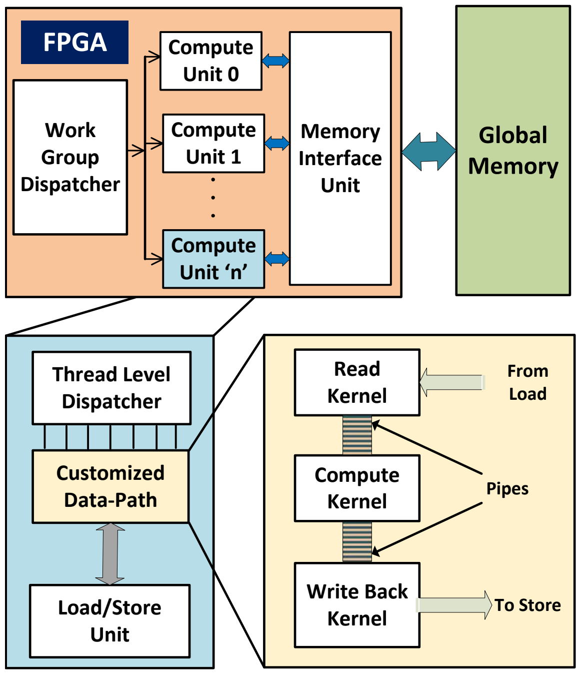


Figure 5.2: Sub-kernel parallelism

The key insight is to decouple the memory access (read and write) from the actual computation. While some of the sub-kernels are only responsible for memory accesses, others sub-kernels perform the computation. The sub-kernel temporal parallelism requires a formalized data communication and synchronization model across the sub-kernels. To maintain the current in-order thread execution model on FPGAs, we restrict the sub-kernel parallelism to producer-consumer model.

Figure. 5.2 shows a conceptual architecture model of sub-kernel parallelism. We propose to divide the kernel running per each Compute Unite (CU) to three major sub-kernels: (1) Read kernel, (2) Compute kernel, (3) Write-back kernel. The kernels execute concurrently but in an asynchronous fashion. The Read and Write back kernels are responsible for loading from and storing to the global memory while the Compute kernel only deals with computation.

For the data-communication and synchronization across the sub-kernels, we suggest using the OpenCL Pipe construct as in Figure. 5.1. To maintain the data communicating across the sub-kernels on-chip, we use the *Channel* implementation of Pipe construct. The memory accesses are issued in parallel with the computation kernel, exchanging the data through channels. As long as the channels are not empty, the sub-kernels execute concurrently across multiple threads.

The sub-kernel parallelism provides an opportunity to overlap the memory access of future threads (memory read sub-kernel) with the execution of current threads (compute sub-kernel). This offers memory prefetching at a massive scale. Sub-kernel parallelism also hides the write back latency to the memory.

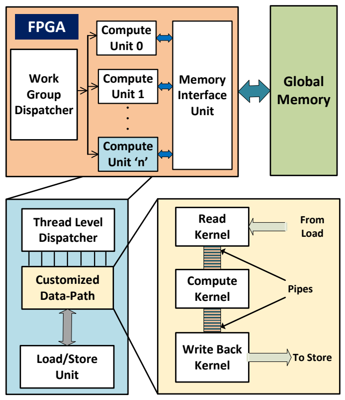


Figure 5.3: Execution pattern comparison

Figure. 5.3 provides further insight on hiding the memory latency with sub-kernel temporal parallelism. It plots an abstract execution model of one OpenCL workgroup for three scenarios: (1) baseline (generated by OpenCL-HLS), (2) baseline with local memory, and (3) sub-kernel parallelism. In the baseline model (1), all memory stalls directly exposed to execution. In the local memory model (2), the number of actual memory stalls exposed to execution path can reduce noticeably. However, the latency of memory copy operations, to copy the next workgroup data from global space (off-chip) to local space (on-chip), are directly exposed to the execution. In contrast, sub-kernel parallelism model (3) provides the opportunity to overlap memory access

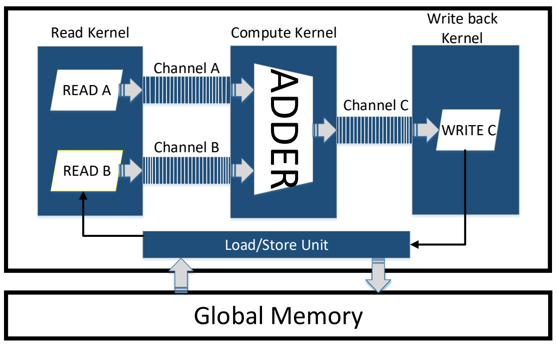


Figure 5.4: Sample kernel breakdown

latency and computation by decoupling and concurrent running of both processes. With a massive number of threads and sufficient Channel depth size, the sub-kernel parallelism can remove most of the memory stalls from execution.

5.1.1 Case Study: Vector Add

Figure. 5.4 shows the sub-kernel execution model of *Vector Add*. Read sub-kernel access the vector data from global memory and copies them into channels A and B. The compute kernel reads input channels and performs vector addition. Concurrently, it passes the results to channel C. Write back kernel reads results from channel C and write back them to global memory. All three kernels work in a parallel (producer/consumer fashion) while synchronization managed by the channels. We consider a dedicated channel per each variable. The threads in compute sub-kernel will start the execution when all input channels have the data associated with its thread ID.

Table 5.1: Variable Details

Benchmarks	Number of Threads	Prefetchable		Non-Prefetchable		Prefetchable(%)
		Size per Thread(Bytes)	Total Size(Bytes)	Size per Thread(Bytes)	Total Size(Bytes)	
B+ Tree FindK	65536	12	786432	8	524288	60
B+ Tree RangeK	65536	20	1310720	0	0	100
Gaussian	65536	12	786432	0	0	100
HotSpot	16384	12	196608	0	0	100
BFS	1048576	8	8388608	12	12582912	40
NN	42764	8	342112	0	0	100
Srad Extract	65536	4	262144	0	0	100
LUD Diagonal	4096	4	16384	0	0	100

5.2 LLVM-Based Memory Access Analysis

The promise of sub-kernel parallelism is prefetching at massive scale by overlapping the data access latency of future threads with the execution of current threads. The major restriction for sub-kernel parallelism is uncertainty about variables addresses when the address of the variables are not identified prior to the computation.

To formalize the restriction of sub-kernel parallelism, we classify the global variables, based on their memory access pattern, into two major categories: (1) prefetchable and (2) non-prefetchable. A global variable which is statically predictable is categorized as 'Prefetchable'. The examples of prefetchable variables are streaming pixels or algorithm-intrinsic variables with fixed strides such as Gaussian coefficients. Reversely, a variable is 'Non-Prefetchable' if its address is runtime dependent. Example of non-prefetchable data is the next data index in quicksort algorithm.

Consequently, in the underlying sub-kernel parallelism,. Figure. 5.5 shows the architecture realization model. Prefetchable data are accessed and written back by the read and write sub-kernels while the non-prefetchable data are directly accessed by the compute kernel. Therefore, some of the memory stalls would be still exposed to the execution path.

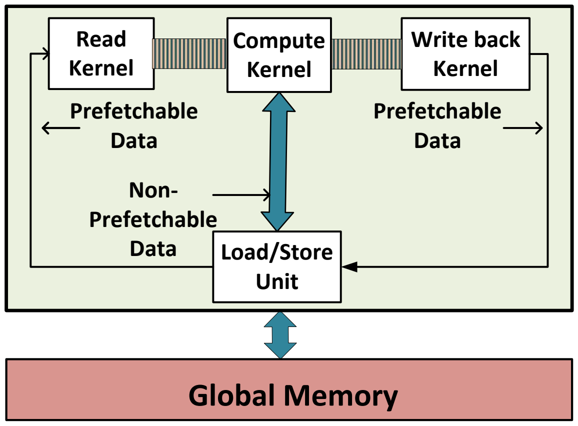


Figure 5.5: Memory prefetching with Pipes

Overall, the efficiency of sub-kernel parallelism depends on amount and ratio of prefetchable variables over non-prefetchable variables. Our experimental analysis shows that the prefetchable variables often dominate the global memory access in massively parallel applications.

5.2.1 Static Memory Analysis

This part presents our proposed static analysis approach to automatically identify the prefetchable variables for the purpose of sub-kernel parallelism.

Algorithm 1 LLVM Analyzer

```
 1: function ADDRESSANALYSIS($FILE$)
 2:     for each $memAccess \in FILE$ do
 3:         $kern \leftarrow memAccess.kernel$
 4:         $reg \leftarrow memAccess.regOperation$
 5:         $ptr \leftarrow memAccess.pointer$
 6:         $indices \leftarrow memAccess.indices$
 7:         if $ptr \in pointerList$ then
 8:             $ptrList[ptr].indices \leftarrow indices$
 9:             if $areVariable(indices)$ then
10:                 if $indices \neq ptrList[ptr].indices[0]$ then
11:                     $ptrList[ptr].pref \leftarrow$ FALSE
12:                 end if
13:             end if
14:         else
15:             $ptrList.addPointer(ptr, indexes)$
16:         end if
17:         $ptrList[ptr].pref \leftarrow$ PREFETCH($reg$)
18:     end for
19:     print to file $ptrList$
20: end function

21: function PREFETCH($destRegister$)
22:     for each $index \in destRegister$ do
23:         if $resolved(index)$ then
24:             return $ptrList[ptr].pref$ and TRUE
25:         else if $resolvable(index)$ then
26:             return $ptrList[ptr].pref$ and PREFETCH($index$)
27:         else
28:             return FALSE
29:         end if
30:     end for
31: end function
```

Abstractions introduced by high-level languages like OpenCL often hide the details of memory address calculations from the end user. To get more insight into address calculations, we move to the lower level abstraction of LLVM. LLVM is a target-independent programming abstraction that provides a view of an application at the level of micro-ops, without a limitation on the number of available registers. This allows for easy identification of memory accesses, as well as insight into how array indexes change during execution. To simplify this exploration, we developed a C++ based tool that automatically parses LLVM to identify prefetchable variables.

Algorithm 1 presents our proposed static memory access analysis. Memory address calculations are captured in LLVM with the *getelementptr* operation (line 3 to line 16). This command is composed of a destination register, the pointer being accessed, and the indexes of the pointer. We use this information to build a list of every pointer access within a kernel. Each new variable is initially viewed as prefetchable. Every time a new memory address is calculated, the $PREFETCH$ function is called (line 17).

The $PREFETCH$ function recursively traces the indexes of this access to determine if they can be predicted statically, or not. If an index cannot be resolved to a constant or OpenCL index (e.g., the thread ID), the entire memory access is marked as NON-PREFETCHABLE. Similarly, if a new access to a previously identified variable is found, we check to see if this new access could lead to an inter-thread dependence. If an inter-thread dependence is found, the variable is marked as NON-PREFETCHABLE. In both cases, the the $PREFETCH$ function returns $FALSE$ (line 28). Otherwise, if the variable index is resolved to a constant or OpenCL index, it marks as PREFETCHABLE, and the $PREFETCH$ function returns $TRUE$ (line 24). Once the tool has identified the prefetchability of every variable, it will output the results to a file, giving the user a simplified overview of memory accesses and prefetchability (line 19).

5.2.2 OpenCL-HLS Integration

We proposed a design methodology for sub-kernel parallelism as well as static LLVM-memory access analysis to guide the access/execution decoupling. At this stage, we guided the OpenCL-HLS tool to create a sub-kernel execution model by leveraging OpenCL pipe semantic. The design and automation principles introduced in this paper can be easily integrated to the commercial OpenCL-HLS tools. However, such integration requires access to the internal structure of commercial OpenCL-HLS tools.

5.3 Summary

Overall, sub-kernel temporal parallelism introduces a new level of parallelism for OpenCL applications when running on FPGAs. The reconfigurability of FPGA platforms provide the opportunity to exploit the temporal parallelism at multiple levels with respect to computation and memory access patterns. Figure. 5.6 classifies the possible levels of temporal parallelism for OpenCL abstraction on FPGA devices. Thread-level parallelism, on the right side, is constructed by OpenCL-HLS through a deeply pipelined data-path. Kernel-level parallelism, on the left side, is often created by the OpenCL programmer to efficiently construct larger applications out of multiple kernels working in a producer and consumer fashion.

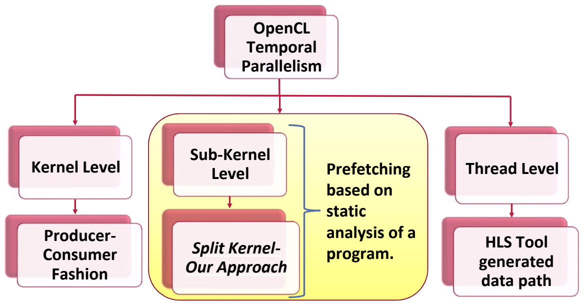

OpenCL Temporal Parallelism
Kernel Level
Producer-Consumer Fashion
Sub-Kernel Level
Split Kernel-Our Approach
Prefetching based on static analysis of a program.
Thread Level
HLS Tool generated data path

Figure 5.6: Temporal parallelism at multiple levels

The sub-kernel parallelism, in the middle, introduces a new level of parallelism for running massively parallel applications on FPGAs. In this paper, we use sub-kernel parallelism to decouple memory access from actual computation and hiding the latency of memory stalls. At this stage, we guided the OpenCL-HLS tool to create a sub-kernel execution model by leveraging OpenCL pipe semantic. However, the design and automation principles introduced in this paper can be easily integrated to the commercial OpenCL-HLS tools.

5.4 Evaluation

This section presents our experimental results for evaluating the efficiency of sub-kernel parallelism.

For the experiments, first, we leveraged our proposed LLVM analyzer tool (presented in Algorithm 1) to identify prefetchable variables per kernel. the number of prefetchable and non-prefetchable variables for each kernel. Table 5.1 lists the results of our static LLVM analysis per each OpenCL kernel. It presents the parallelism size (number of threads per each kernel), size of the prefetchable and non-prefetchable variables per thread for each kernel. It also lists the overall amount of prefetchable and non-prefetchable data size in bytes per each kernel. Overall, most of the variables are prefetchable (suitable for decoupled memory access). Only two kernels contain non-prefetchable variables: B+ Tree FindK and BFS. Please note that the table only shows the global memory access demand per each thread. OpenCL-HLS tools automatically allocate on-chip memory for local variables (private memory in OpenCL semantic).

5.4.1 Performance Evaluation

Figure. 5.7 shows the relative performance improvement of benchmarks (with sub-kernel temporal parallelism) over the baseline implementation. It also reports the relative improvement of local memory approach over the baseline implementation. Overall, sub-kernel parallelism achieves up to 2x performance improvement over baseline implementation. It also achieves the maximum performance improvement of 4.6x for *hotspot* benchmark. In contrast, local memory approach is only able to achieve 1.27x speedup over baseline implementation.

In order to provide further insight regarding the source of speedup, Figure. 5.8 and Figure. 5.9 reveal the percentage of memory stalls reduction and memory bandwidth

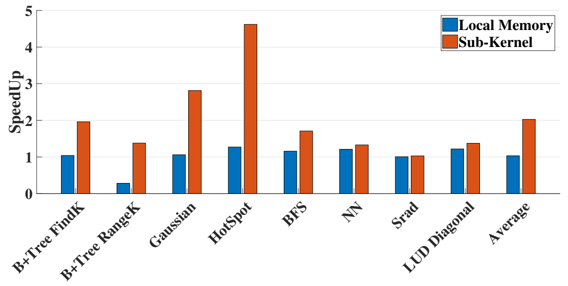


Figure 5.7: Performance improvement over the baseline

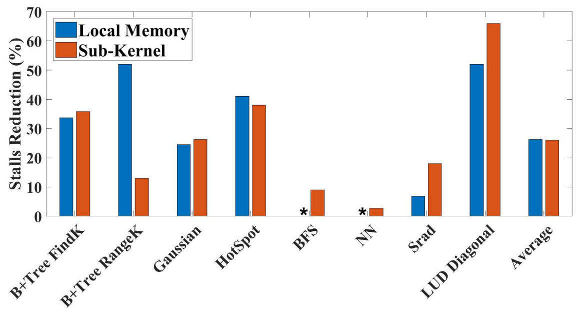


Figure 5.8: Memory stalls reduction over the baseline

improvement over baseline implementation, respectively [1]. On average, we observe 26% of stalls reduction over baseline. We also observe 2x improvement in bandwidth utilization on average. As an example, *srad* benchmark has low stalls reduction (only 18%) and small amount of prefetchable data (reflected in Table 5.1). This leads to a very low performance improvement, only 1.03x. In contrast, *hotspot* benchmark has higher stalls reduction of 38% and more prefetchable data (as reflected in Table 5.1). This gives a higher performance improvement of 4.6x times over baseline. Two other benchmarks namely *B+ Tree Find K* and *BFS* do not achieve comparable speedup since they have more number of non-prefetchable variables which affects their percentage of stall reduction.

Please note that, the local memory approach is also able to achieve the comparable reduction in the memory stalls (26% over baseline). However, due to the fact that the memory copies, from global to local space, are directly exposed to execution path, it achieves very limited performance improvement.

[1] In Figure. 5.8, no improvement in memory stalls are shown by asterisk

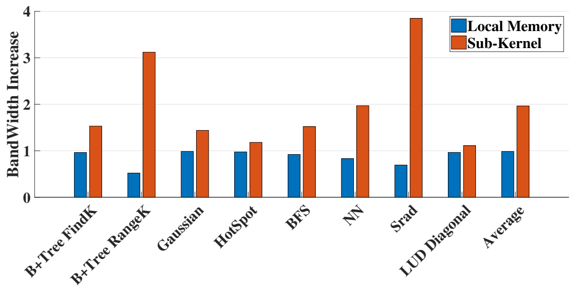


Figure 5.9: Memory bandwidth improvement over the baseline

5.4.2 Resource Overhead

This section briefly presents the effect of our proposed sub-kernel parallelism on power, energy and resources utilization. Figure. 5.10 shows the percentage of resource overhead over baseline [2]. The resource overhead is mainly introduced due to additional register blocks, memory block and combinational logic which are required for constructing the pipe (channel) semantic. On average we observe 3% increase in register blocks, 4% increase in memory blocks and 3% increase in combinatorial logic.

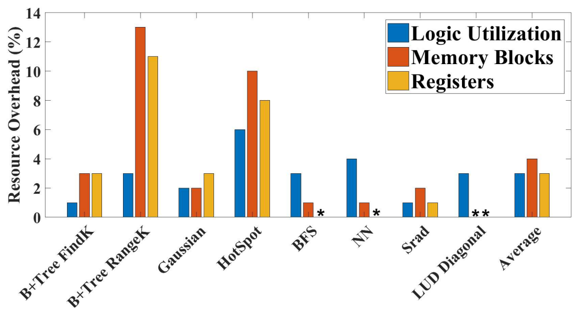


Figure 5.10: Resource utilization overhead over the baseline

5.4.3 Power Overhead and Energy Saving

This section presents power overhead caused by increase in FPGAs resource utilization. To calculate the power overhead, we used the Stratix V PowerPlay Early Power Estimator tool. Total thermal power is calculated as a sum of the Static Power(P_{STATIC}), Dynamic Power($P_{DYNAMIC}$) and I/O Power[61]. Static power is the leakage power dissipated from the chip independent of user clocks. The I/O power

[2]In Figure. 5.10, zero resource overhead are shown by asterisk

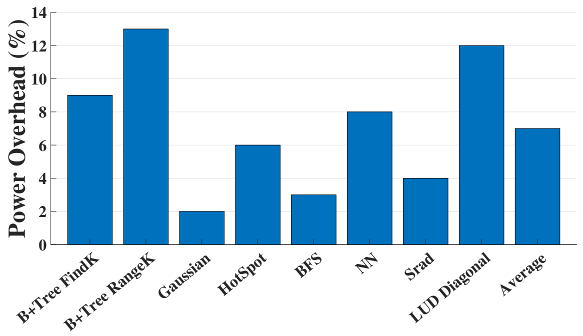


Figure 5.11: Power overhead over the baseline

is the DC bias transceiver DC bias power. The Dynamic power is calculated from internal nodes changing logic levels internal to the device in the form of equivalent lumped capacitance's, it is given as below:-

$$\begin{aligned} DynamicPower = V_{\mathrm{CCINT}} \times \sum I_{\mathrm{CCINT}}(&LE/ALM, RAM, \\ &DSP, PLL, Clocks, HSDI, Routing) \end{aligned} \tag{5.1}$$

Here dynamic power dissipated is measured across each of Adaptive Logic Modules(ALMs), RAM Blocks(RAM), DSP blocks(DSP), Phase Lock loops(PLLs), Clock, High Speed Differential I/Os (HSIO) and associated routing modules.

Figure. 5.11 and Figure. 5.12 presents the relative power overhead and energy saving over the baseline implementation. On average, we observe 7% increase in power consumption over baseline implementation. In contrast, we observe overall 40% energy saving compared to the baseline implementation. The energy saving stems from significant speedup and reduction in overall execution time. The maximum power overhead of 13% is obtained for *b+tree rangek* benchmark (due to its relatively noticeable resource overhead, presented in Figure. 5.10). This results in only 19% of energy saving for *b+tree rangek* benchmark. On the other hand, we observe only 2% power overhead for *gaussian* benchmark (due to its low resource overhead). The results also show 65% energy saving for *gaussian*.

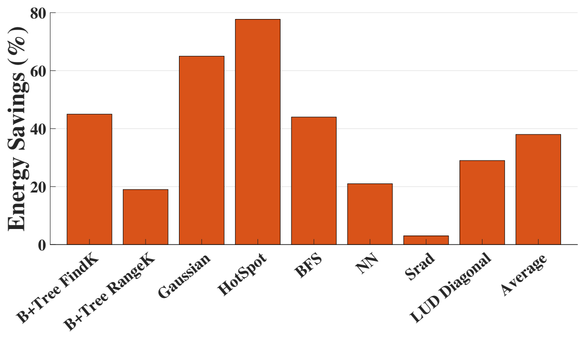


Figure 5.12: Energy saving over the baseline

5.4.4 Performance per Watt

In this section, we present Performance per watt as an evaluation metric to compare the combined performance power efficiency of our proposed approach. To calculate Performance per watt, we need to capture actual computation demand (arithmetic operations) per each kernel. To derive computation demand, we profiled the kernels, using the PIN PYtracer tool[63], and extract the total number of instructions, memory and branch accesses per kernel. We then estimated the total computation demand per kernel based on Equation (5.2).

$$\begin{split} \#ComputeOps = \#TotalInstructions - \\ (\#MemoryInstructions + \#BranchInstructions) \end{split} \tag{5.2}$$

$$Perf/Watt = \frac{\#ComputeInstructions/sec}{Powerconsumption(Watts)} \tag{5.3}$$

Equation (5.3) shows the Performance/Watt per OpenCL kernel. The Figure. 5.13 shows a comparison between Baseline FPGA performance vs Local Memory version vs Sub-kernel version. We normalize the values obtained with respect to our proposed sub-kernel parallelism approach. We clearly see our approach beating baseline and local memory versions for all kernels, except *LUD*. The highest performance/Watt was achieved for *Hotspot* (with 5x improvement over the baseline) and *B+Tree Range K* (with more than 20x improvement over the baseline). The improvement stems from combined effect of large size of prefetchable variables(Table 5.1), significant reduction in memory stalls(Figure. 5.8) and considerable improvement in performance (Figure. 5.7), against marginal power overhead (Figure. 5.11) due to increase in resources (Figure. 5.10) for implementing the pipe.

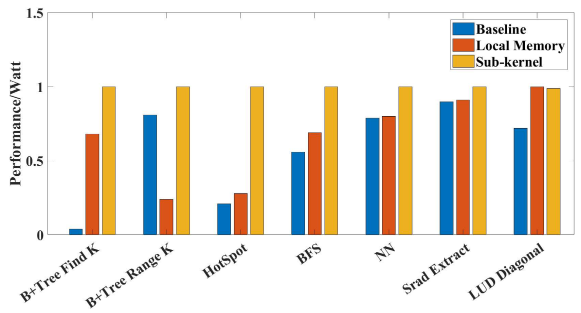


Figure 5.13: Normalized Performance/Watts

5.4.5 Detailed Evaluation

To provide more insights on our proposed approach, this section briefly studies the impact of the Pipe size (channel depth) on the sub-kernel temporal parallelism. As an example, we choose *b+tree rangek* for the exploration. We allocate separate channels for each global variable that is statically prefetchable. The private variables that are internal to each kernel are part of the data-path. These variables are not exposed to the memory fetch overhead. For each of the global variables we increase

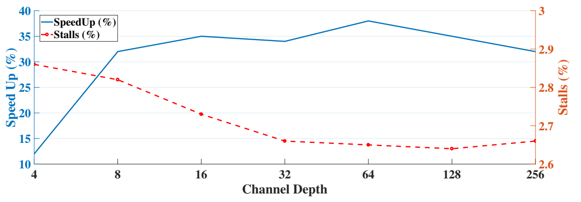


Figure 5.14: *B+ Tree RangeK* performance improvement and memory stalls over increasing channel depth

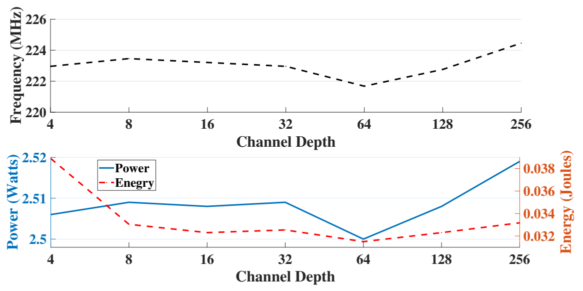


Figure 5.15: *B+ Tree RangeK* power, energy and clock frequency over increasing channel depth

the pipe depths from *4* to *256* thereby increasing the channel buffer size per variable. Figure. 5.14 shows the percentage of performance improvement (compared to the baseline implementation) as well as the percentage of overall memory stalls over an exponentially increasing channel depth. We observe a gradual increase in performance improvement (with peak improvement at the channel depth of 64). The channel depth of 64 also results in minimum memory stalls (2.65% of total memory stalls). We observe a reduction in performance improvement after the channel depth of 64 which is due to the propagation latency of the larger channel depth.

Figure. 5.15 plots the absolute power and energy consumption over varying channel depth for *B+ Tree RangeK*. We almost observe the similar pattern of performance improvement and memory stalls. Interestingly, the channel depth of 64 results the lowest frequency (221MHz) lower power consumption (2.5Watts) and energy consumption (0.0315 Joules).

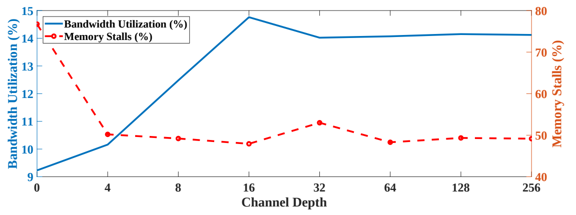


Figure 5.16: *Hotspot* bandwidth utilization and memory stalls over increasing channel depth

As additional examples, Figure. 5.16 and Figure. 5.17 present the impact on mem-

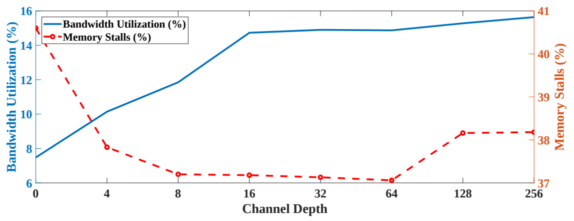


Figure 5.17: *BFS* bandwidth utilization and memory stalls over increasing channel depth

ory bandwidth utilization and memory stalls over increasing depth of pipe for *Hotspot* and *BFS* kernels. The pipe depth zero reflects the baseline implementation (no sub-kernel parallelism). We observe a significant reduction in memory stalls and increase in memory bandwidth for both applications. In both kernels, we observe a huge reduction in memory stalls even with very small pipe depth (the depth of 4). For *Hotspot*, the maximum benefits appear in the pipe size of 16. For *BFS*, the maximum benefits appear in the pipe size of 64. Overall, the results show a variation for optimum channel depth size across the OpenCL kernels. This brings an interesting research question about optimum channel depth for our sub-kernel temporal parallelism.

CHAPTER 6: Open Source OpenCL-HLS Tool

Primary focus of this work is to propose a novel OpenCL-HLS optimization method to remove few of bottlenecks observed at comercial OpenCL-HLS compiler tool such as

- Expand Sub-Kernel parallelism to multiple compute units
- Integrating the results of analysis(Taxonomy) to the OpenCL-HLS

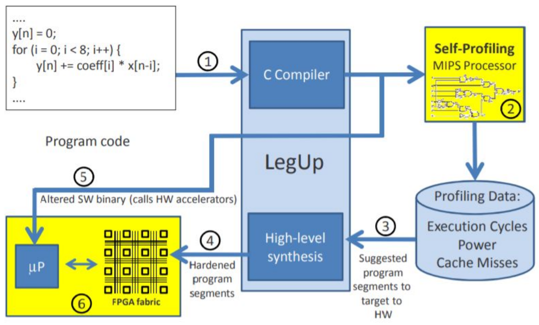
In this project we combine the collected insights of previous two projects to develop novel integrated architecture and EDA tools.

- Create open source OpenCL-HLS tool
- Create run-time OpenCL thread scheduling at FPGAs similar to GPUs

Previously, there has been a lot of research going on in the feild of compiler development for GPUs or for CPUs[65, 66, 67]. We make use of few of the existing open source tool chains and merge them to produce the OpenCL-HLS tool chain. We explore LegUp and Pocl tool chains to obtain OpenCL-HLS compiler.

6.1 LegUP

LegUp is an open source high-level synthesis tool being developed at the University of Toronto. The LegUp framework allows researchers to improve C to Verilog synthesis without building an infrastructure from scratch. Our long-term vision is to make FPGA programming easier for software developers. LegUp accepts a standard C program as input and automatically compiles the program to a hybrid architecture



```
....
y[n] = 0;
for (i = 0; i < 8; i++) {
      y[n] += coeff[i] * x[n-i];
}
....
```

Figure 6.1: Design Flow with LegUP

containing an FPGA-based MIPS soft processor and custom hardware accelerators that communicate through a standard bus interface.

The LegUp design flow comprises first compiling and running a program on a standard processor, profiling its execution, selecting program segments to target to hardware, and then re-compiling the program to a hybrid hardware/software system. Figure. 6.1 illustrates the detailed flow. Referring to the labels in the figure, at step 1, a C compiler compiles a program to a binary executable. At step 2, the executable runs on an FPGA-based MIPS processor. At step 3 LegUp is invoked to compile these segments to synthesizeable Verilog RTL. LegUp's hardware synthesis and software compilation are part of the same compiler framework. Presently, LegUp HLS operates at the function level: entire functions are synthesized to hardware from the C source. The RTL produced by LegUp is synthesized to an FPGA implementation using standard commercial tools at step 4. In step 6, the C source is modified such that the functions implemented as hardware accelerators are replaced by wrapper functions that call the accelerators (instead of doing computations in software). This new modified source is compiled to a MIPS binary executable. Finally, in step 6 the hybrid processor/accelerator system executes on the FPGA.

6.2 Portable OpenCL(PoCL)

PoCL is a portable open source synthesis tool developed by MIT. PoCL is implementation of the OpenCL standard (1.2 with some 2.0 features supported). In addition to producing an easily portable open-source OpenCL implementation, another major goal of this tool is improving performance portability of OpenCL programs with the kernel compiler and the task runtime, reducing the need for target-dependent manual optimizations.

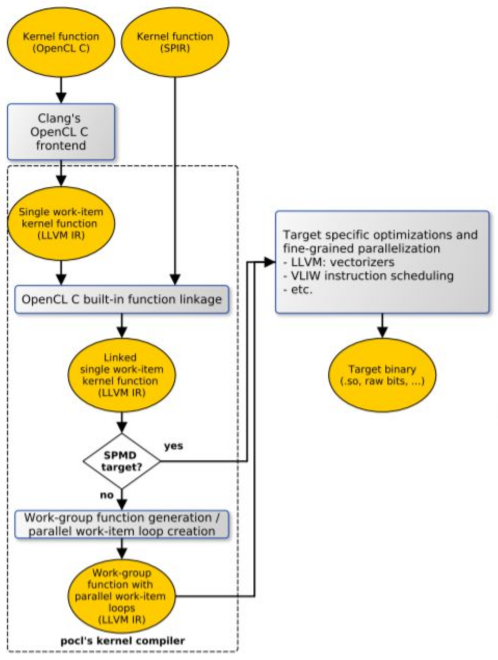

Kernel function (OpenCL C)
Kernel function (SPIR)
Clang's OpenCL C frontend
Single work-item kernel function (LLVM IR)
OpenCL C built-in function linkage
Linked single work-item kernel function (LLVM IR)
SPMD target?
yes
no
Work-group function generation / parallel work-item loop creation
Work-group function with parallel work-item loops (LLVM IR)
pocl's kernel compiler
Target specific optimizations and fine-grained parallelization
- LLVM: vectorizers
- VLIW instruction scheduling
- etc.
Target binary (.so, raw bits, ...)

Figure 6.2: Design Flow with PoCL

pocl shown in Figure. 6.2 uses Clang as an OpenCL C frontend and LLVM for kernel compiler implementation, and as a portability layer. Thus, if your desired target has an LLVM backend, it should be able to get OpenCL support easily by using pocl.

Figure. 6.2 is illustration of pocl's kernel compilation chain. The source code of the kernel is read by Clang which produces an LLVM IR for the single work-item kernel description. Alternatively, a pre-built SPIR bitcode binary can be used as an input.

The OpenCL C built-in functions are linked at the LLVM IR level to the kernel after which the optional work-group function generation is done. In case the target can execute the SPMD single work-item kernel description directly for all work-items in the work-group (as is the case with most GPUs), or the local size is one, this step is skipped. The work-group function generation is the last responsibility of the pocl's kernel compiler; it helps the later target-specific passes (such as vectorization) by creating parallel work-item loops which are annotated using LLVM metadata.

pocl currently has backends supporting many CPUs, ASIPs (TCE/TTA), NVIDIA GPUs (via CUDA), HSA-supported GPUs and multiple private off-tree targets.

In addition to providing an open source implementation of OpenCL for various platforms, an additional purpose of the project is to serve as a research platform for issues in parallel programming of heterogeneous platforms.

6.3 Integration of LegUP and PoCL

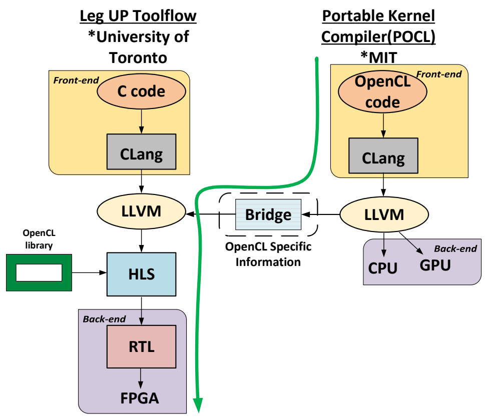


Figure 6.3: Open Source OpenCL-HLS flow

This section gives brief description of how the open source OpenCL-HLS is integrated as show Figure. 6.3 keeping PoCL and LegUP as base synthesis tools. We make use of PoCL for OpenCL kernel to produce LLVM generated code. The LLVM code will also have meta data related to OpenCL constructs such as Barrier, local id, global id and more. Then in second part we integrate LegUP to use the populated LLVM from PoCL and generate the RTL which can be used to execute on FPGAs to analyze the metrics such as performance improvements, Stalls, Bandwidth, Power/watts, Energy consumption, resource utilization.

6.4 Evaluation

This section gives the explanation for results of Open source OpenCL-HLS tool. This work is still in progress.

CHAPTER 7: CONCLUSIONS and FUTURE WORK

7.1 Conclusions

This research explores and studies various programming decisions that can improve the thread-level utilization, performance and occupancy on FPGAs, as well as decisions that result in a more efficient data-path. The focus of our study is to identify the granularity of OpenCL threads that can exploit spatial parallelism on top of temporal parallelism on FPGAs. The work primarily focuses on proposing a new taxonomy based on the correlation between OpenCL parallelism abstraction and execution semantic of FPGAs. The aim is to provide an early insight to formalize OpenCL written codes on FPGAs in order to enhance their efficiency guiding both OpenCL programmers and OpenCL synthesis tools.

Several challenges were however observed in the process of carrying out various experiments on our FPGA board. One key issue faced is hardware limitation due to unavailability of resources when increasing the number of compute units and data-paths for our taxonomy categories. Another key limitation occurs during data-path replication in SCUMDP and MCUMDP where applications that have complex kernels containing conditional branches cannot be vectorizable. This is a limitation of the HLS tool.

Overall, taxonomy results on eight applications of rodinia benchmark indicate that FPGA-aware OpenCL written codes can achieve up to 6.7X maximum throughput and perform consistently for various classifications. Furthermore, OpenCL source-code decisions that can exploit spatial and temporal parallelism will be able to hide the memory access latency and thus result in a higher speedup.

The proposed sub-kernel temporal parallelism to hide the memory latency of massively parallel applications running on FPGA devices provides the opportunity to prefetch the data of future threads concurrent with the execution of current threads by creating a concurrency execution model between the computation and memory access among the OpenCL threads. This proposed approach uses OpenCL Pipe semantic to realize sub-kernel parallelism, and LLVM static analysis to identify the statically prefetchable data. Our experimental results over eight Rodinia kernels running on Intel Stratix-V FPGA demonstrates 2x speedup with 40% energy reduction and minimum resource utilization overhead compared to baseline implementation.

7.2 Future Work

As a future work we will be working on the development of open source Open-HLS compiler development. This work is the integration of different already existing tool-chains such as LegUP and PoCL. In this work we mainly focus on the OpenCL attributes which need to handled in the Backend. The front-end of the Integrator will be the libclc or PoCL which converts OpenCL kernel into a Low Level intermediate representation LLVM. which has been already implemented. Next step will be to develop a back-end tool to convert LLVM to RTL design for FPGA devices. For which we will be using LegUP as the base which supports c to RTL and give support for OpenCL attributes. The whole motivation behind the development of this integrator is to help in simulation of FPGA devices and understand new architecture exploration.

REFERENCES

[1] "Altera sdk for opencl." `http://www.altera.com/literature/lit-opencl-sdk.jsp`, 2015.

[2] "Xilinx opencl." `http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`, 2015.

[3] "Fpga developer ami." `https://aws.amazon.com/marketplace/pp/B06VVYBLZZ`, 2015.

[4] "Project brainwave for real-time ai," 2017.

[5] J. Müller, J. Müller, and R. Tetzlaff, "A new high-speed real-time video processing platform," in *2014 14th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA)*, 2014.

[6] T. Kryjak, M. Komorkiewicz, and M. Gorgon, "Real-time hardware–software embedded vision system for its smart camera implemented in zynq soc," *Journal of Real-Time Image Processing*, 2016.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, 2009.

[8] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, pp. 948–960, Sept. 1972.

[9] D. Chen and D. P. Singh, "Fractal video compression in opencl: An evaluation of cpus, gpus, and fpgas as acceleration platforms," in *18th Asia and South Pacific Design Automation Conference*, 2013.

[10] J. Andrade, G. Falcão, V. Silva, and K. Kasai, "Flexible non-binary ldpc decoding on fpgas," in *IEEE International Conf. on Acoustics, Speech, and Signal Processing - ICASSP*, vol. 1, pp. 1–5, 2014.

[11] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl," in *International Conference on Field-Programmable Technology (FPT)*, 2014.

[12] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl(tm) deep learning accelerator on arria 10," *CoRR*, 2017.

[13] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson,

S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, “A reconfigurable fabric for accelerating large-scale datacenter services,” *Commun. ACM*, 2016.

[14] Z. Wang, B. He, W. Zhang, and S. Jiang, “A performance analysis framework for optimizing opencl applications on fpgas,” in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016.

[15] K. O. W. Group *et al.*, “The opencl specification,” *Version*, vol. 1, no. 29, p. 8, 2008.

[16] C. Grozea, Z. Bankovic, and P. Laskov, “Fpga vs. multi-core cpus vs. gpus: hands-on experience with a sorting application,” in *Facing the multicore-challenge*, pp. 105–117, Springer, 2010.

[17] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, “Accelerating computer vision algorithms using opencl framework on the mobile gpu-a case study,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2629–2633, IEEE, 2013.

[18] T. Shimobaba, T. Ito, N. Masuda, Y. Ichihashi, and N. Takada, “Fast calculation of computer-generated-hologram on amd hd5000 series gpu and opencl,” *Optics Express*, vol. 18, no. 10, pp. 9955–9960, 2010.

[19] P. Mistry, Y. Ukidave, D. Schaa, and D. Kaeli, “Valar: a benchmark suite to study the dynamic behavior of heterogeneous systems,” in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pp. 54–65, ACM, 2013.

[20] D. B. Skillicorn, “A taxonomy for computer architectures,” *Computer*, Nov. 1988.

[21] C. D. C. Ralph Duncan, “A survev of parallel computer architectures,” *IEEE Computer Society*, Feb. 1990.

[22] P. O. Jäskeläinen, S. Carlos, P. Huerta, and J. H. Takala, “Opencl-based design methodology for application-specific processors,” in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pp. 223–230, IEEE, 2010.

[23] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, “Synthesis of platform architectures from opencl programs,” in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 186–193, IEEE, 2011.

[24] “Intel sdk for opencl applications,” 2017.

[25] J. He, M. Lu, and B. He, “Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture,” *ArXiv e-prints*, July 2013.

[26] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *PVLDB*, vol. 8, no. 4, pp. 329–340, 2014.

[27] J. E. Smith, "Decoupled access/execute computer architectures," in *25 Years ISCA: Retrospectives and Reprints*, pp. 231–238, ACM, 1998.

[28] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 46:1–46:12, IEEE Press, 2016.

[29] S. Cheng and J. Wawrzynek, "Architectural synthesis of computational pipelines with decoupled memory access," in *FPT*, pp. 83–90, IEEE, 2014.

[30] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *23rd IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2012, Delft, The Netherlands, July 9-11, 2012*, 2012.

[31] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," 2000.

[32] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, 2015.

[33] B. Panda and S. Balachandran, "Hardware prefetchers for emerging parallel applications," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, 2012.

[34] T. Kim, D. Zhao, and A. V. Veidenbaum, "Multiple stream tracker: A new hardware stride prefetcher," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, 2014.

[35] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, 2004.

[36] D. Bernstein, D. Cohen, and A. Freund, "Compiler techniques for data prefetching on the powerpc," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '95, 1995.

[37] G. Marin, C. McCurdy, and J. S. Vetter, "Diagnosis and optimization of application prefetching performance," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, 2013.

[38] A. C. Klaiber and H. M. Levy, "An architecture for software-controlled data prefetching," *SIGARCH Comput. Archit. News*, 1991.

[39] D. F. Zucker, R. B. Lee, and M. J. Flynn, "Hardware and software cache prefetching techniques for mpeg benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, 2000.

[40] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, 1990.

[41] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[42] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016.

[43] O. Segal, N. Nasiri, M. Margala, and W. Vanderbauwhede, "High level programming of fpgas for HPC and data centric applications," in *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, 2014.

[44] S.-D. W. Tahsin Turker Mutlugun, "OpenCL computing on FPGA using multiported," 2015.

[45] D. Weller, F. Oboril, D. Lukarski, J. Becker, and M. B. Tahoori, "Energy efficient scientific computing on fpgas using opencl," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017.

[46] Y. Ukidave, C. Kalra, D. R. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*, 2014.

[47] A. Momeni, H. Tabkhi, Y. Ukidave, G. Schirner, and D. R. Kaeli, "Exploring the efficiency of the opencl pipe semantic on an FPGA," *SIGARCH Computer Architecture News*, 2015.

[48] J. Zhang and J. Li, "Improving the performance of opencl-based FPGA accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*, 2017.

[49] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, 2017.

[50] M. Z. Hasan and S. G. Sotirios, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems*, 2009.

[51] E. Nurvitadhi, J. C. Hoe, S.-L. L. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference*, DAC '10, ACM, 2010.

[52] M. Tan, B. Liu, S. Dai, and Z. Zhang, "Multithreaded pipeline synthesis for data-parallel kernels," in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, 2014.

[53] R. J. Halstead, J. Villarreal, and W. Najjar, "Exploring irregular memory accesses on fpgas," in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '11, ACM, 2011.

[54] R. J. Halstead and W. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, 2013.

[55] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, 2015.

[56] K. Turkington, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Outer loop pipelining for application specific datapaths in fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2008.

[57] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao, "Single-dimension software pipelining for multi-dimensional loops," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[58] M. Z. Hasan and S. G. Ziavras, "Customized kernel execution on reconfigurable hardware for embedded applications," *Microprocessors and Microsystems - Embedded Hardware Design*, 2009.

[59] E. Nurvitadhi, J. C. Hoe, S. Lu, and T. Kam, "Automatic multithreaded pipeline synthesis from transactional datapath specifications," in *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, 2010.

[60] R. J. Halstead and W. A. Najjar, "Compiled multithreaded data paths on fpgas for dynamic workloads," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2013, Montreal, QC, Canada, September 29 - October 4, 2013*, 2013.

[61] "Intel powerplay early power estimator user guide," 2017.

[62] "Rodinia:accelerating compute-intensive applications with accelerators." `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators`.

[63] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, 2004.

[64] "Amd. codexl 3.1 edition," 2017.

[65] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.

[66] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," 2013.

[67] J. H. Song Ruiling, "Beignet: Intel opensource project," 2017.

VITA

SUHAS ASHOK SHIDDIBHAVI

EDUCATION:

Master of Science, Computer Engineering at University of North Carolina at Charlotte, August 2016 - Present. Thesis Title: "FPGAs for Massively Parallel Applications."

Bachelor of Engineering, Electronics & Communication Engineering at Visvesvaraya Technological University (VTU), India, June 2010 - May 2014.

ACADEMIC EMPLOYMENT:

Graduate Teaching Assistant, Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, May 2017 - August 2017, Responsibilities include: assisting professors with the preparation and presentation of undergraduate courses, grading, and tutoring.

Graduate Research Assistant to Dr Hamed Tabkhi, Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, January 2017 - present.

PUBLICATIONS and ACHIVEMENTS:

Submitted a System on Chip Conference paper 2018 on "**Taxonomy of Spatial parallelism on FPGAs for Massively Parallel Applications.**"

Submitted a International Conference on Computer-Aided Design paper 2018 on **"LLVM-Based OpenCL Sub-Kernel Parallelism for Automated Decoupled Memory Access on FPGAs."**