Impact Of Thread Scheduling On Modern Gpus

Orevaoghene Addoh
University of Mississippi

IMPACT OF THREAD SCHEDULING ON MODERN GPUs

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Orevaoghene Addoh

May 2014

ABSTRACT

The Graphics Processing Unit (GPU) has become a more important component in high-performance computing systems as it accelerates data and compute intensive applications significantly with less cost and power. The GPU achieves high performance by executing massive number of threads in parallel in a SPMD (Single Program Multiple Data) fashion. Threads are grouped into workgroups by programmer and workgroups are then assigned to each compute core on the GPU by hardware. Once assigned, a workgroup is further subgrouped into wavefronts of the fixed number of threads by hardware when executed in a SIMD (Single Instruction Multiple Data) fashion.

In this thesis, we investigate the impact of thread (at workgroup and wavefront level) scheduling on overall hardware utilization and performance. We implement four different thread schedulers: Two-level wavefront scheduler, Lookahead wavefront scheduler and Two-level + Lookahead wavefront scheduler, and Block workgroup scheduler. We implement and test these schedulers on a cycle accurate detailed architectural simulator called *Multi2Sim* targeting AMD's latest Graphics Core Next (GCN) architecture.

Our extensive evaluation and analysis show that using some of these alternate mechanisms, cache hit rate is improved by an average of 30% compared to the baseline round-robin scheduler, thus drastically reducing the number of stalls caused by long latency memory operations. We also observe that some of these schedulers improve overall performance by an average of 17% compared to the baseline.

DEDICATION

I dedicate this thesis to my loving parents. To my father, who serves as a contant source of inspiration and to my mother, whose kindness and grace never ceases to amaze me.

ACKNOWLEDGEMENTS

Foremost, I would like to express my thanks and sincere appreciation to my advisor Dr. Byunghyun Jang for the unwavering support and encouragement during my Masters study. His patience, availability and immense knowledge of computer architecture and GPU computing helped me at all points of my graduate study and research.

I would also like to thank the rest of my thesis committee, Dr. Philip Rhodes and Dr. Jianxia Xue, whom I've also been blessed with the opportunity to learn and get advice from. Their immense knowledge of cloud computing, parallel computing, graphics, among other topics have helped me grow my toolset as a young scientist.

I thank my department chair, Dr. Conrad Cunningham and the rest of our amazing faulty and staff. Thank you for your support and the wonderful job at the department.

To my fellow labmates at the Heterogeneous Systems Research Lab: Kyoshin Choo, David Troendle, Mainul Hassan, Stephen Adams, William Panlener, Mengshen Zhao, Tuan Ta, Sampath Gowrishetty, Michael Ginn and Elijah Allen, thank you for all your assistance with my research and stimulating discussions during our meetings.

Finally, to my wonderful parents and siblings, thank you for all your advice, sacrifice, encouragement and support thoughout my life. I couldn't ask for a better family.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 GPU as a general-purpose co-processor

The performance improvement of traditional CPUs over generations is slowly declining as microprocessor manufacturers are faced with obstacles such as the frequency and power walls. This implies that multi-core CPU is limited by the amount of task-level parallelism present in applications. Recently the Graphics Processing Unit (GPU) is emerging as an excellent general-purpose co-processor for data-intensive tasks. This parallel machine was originally designed to perform graphics rendering and had fixed function hardware. However, recent advances has been made in hardware and software of GPU to perform non-graphics computation - leading to a new computing paradigm called General Purpose Computation on Graphics Processing Units (GPGPU).

GPGPU has been very successful and many applications in various fields have been successfully accelerated. However, it is still far from mature and computer architects are constantly finding new ways to improve it in order to achieve its peak potential. A particularly challenging problem in GPGPU is under-utilization of compute cores in current GPU architectures. This underutilization can be caused by different factors. Two major factors are in memory subsystem and thread scheduling. Memory subsystem performance is affected by various reasons which make it hard for the GPU to hide long latency memory operations due to cache performance. Cache performance is affected by various reasons but a major factor in cache performance is the locality present in applications. The more the spatial and temporal locality present in application, the better the cache performs and the number of long latency operations to main memory can be reduced.

In a single threaded application running on a single core processor, the locality present in the cache can be preserved because no other threads compete for resources. However, for architectures like the GPU where thousands of threads run simultaneously, the contention for cache resources easily increases and as a result the locality[1] present in the application can be destroyed leading to inefficient cache performance [29] [32].

This destruction of locality by many threads must be mitigated in order to achieve better cache performance and increased utilization of the GPUs execution units. One way to do this is by improving scheduling mechanism of the units of work (thread) to the GPU cores. A better scheduler would assign threads to the GPU cores in such a way that as much locality as possible is preserved throughout program execution.

Apart from helping to preserve the locality in the cache, the scheduler algorithm can also improve efficiency by increasing the achieved *instruction scheduling width* per cycle. If we increase the rate of thread assignment to the execution units, we in turn increase their utilization.

1.2 Objectives of Thesis

GPUs use the abundance of parallelism found in data-parallel applications to tolerate memory access latency by interleaving the execution of wavefronts[2]. These wavefronts may be

[1]Phenomenon describing the same value or related storage locations being frequently accessed.

[2]A wavefront consists of exactly 64 threads. A workgroup is also a group of threads, but its size is configurable by the user up to 256 threads.

from the same workgroup or from different workgroups running on the same core [6]. While this approach works, its effectiveness is dependent on what scheduling algorithm is used to assign the workgroups and wavefronts to the cores. The default scheduler used in today's GPUs is a round-robin scheduler [10][15]. All active wavefronts in the core proceed at roughly the same speed using this algorithm. Given the abundance of threads in the core, the GPU hides long latency operations by scheduling more threads when there is a stall. However, since all the warps proceed at roughly the same speed, they typically all reach these long latency operations at roughly the same time causing the core to stall and valuable execution cycles to be wasted - leading to underutilization of resources.

The main objective of this research is to investigate and demonstrate the impact these scheduling decisions have on the GPU performance. To this end we implement alternative thread scheduling algorithms and compare their performance to that of the default round-robin scheduler used in production GPUs. Some of the contributions in our work are:

- Schedulers: We design a novel *Lookahead scheduler* that aims to performance of our target GPU architecture when it is unsaturated. We also, investigate the performance of schedulers proposed in related research on our target architecture.

- Micro-architecture: For our work, we use a micro-architecture based off AMD's Southern Islands micro-architecture. This architecture features Branch and Scalar functional units as well as an execution pipeline not present in architectures used in similar studies. These present new challenges to the algorithms proposed in previous work. To our knowledge, this is the first time this kind of work has been done using Southern Islands.

- Architectural simulator: We use a detailed cycle-accurate architectural simulator called Multi2Sim [33]. This simulator fully models the architectural pipeline in detail and to our knowledge no similar studies have been done using it.

- GPU workload: Previous research we could find showed the impact of scheduling decisions in cases where the GPU cores were saturated with a massive number of threads. While we do that, we also show the impact of these decisions in cases where the GPUs cores are unsaturated. We consider this case in order to provide another performance comparison point between the various schedulers.

To demonstrate the performance impact of thread scheduling on modern GPU performance, we followed a step-by-step, modular approach to our study. First, we characterized benchmarks to understand the behavior of the default round-robin scheduler. This characterization provided insight that inspired the design and implementation of a *lookahead scheduler*. This scheduler looks at the next fetch buffers (section 2.3.3) to find a valid instructions to schedule if none is found in the fetch buffer chosen in the current clock cycle. We then implemented a *two-level wavefront scheduler* proposed in related research. To increase the locality preserved in the cache, we implemented a block workgroup scheduler. This scheduler assigned assigns a block of sequential workgroups to each core. Different combinations of these schedulers were also studied.

To understand the impact of our proposals, we profiled several benchmarks with a widely varying instruction mix characteristics. The results were then analyzed compared to that of the baseline scheduler. Chapter 3 discusses the different schedulers and in Chapter 4, we discuss the obtained results.

1.3 Related work

Several researchers have identified the aforementioned problems with GPU core underutilization and thread scheduling inefficiency. As a result, there has been efforts to find ways to solve these issues. Lakshminarayana et al study the effects of thread and memory scheduling on GPU performance in their work. They vary the fetch and memory scheduling policies and analyze the performance of GPU kernels [15].

Chen et al report that the traditional round-robin scheduling algorithm used in GPUs are inefficient in handling instruction execution and memory accesses with disparate latencies. They propose a wavefront scheduling algorithm that enables flexible round-robin distance for efficiently utilizing multi-threaded parallelism and they use a program-guided priority shift among concurrent threads (wavefronts) to allow for more overlaps between short-latency compute instructions and long-latency memory accesses [10].

Narasiman et al propose two-level warp [3] scheduling [21]. They observe that a scheduling policies such as the round-robin policy usually result in warps arriving at the same long latency operations at about the same time and this results in having no more warps to occupy the execution units, thus unable to hide this long latency memory operation. To increase the number of active functional unit cycles, they split all concurrently executing warps into fetch groups. Warps in a single fetch group are prioritized until they reach a single stalling point (long latency operation). Then the next fetch group is chosen and the execution continues in the same manner. The warps within the fetch group are scheduled in round robin order and the switch from one fetch group to another is also done in round robin fashion [21]. The motivation of this scheme is that each fetch group reaches a long latency operation at different points. As a result, when warps in one fetch group are stalled, warps from a different fetch group can execute thereby tolerating the long latency operation [21].

Rogers et al propose the idea of Cache-Conscious Wavefront scheduling. To do this, they use an adaptive mechanism that makes use of an intra-wavefront locality detector to capture locality lost by other schedulers due to excessive contention for cache capacity [27]. Their scheme shapes the access pattern to avoid thrashing in the L1 cache. Their policy improves the performance of the cache and as a result increases the utilization of the execution units. Adwait et al propose a CTA aware two level warp scheduling as well as a locality aware scheduler. Their two level warp scheduler works similar to that proposed by nasariman et al while their locality aware scheduler works by prioritizing execution of a

[3]A warp is a group of 32 threads

group of CTAs in the core. The motivation is to further exploit the locality between nearby warps [14].

Minseouk et al propose a lazy CTA [4] scheduler which works by restricting the amount of thread blocks allocated to each core. The lazy CTA scheduler works by predicting the optimal number of CTAs that should be scheduled on the core in order to make more efficient use of resources on the core. They also propose a block CTA scheduler which works by scheduling blocks of consecutive CTAs to the same core with the aim of exploiting the locality between ctas [17]. Similar to other proposals, their work enhances per core performance by reducing cache contention and improving latency hiding capability.

Gebhart et al point out that the massive multi-threading of a GPU requires a complicated thread scheduler as well as a large register file which is expensive to access both in terms of energy and latency. To mitigate these drawbacks, they propose the idea of register file caching to replace accesses to the large main register file with accesses to a smaller memory containing the immediate working set of the active threads. Similar to the aforementioned works, they also present the idea of a two-level wavefront scheduler. Though they approach the issue from a view of energy consumption reduction, their ideas can also be leveraged to work from a standpoint of increasing GPU resource utilization [11].

[4]CTA: Cooperative Thread Array. This is a group of thread blocks (warps). It is NVIDIA's terminology for a Workgroup.

CHAPTER 2

TECHNICAL BACKGROUND

In this chapter, we discuss technical background that will help readers understand our work. This chapter is organized as follows. In section 2.1 we discuss the basics of GPU computing. In section 2.2 we present the GPU architecture used in our work. Next, we discuss the thread execution model in the GPU. Lastly, in section 2.4, we discuss the architectural simulator used for our work.

2.1 GPU computing

Recently, the graphics processing unit (GPU) has become an integral part of mainstream computing systems. Over past couple of years, there has been steady and substantial increase in the performance and capabilities of GPUs. Modern GPUs are not only powerful graphics engines but are also highly parallel programmable processors whose peak arithmetic computing capability and memory bandwidth substantially out-pace its CPU counterpart. This rapid increase in both programmability and capability of modern GPUs enabled the research community to successfully map a broad range of computationally demanding, complex problems to the GPU. This effort to use the GPU for general purpose computing is called general purpose computing on a graphics processing unit (GPGPU) [24]. Figure 2.1

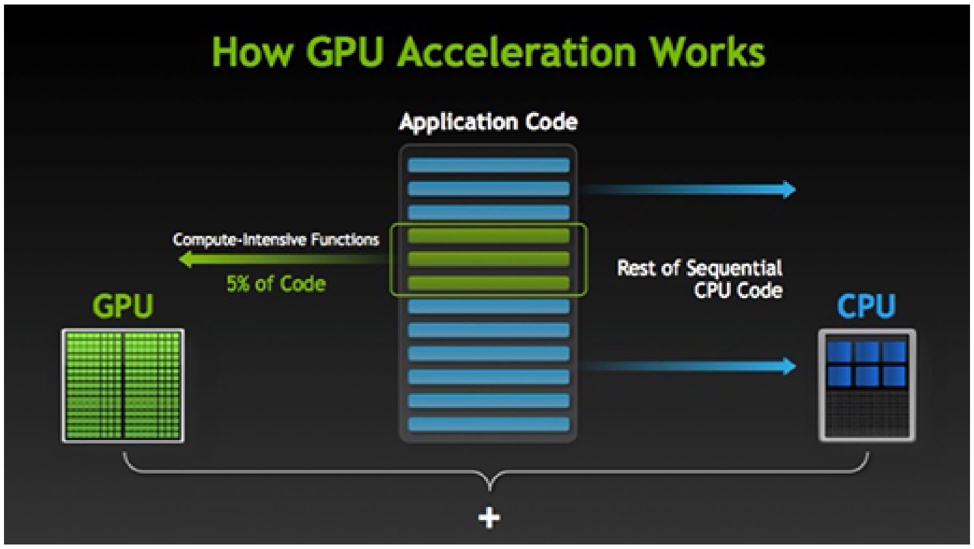

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
Rest of Sequential CPU Code
GPU
CPU
+

Figure 2.1. Application acceleration using a GPU.

shows how a GPU is used to accelerate an application.

Not all general purpose applications are suitable for GPU acceleration. In order for an application to benefit from GPGPU it is recommended to have the following characteristics.

- Data parallel: This means that a processor can execute the operation on different data elements simultaneously. For an application to map to the GPGPU computing model, it has to have this property [7].

- Throughput intensive: This means that the application is going to process a lot of elements. This property enables the algorithm to take advantage of the GPUs massive number of compute cores [7].

CPUs consist of a few computation cores that are optimized for sequential processing. On the other hand GPUs consist of thousands of light-weight cores that are optimized for performing multiple ALU operations simultaneously. Figure 2.2 illustrates the difference in the number of cores found in a GPU compared to in a CPU. In traditional graphics applications, programmer writes a single program, and the GPU runs multiple instances of the program in parallel, drawing massive number of pixels in parallel [22]. This parallel

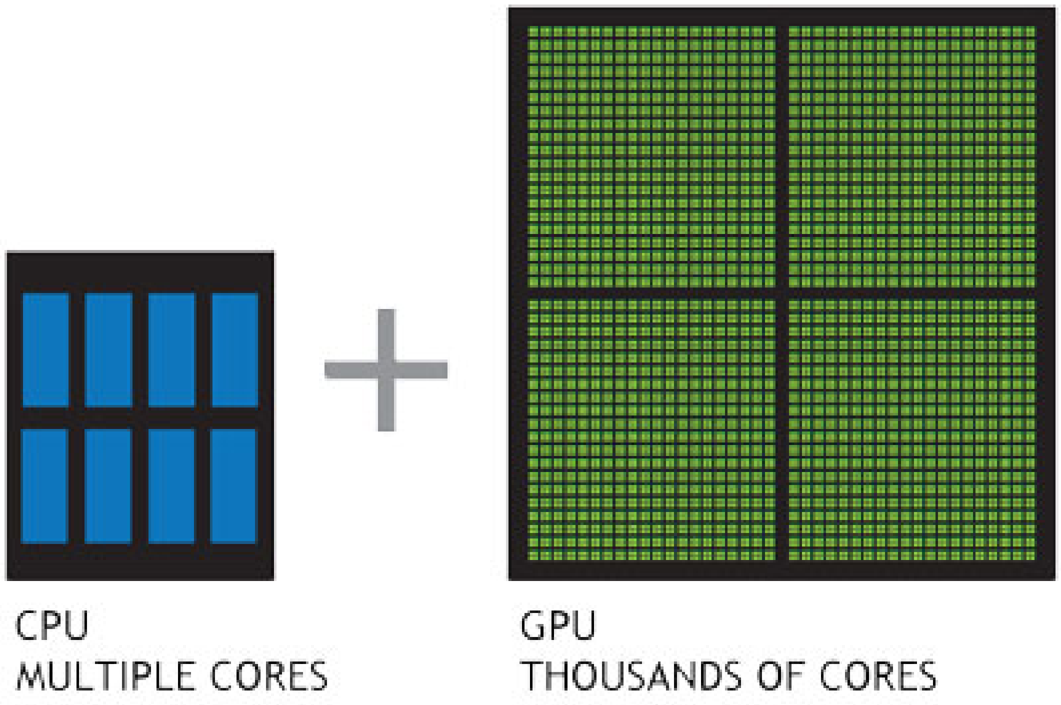


Figure 2.2. Multi-core CPU and many-core GPU.

execution of threads is the idea used in the GPGPU computing paradigm. The GPU executes multiple instances of a non-graphics program, operating on different data points [23].

By successfully mapping many general purpose applications to the GPU, scientists have gotten impressive results that show the GPU outperforms the CPU by a huge margin for certain applications. Meel et al report up to 150x speedup for Molecular Dynamics simulations [34]. For image correlation, Lu et al report achieved speedups of up to 130x [19]. In addition to these works, similar results have been published by other research.

2.2 GPU hardware architecture

With high demand of real time graphics rendering and general purpose computation support, GPU hardware has evolved from a fixed-function special-purpose processor to a full-fledged parallel programmable processor [24]. The GPU targeted in this thesis work, named Graphics

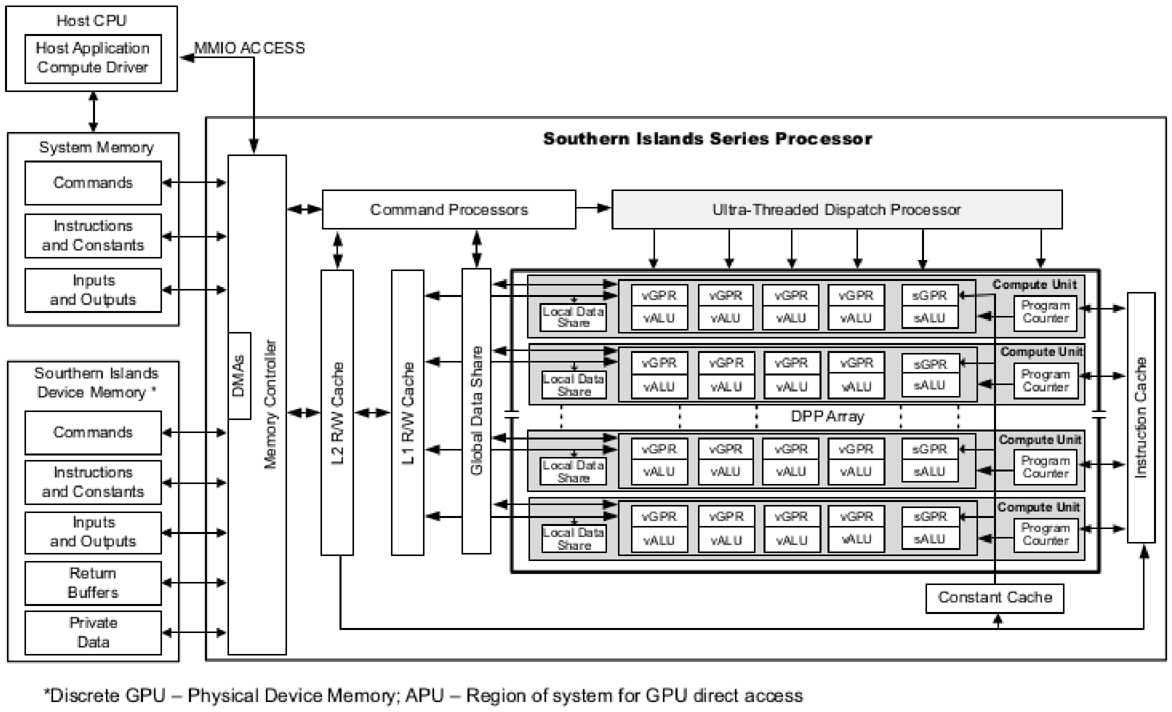


Figure 2.3. AMD Graphics Core Next (GCN) GPU architecture [3].

Core Next (GCN) architecture developed by AMD, is one of such architectures.

The GCN architecture is a completely new architecture designed with a focus on improved general purpose workload performance and better power efficiency while improving graphics experience [33]. It implements a parallel micro-architecture that provides an excellent platform for both graphics and general-purpose applications [3]. In this new architecture, all levels of the GPU from the ISA to the processing elements, to the memory system have been redesigned [33]. To improve performance, AMD shifted from VLIW architecture to scalar-vector hybrid architecture. The reason for this was that while VLIW is very good for processing graphics instructions, it is not good at handling the scalar instructions often found in general purpose applications. This is because VLIW bundles instructions during compilation but dependencies among scalar instructions limits bundling, leading to underutilization of the execution units and wasted clock cycles at run time.

Figure 2.3 illustrates GCN GPU architecture. It consists of a command processor that communicates with the host (CPU) and schedules on-chip workloads. Commands from

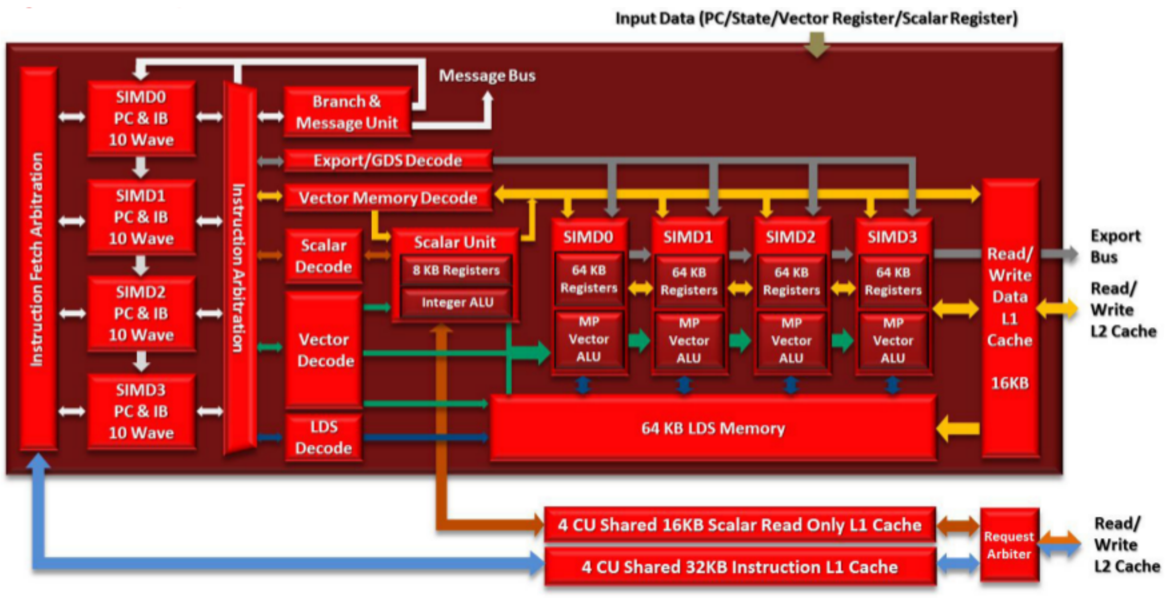


Figure 2.4. AMD GCN Compute Unit [2].

this unit are received by the ultra-threaded dispatch processor which then distributes the work across the compute units (CUs). The CU is a key new component responsible for GCN's improved performance. It is a basic building block of GCN architecture consisting of four SIMD engines together with other functional units. Figure 2.4 shows the details of a GCN CU.

CUs are independent of each other and operate in parallel. Each CU contains instruction logic (fetch, buffer, decode, issue), scalar and vector ALU units with private scalar GPRs and vector GPRs respectively, a high-bandwidth, low-latency local memory (Local Data Store (LDS) in AMD's terminology), and a read/write L1 cache. 4 CUs also share and instruction and constant cache. A read/write L2 cache, a global memory, and memory controllers support the CUs and provide support for the data accessibility necessary to support kernel execution [3].

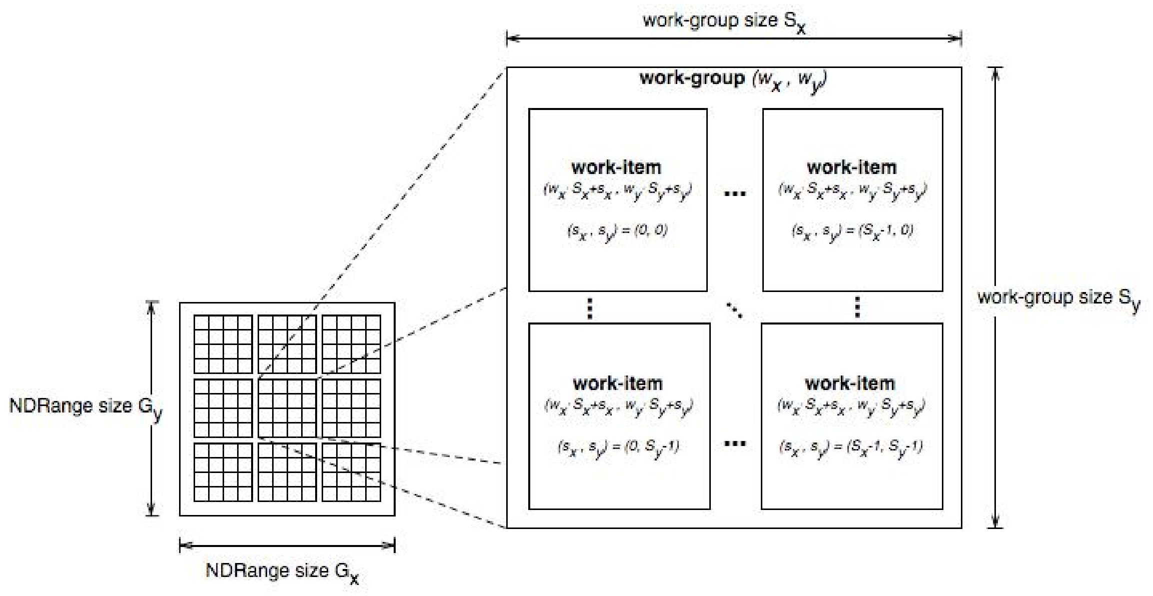

work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, S_y - 1)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, S_y - 1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

Figure 2.5. ND-Range organization [1].

2.3 GPU thread execution model

2.3.1 Kernel Execution

An instance of a kernel is called an ND-Range. An ND-Range, shown in figure 2.5, is comprised of workgroups and these workgroups are further comprised of work-items (threads). The program and execution models of an ND-Range are mapped onto the hardware when it is launched by the graphics device driver. The ultra-threaded dispatcher consumes pending workgroups from the running ND-Range and schedules them to available compute units. The compute units partition the workgroups into wavefronts (sets of 64 work-items). Wavefronts execute instructions in SIMD (single instruction multiple data) fashion. The wavefront scheduler in a CU is responsible for scheduling wavefronts among its 4 SIMD execution units as the units become available. Each SIMD execution unit contains 16 lanes (stream cores). Each SIMD core executes one instruction for 4 work-items, at speed of 1 cycle per work-item, from a wavefront mapped to it. This is done in a time- multiplexed manner and as a result, each SIMD can execute a wavefront in 4 cycles [2][33].

Work-items within a workgroup can share information using a mapped portion of the local memory present on a CU. Each work-item has its private memory which is physically mapped to a portion of the register file. If the work-item uses more memory than allocated on the register file, register spills occur and are handled by using privately allocated portions of global memory.

2.3.2 Workgroup scheduling

When GPU starts execution of the kernel, the workgroups in the NDRange are mapped to the different CUs available on the GPU. This scheduling is done by the ultra threaded dispatcher, The ultra thread dispatcher consumes pending workgroups and assigns them to the CUs when they become available. The default scheduling is done in round robin and first available fashion. This can be implemented in a way that all available compute units are stored in a list and the workgroups are assigned to the compute units in this list in round robin fashion. This list of available compute units is updated every cycle with compute units that become newly available.

Algorithm 1 First available workgroup scheduling

```
while $programIsExecuting$ do
    while $isWaitingWorkgroups$ do
        if $computeUnitIsAvailable$ then
            assignWorkgroupToComputeUnit
        end if
    end while
end while
```

2.3.3 Wavefront scheduling

After command to begin kernel execution is received, the kernel is fetched into the instruction cache and the compute unit begins sending instructions to their respective execution units (e.g SIMD units, scalar ALU, memory units, etc). Each CU runs one or more workgroups at a time and since all work-items in the workgroup run identical code, they are combined

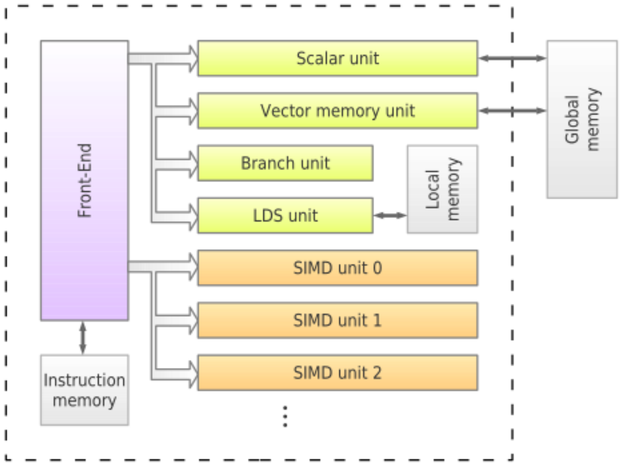


Figure 2.6. Simplified block diagram of AMD GCN CU [33].

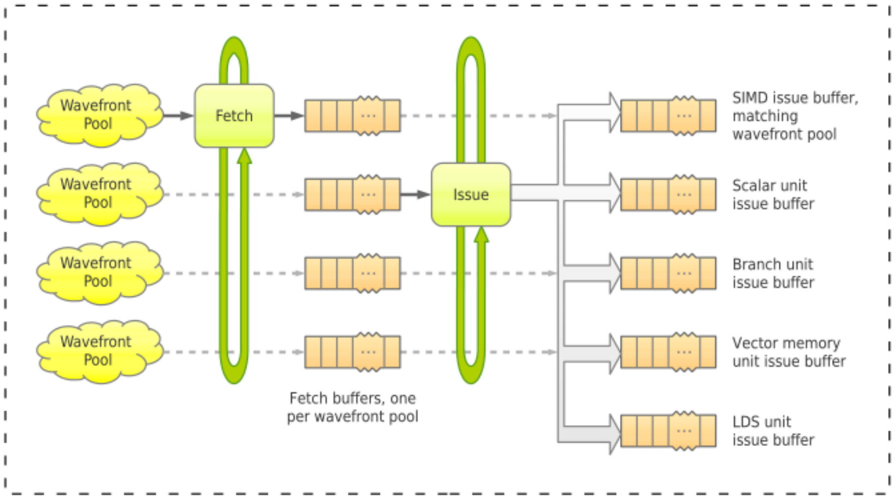


Figure 2.7. Pipeline stages in the compute unit front-end [33].

into sets of 64. A set of 64 work-items is called a wavefront. Each CU can work on multiple wavefronts in parallel, simultaneously executing different instruction types [3].

The compute unit front end is responsible for scheduling wavefront instructions to their respective execution units. Figure 2.6 shows the location of the front end in a simplified illustration of a CU. Figure 2.7 shows the details of a compute unit front end.

The front end consists of a set of wavefront pools, a fetch stage, a set of corresponding fetch buffers, and an issue stage. The number of wavefront pools and fetch buffers equals the number of SIMD units. When a workgroup is first mapped to a compute unit, its corresponding wavefronts are assigned to an available wavefront pool (all wavefronts in the workgroup are assigned to the same wavefront pool). During the fetch stage, the oldest wavefront in the pool is selected, instruction bytes are read from instruction memory at the wavefront's current program counter, and these bytes are put in the corresponding fetch buffer. In the next cycle, the fetch stage operates on the next wavefront pool containing an available wavefront, following a round-robin order [33].

In the issue stage, instructions are consumed from the fetch buffer in a round-robin order. These instructions are then distributed to the private issue buffers of their corresponding execution units where they wait for execution. Arithmetic vector instructions are sent to the SIMD unit mapped to the wavefront pool it was fetched from. Any other type of instruction is sent to shared instances of the scalar, branch, LDS or vector memory units.

Algorithm 2 Round-Robin wavefront scheduling

```
while $programIsExecuting$ do
    $activeFetchBuffer = currentGPUcycle \mod numWavefrontPools$
    while $issuedInstructions < maxInstructionsPerType$ do
        $fbEntries = numberOfInstructionsInActiveFetchBuffer$
        $oldestInst = NULL, i = 0$
        while $i < fbEntried$ do
            $inst = instructionAtIndex\ i\ inActiveFetchBuffer$
            if $inst\ isInvalidForAnyReason$ then
                $continue$
                $i++$
            end if
            if $instructionIsOlderThan\ oldestInst\ or\ oldestInst = NULL$ then
                $oldestInst = inst$
            end if
            $i++$
        end while
        if $oldestInst \neq NULL\ and\ destinationIssueBufferIsNotFull$ then
            $issueTheInstructionToAppropriateIssueBuffer$
        end if
    end while
end while
```

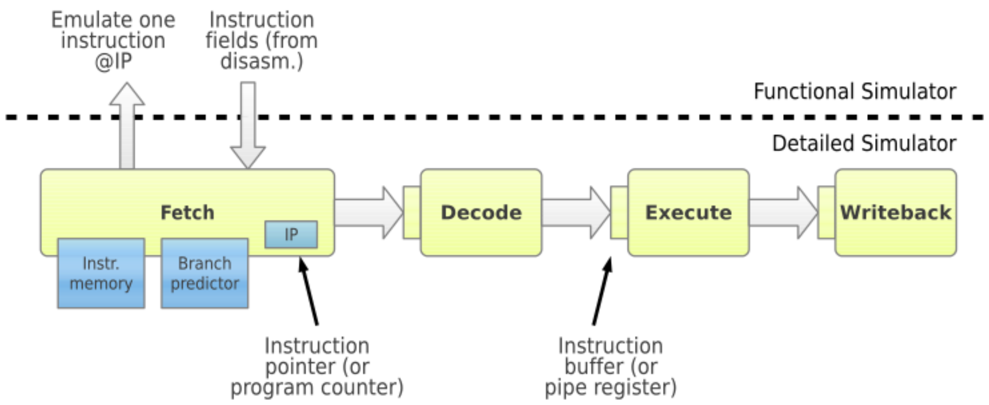

Emulate one instruction @IP
Instruction fields (from disasm.)
Functional Simulator
Detailed Simulator
Fetch
IP
Instr. memory
Branch predictor
Decode
Execute
Writeback
Instruction pointer (or program counter)
Instruction buffer (or pipe register)

Figure 2.8. Interaction between the functional and timing simulators in Multi2Sim [33].

2.4 Multi2sim Simulator

Multi2Sim is a cycle accurate simulation framework for CPU-GPU heterogeneous computing. It currently models superscalar, multithreaded, and multicore CPUs as well as GPU architectures. At the time of this writing, it currently models Nvidia Fermi, AMD GCN and AMD Evergreen architectures.

The framework consists of four independent software modules. They include a disassembler, functional simulator (emulator), timing simulator (detailed/ architectural) and a visual tool. These modules can work independently or communicate with each other. For this research, the architectural simulator was used.

The functional simulator is basically an emulator that reproduces the behavior of a program giving the illusion that it is running on a given micro-architecture. It can be used to execute a program from start to completion, or serve as an interface for the architectural simulator.

The architectural simulator models hardware structures and keeps track of their access times. Among the modeled hardware components include pipeline stages, pipe registers, instruction queues, functional units, cache memory, etc. The flow of instructions used in the architectural simulator is obtained from calls to the functional simulator. The figure 2.8 illustrates the interaction between the architectural and functional simulators.

Instructions are executed in order. When the architectural simulator detects free space in the fetch/decode pipe register, it fetches a new instruction from the address determined by the program counter. It calls for the execution of this new instruction from the functional simulator which then returns all information about the emulated instruction.

After the information about the emulated instruction is received by the architectural simulator, the instruction is propagated through the pipeline stages where it accesses different models of hardware resources (functional units, effective address calculators, data caches, etc) with potentially different latencies [33].

CHAPTER 3

THREAD SCHEDULERS

The goal of the thesis was to investigate the impact of various thread scheduling schemes on GPU performance. To this end, we implemented the following schedulers on the Multi2Sim architectural simulator.

- Lookahead wavefront scheduler
- Two-level wavefront scheduler
- Two-level + Lookahead scheduler
- Block Workgroup scheduler

The wavefront schedulers focus on improving the issue stage in the CU front end. For workgroup scheduling, focus is placed on improving the ultra threaded dispatcher logic. In the following sections, we discuss these schedulers and give details of their implementation.

3.1 Lookahead wavefront scheduler

The default (i.e., baseline) round-robin wavefront scheduler tries to issue a wavefront from a chosen fetch buffer in a particular cycle for each type of instruction. If there is an unsuccessful issue, the scheduler waits till the next cycle to try to issue from the next fetch buffer. However, there may be valid instructions in the other fetch buffers ready to be issued. The proposed lookahead scheduler aims to exploit this and searches other fetch buffers for a valid instruction if there is an unsuccessful issue from the current fetch buffer. This search is done for all instruction types. Algorithm 3 shows the algorithm for this scheduler.

Algorithm 3 Lookahead wavefront scheduling

```
while $kernelIsExecuting$ do
    $success = 0,\ j = 0$
    while $j < numWavefrontPools$ do
        while $issuedInstructions < maxInstructionsPerType$ do
            $fbEntries = numberOfInstructionsInActiveFetchBuffer$
            $oldestInst = NULL, i = 0$
            while $i < fbEntries$ do
                $inst = instructionAtIndex\ i\ inActiveFetchBuffer$
                if $inst\ isInvalidForAnyReason$ then
                    $i++$
                    $continue$
                end if
                if $instIsOlderThan\ oldestInst\ or\ oldestInst = NULL$ then
                    $oldestInst = inst$
                end if
                $i++$
            end while
            if $oldestInst \neq NULL\ and\ destinationIssueBufferIsNotFull$ then
                $issueTheInstructionToAppropriateIssueBuffer$
                $sucess++$
                $issuedInstructions++$
            end if
        end while
        if $success = 1$ then
            $break$
        end if
        $setActiveFetchBuffer\ to\ theNextFetchBuffer\ ifNotYetVisited$
        $j++$
    end while
end while
```

3.2 Two-level wavefront scheduler

This scheduler is based on the ideas proposed in several papers [21] [14]. The key idea is to split in-flight wavefronts into *fetch groups*. Wavefronts in a fetch group are scheduled in round robin order until all the wavefronts in the fetch group stall. The next fetch group is then selected and the wavefronts from it are scheduled again in round robin fashion. This is continued in this manner, hence the name - *two-level wavefront scheduling*. In the proposed two-level scheduler, we use each fetch buffer as a fetch group. Wavefronts from each fetch buffer are scheduled in round-robin order until they all stall. Then the next fetch buffer (i.e., fetch group) is selected for scheduling. Algorithm 4 details our implementation of the two-level wavefront scheduler.

Algorithm 4 Two level wavefront scheduling

```
while $kernelIsExecuting$ do
    $stall = 1$
    while $issuedInstructions < maxInstructionsPerType$ do
        $fbEntries = numberOfInstructionsInActiveFetchBuffer$
        $oldestInst = NULL, i = 0$
        while $i < fbEntries$ do
            $inst = instructionAtIndex\ i\ inActiveFetchBuffer$
            if $inst\ isInvalidForAnyReason$ then
                $i++$
                $continue$
            end if
            if $instIsOlderThan\ oldestInst\ or\ oldestInst = NULL$ then
                $oldestInst = inst$
            end if
        end while
        if $oldestInst \neq NULL\ and\ destinationIssueBufferIsNotFull$ then
            $issueTheInstructionToAppropriateIssueBuffer$
            $issuedInstructions++$
            $stall = 0$
        end if
    end while
    if $stall = 1$ then
        $setActiveFetchBuffertotheNextFetchBuffer$
    end if
end while
```

3.3 Lookahead + Two-level wavefront scheduler

This scheduler combines the lookahead and two-level wavefront schedulers. Here, scheduling is done in a round robin fashion from a particular fetch buffer until all the wavefronts in the fetch buffer stall. However, if there is an unsuccessful issue for a particular instruction type, the scheduler checks for that instruction type in the other fetch buffers. It is important to note that this lookahead portion does not permanently change the selected fetch buffer for issue. It just checks them for valid instructions and schedules if it finds one, reverting to the originally selected fetch buffer after this search. Algorithm 5 shows the operation of this scheduler.

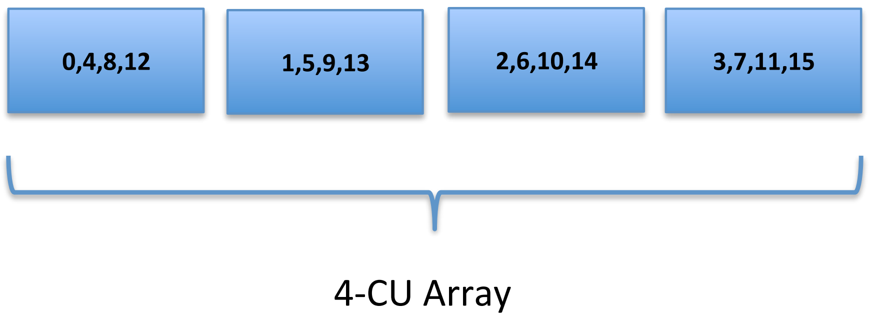
3.4 Block Workgroup scheduler

The aim of the block workgroup scheduler is to exploit the spatial locality between workgroups. The default first-available policy destroys this locality and ends up scattering these workgroups among the compute units. Our workgroup scheduler addresses this issue by assigning contiguous workgroups to each CU. We use a delayed WG scheduling in situations where a calculated destination compute unit is not available. By delayed, we mean that we put off scheduling the workgroup to that CU until it becomes available. Figure 3.1 illustrates the difference between the default first-available workgroup scheduler and our workgroup scheduler.

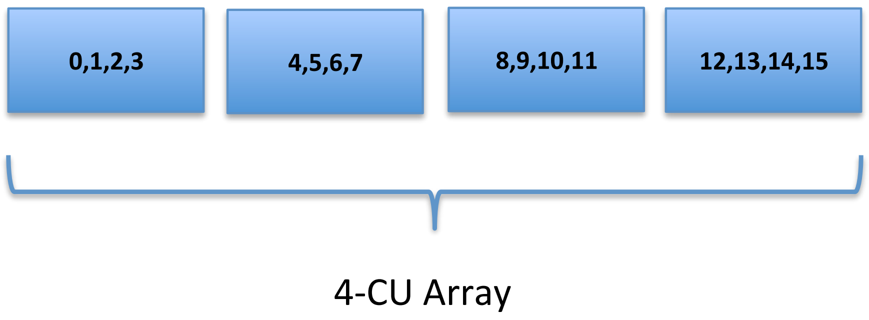
Algorithm 6 describes the operation of this scheduler.

Algorithm 5 Lookahead wavefront scheduling

```
while $kernelIsExecuting$ do
    $success = 0,\ j = 0,\ stall = 0$
    while $j < numWavefrontPools$ do
        while $issuedInstructions < maxInstructionsPerType$ do
            $fbEntries = numberOfInstructionsInActiveFetchBuffer$
            $oldestInst = NULL, i = 0$
            while $i < fbEntried$ do
                $inst = instructionAtIndex\, i\, inActiveFetchBuffer$
                if $inst\, isInvalidForAnyReason$ then
                    $i++$
                    $continue$
                end if
                if $instIsOlderThan\, oldestInst\, or\, oldestInst = NULL$ then
                    $oldestInst = inst$
                end if
                $i++$
            end while
            if $oldestInst \neq NULL\, and\, destinationIssueBufferIsNotFull$ then
                $issueTheInstructionToAppropriateIssueBuffer$
                $sucess++$
                $issuedInstructions++$
            end if
        end while
        if $success = 1$ then
            $break$
        end if
        $setActiveFetchBuffer\, to\, theNextFetchBuffer\, ifNotYetVisited$
        $j++$
    end while
    if $stall = 1$ then
        $setActiveFetchBuffertotheNextFetchBuffer$
    end if
end while
```



(a) First-available scheduling



(b) Block scheduling

Figure 3.1. Workgroup scheduling

Algorithm 6 Block workgroup scheduling

```
while $kernelIsExecuting$ do
    $i = 0$
    while $count(waitingWorkgroupList) \neq 0$ do
        $wg = waitingWorkgroupList[i]$
        $destinationComputeUnit = floor(workgroupID/(numberOfWorkgroups/numberOfCUs))$
        if $destinationComputeUnitIsAvailable$ then
            $removeWorkgroupFromWaitingListAndAssignToComputeUnit$
            $i = 0$
        else
            $i++$
        end if
    end while
end while
```

CHAPTER 4

EXPERIMENTS AND RESULTS

In this chapter, we present the experimental results and analysis of the proposed thread schedulers. This chapter is organized as follows. In section 4.1 we present the architectural configuration used in our experiments. Next, we discuss the performance metrics and benchmarks used in our experiments in section 4.2 and 4.3 respectively. Finally we elaborate and compare the performance of each scheduler in section 4.4.

4.1 Architectural specifications

Table 4.1 shows in-depth architectural parameters used in our experiments. These parameters closely resemble the hardware specification of AMD HD 7970 (Southern Island). The configuration uses 32 compute units and clock frequency of 925MHz. As units, *BufferSize* is measured in *instructions*, *IssueWidth* is the number of instructions that can be issued in a cycle for that execution unit, and *latency* is the number of cycles to take for an operation to complete.

Table 4.1. GPU architectural configuration

Device	
Frequency	925
NumComputeUnits	32

ComputeUnit	
NumWavefrontPools	4
NumVectorRegisters	65536
NumScalarRegisters	2048
MaxWorkGroupsPerWavefrontPool	10
MaxWavefrontsPerWavefrontPool	10

FrontEnd	
FetchLatency	1
FetchWidth	1
FetchBufferSize	10
IssueLatency	1
IssueWidth	5
MaxInstIssuedPerType	1

SIMDUnit	
NumSIMDLanes	16
Width	1
IssueBufferSize	1
DecodeLatency	1
DecodeBufferSize	1
ReadExecWriteLatency	8
ReadExecWriteBufferSize	2

ScalarUnit	
Width	1
IssueBufferSize	1
DecodeLatency	1
DecodeBufferSize	1
ReadLatency	1
ReadBufferSize	1
ALULatency	4
ExecBufferSize	32
WriteLatency	1
WriteBufferSize	1

BranchUnit	
Width	1
IssueBufferSize	1
DecodeLatency	1
DecodeBufferSize	1
ReadLatency	1
ReadBufferSize	1
ExecLatency	16
ExecBufferSize	16
WriteLatency	1
WriteBufferSize	1

LDSUnit	
Width	1
IssueBufferSize	1
DecodeLatency	1
DecodeBufferSize	1
ReadLatency	1
ReadBufferSize	1
MaxInflightMem	32
WriteLatency	1
WriteBufferSize	1

VectorMemUnit	
Width	1
IssueBufferSize	1
DecodeLatency	1
DecodeBufferSize	1
ReadLatency	1
ReadBufferSize	1
MaxInflightMem	32
WriteLatency	1
WriteBufferSize	1

LDS	
Size	65536
AllocSize	64
BlockSize	64
Latency	2
Ports	2

4.2 Performance metrics

Three performance metrics are used in this research. We explain each metric in the following sections.

4.2.1 Cache hit rate

Cache is a fast on-chip memory space between processor and main memory [31]. This cache memory is to exploit temporal and spatial locality present in programs by keeping frequently used data or instructions in faster space. The cache hit rate is denoted as the fraction of cache accesses that result in a hit [1] as calculated in equation 4.1 [13]. Since our target GPU (i.e., AMD GCN) has a private L1 cache per CU, we get an average hit rate of all the L1 caches using equation 4.2.

$$Hit\,rate = \frac{number\ of\ cache\ hits}{total\ number\ of\ cache\ accesses} \tag{4.1}$$

$$Average\ hit\ rate = \frac{\sum_{i=0}^{k-1} \frac{number\ of\ cache\ hits_i}{total\ number\ of\ cache\ accesses_i}}{number\ of\ CUs} \tag{4.2}$$

where k is the compute unit index

Theoretically, the higher the hit rate, the faster the program executes because it yields less number of trips to high-latency main memory. Conversely, the lower the hit ratio, the more long latency requests that have to be made to main memory, thus resulting slower execution time.

4.2.2 Utilization of issue unit

The utilization of hardware issue unit (i.e., issue rate) has close relationship with hardware resource utilization; the higher rate of instruction issue in the CU front-end, the higher utilization of the CU's resources. To have an idea of the rate of instruction issue, we keep track of clock cycles where the issue unit in the CU successfully issues an instruction. When there is a successful instruction issue, we call this an *active issue cycle*. On the contrary, when there is no successful instruction issue, we call this an *idle issue cycle*. Equation 4.3 shows the calculation of the idle issue cycles for one functional unit in the CU. For brevity, we use IIC to refer to idle issue cycles in the following equations.

[1]A hit means that the data requested is found in the cache.

$$Idle\ issue\ cycles(\%) = \frac{(total\ CU\ cycles - active\ issue\ cycles\ (for\ an\ execution\ unit))}{total\ CU\ cycles} \tag{4.3}$$

Since we have 5 functional units in a CU, we average the sum of the idle issue cycles to each functional unit using equation 4.4 .

$$CU_{IC}(\%) = \frac{branch_{IIC}\ + scalar_{IIC}\ + vector\_memory_{IIC}\ + lds_{IIC}\ + simd_{IIC}}{5} \tag{4.4}$$

Finally, to get the idle issue cycles for the device, we average the CU_{IIC} across all 32 compute units using equation 4.5.

$$Device_{IIC} = \frac{\sum_{i=0}^{k-1} CU_k\ idle\ issue\ cycles}{number\ of\ active\ CUs} \tag{4.5}$$

where k is the compute unit index

4.2.3 Instructions per cycle

Computer architects heavily use Instructions Per Cycle (IPC) to evaluate computer system performance. IPC is the average number of instructions executed per clock cycle and it gives a sense of the overall instruction throughput in the system [4]. In order for this metric to be used to compare performance of different systems, the number of dynamic instructions per program and clock cycle time have to be the same across the different systems. Our experiments meet these requirements. Equations 4.6 and 4.7 show how IPC represents system performance compared to the gold standard - *execution time.* Assuming two systems A and B,

$$Speedup(A) = \frac{(time/program)_B}{(time/program)_A} \tag{4.6}$$

$$time/program = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle} \tag{4.7}$$

Since the first and last terms are constant in our case in equation 4.7, we infer that execution time is a function of the second term only - *cycles/instruction* and as such, *instructions/ cycle.*

This speedup formula is justified in this study because our simulator generates deterministic results (no operating system or interrupts) and provides the same results for every run on given inputs. Instruction per program for the device and compute units are also always the same for every run on the same input. As a result, we use the IPC to evaluate the impact of the different schedulers and gain valuable insight on the architectural performance.

4.3 Benchmarks

This section presents the benchmarks used in this research. We used two benchmark suites: AMD SDK [3] and Rodinia [9]. For each benchmark in both suites, we used two different input sizes to have a certain number of workgroups in the NDRange. This was done so that we have two cases in terms of hardware occupancy - *unsaturated and saturated.*

- *Saturated:* This case uses large input sizes for the benchmarks. This was done so that there will be a large number of workgroups in the GPU and also a large number of workgroups per compute unit. Doing this allows us to investigate the impact of the different thread scheduling techniques when the GPU resources are stressed to the limit.

- *Unsaturated:* This case uses small input sizes for the benchmarks. This was done so that there will be a small number of workgroups in the GPU and also a small number of workgroups per compute unit. Doing this allows us to investigate the impact of the different thread scheduling techniques when the GPU resources are underutilized.

4.3.1 Benchmark descriptions

Benchmarks play very important role in computer architecture research. They allow the research community to focus on a shared codebase and allow researchers to easily understand each other's results. If a set of benchmarks is generally accepted, it helps dispel concerns of bias that may arise if the researcher creates their own benchmark [30]. In light of this, we carefully chose a set of benchmarks generally accepted by the scientific community. They were written in OpenCL and are briefly described below together with figures of the dynamic instruction mix [2] in their kernels. Tables 4.2 and 4.3 summarize the benchmark configurations. Benchmarks that contain a significant portion of vector-memory instructions - i.e. memory intensive - during program execution are listed in the top half while non-memory intensive benchmarks are listed in the bottom half.

- *MatrixMultiplication:* This benchmark performs matrix multiplication operation on two input arrays and stores the result in an output array.

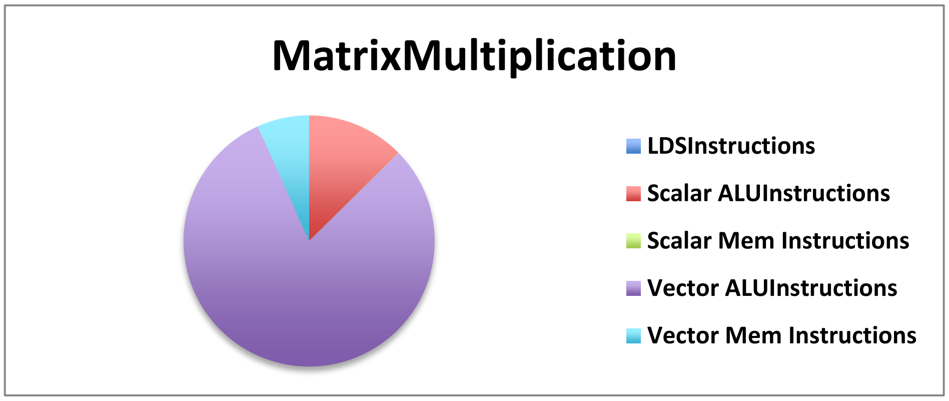


Figure 4.1. Dynamic instruction distribution by type of MatrixMultiplication.

[2]The dynamic instruction mix shows how much of each type of instruction is processed during kernel execution. For this study it enable us to see if a benchmark is memory intensive (i.e. a significant portion of processed instructions are memory instructions.)

- *MatrixTranspose:* This benchmark performs a matrix transpose on an input matrix and stores the result in an output matrix [28].

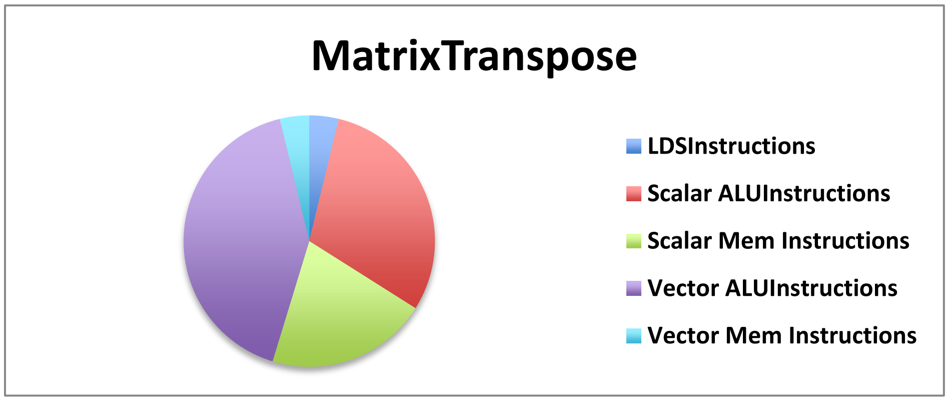


Figure 4.2. Dynamic instruction distribution by type of MatrixTranspose.

- *BlackScholes:* This benchmark provides the partial differential equation for the evolution of an option price under certain assumptions [5].

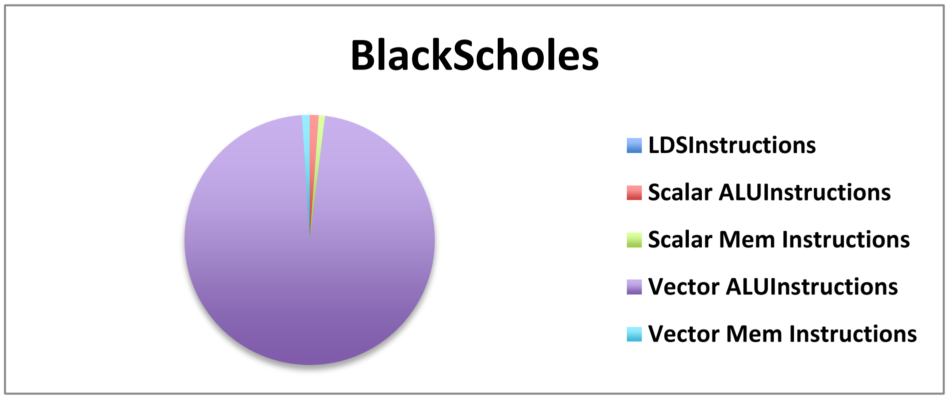


Figure 4.3. Dynamic instruction distribution by type of BlackScholes.

- *BinomialOption:* Option pricing is an important problem encountered in financial engineering. This benchmark implements the binomial option pricing for the European options [5].

- *BitonicSort:* This benchmark sorts an arbitrary sequence of numbers using the bitonic sort algorithm. It works by creating bitonic sub-sequences in the original array, starting

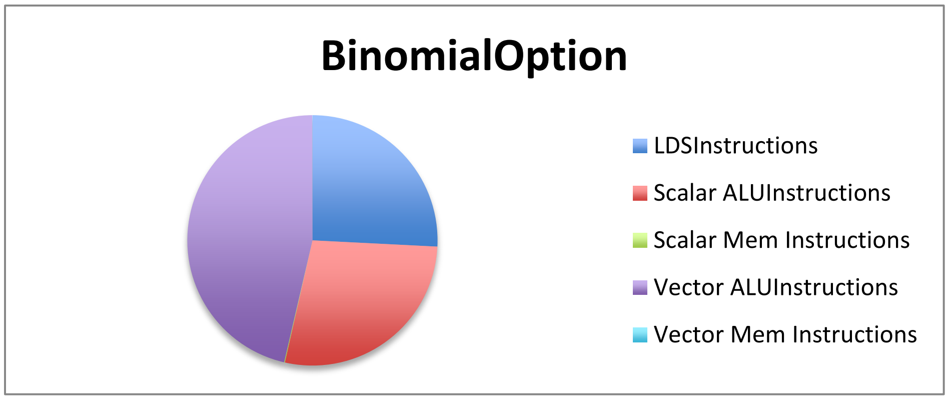


Figure 4.4. Dynamic instruction distribution by type of BinomialOption.

with sequences of size 4 and continuously merging the sub-sequences to create bigger bitonic subsequences [5][25].

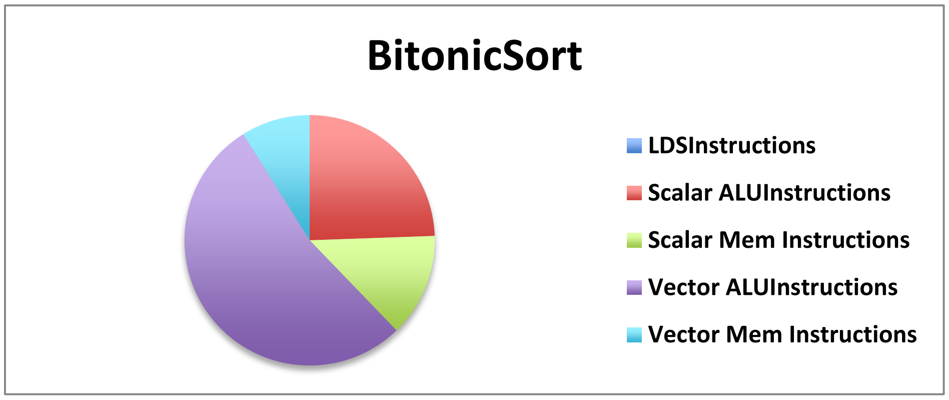


Figure 4.5. Dynamic instruction distribution by type of BitonicSort.

- *FastWalshTransform:* This benchmark implements an efficient version of the walsh transform that can be done in $O(nln(n))$ complexity [5].
- *ScanLargeArrays:* This benchmark implements a parallel prefix-sum algorithm [12] [16].
- *FloydWarshall:* This benchmark implements the Floyd Warshall algorithm. This algorithm computes the shortest path between each pair of node in a graph [20]. It uses a dynamic programming approach that iteratively refines the adjacency matrix of the graph in question until each entry in the matrix reflects the shortest path between the corresponding nodes.

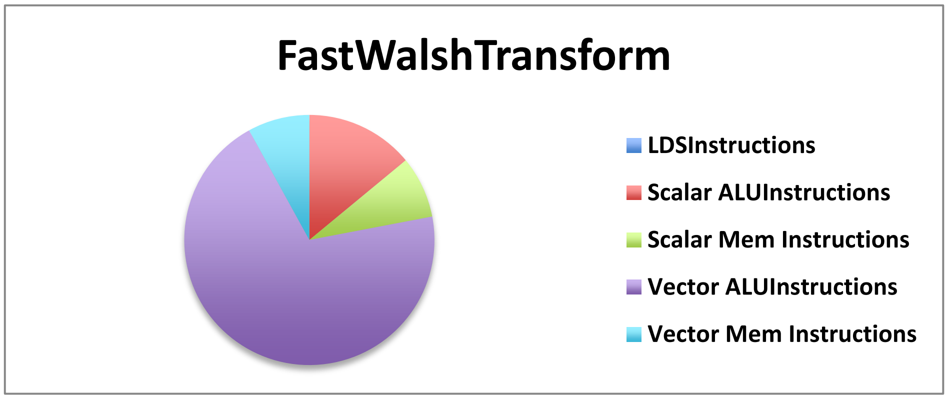


Figure 4.6. Dynamic instruction distribution by type of FastWalshTransform.

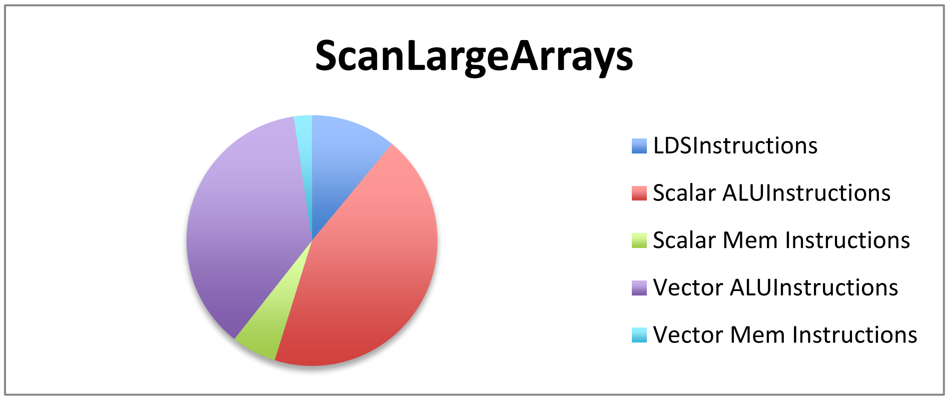


Figure 4.7. Dynamic instruction distribution by type of ScanLargeArrays.

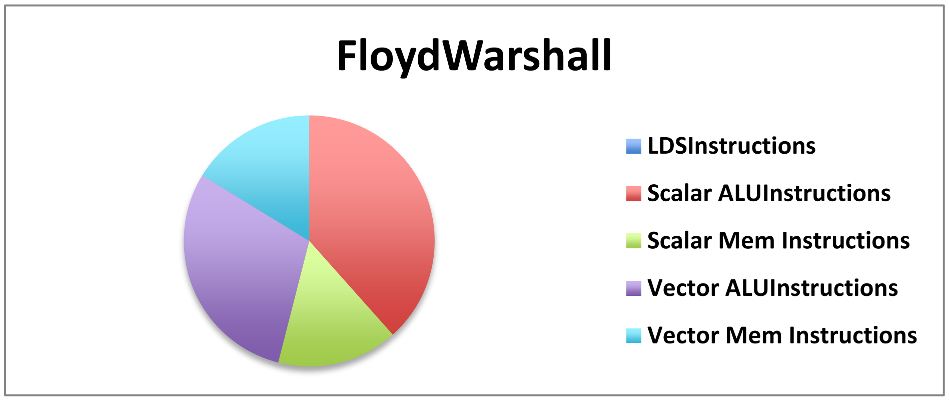


Figure 4.8. Dynamic instruction distribution by type of FloydWarshall.

- *RadixSort:* This benchmark implements a parallel radix sort algorithm. The implementation breaks keys (32 integers) into 8-bit digits and sorts one 8-bit digit at a time, starting with the least significant digit. It loops four times to complete sorting [5].

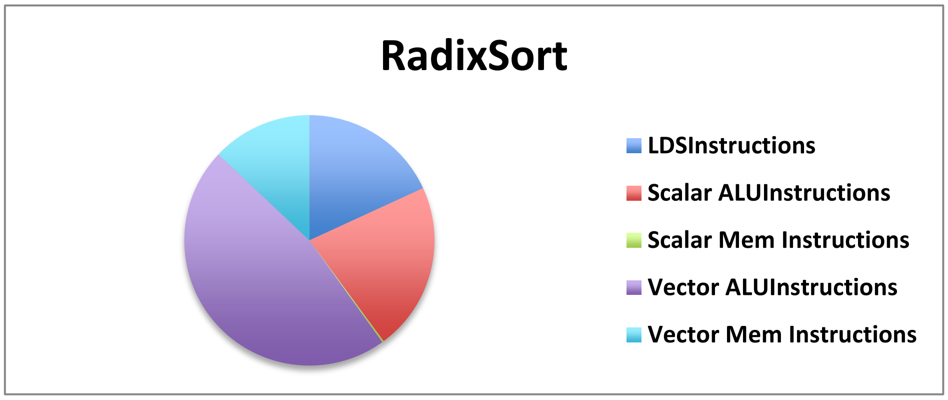


Figure 4.9. Dynamic instruction distribution by type of RadixSort.

- *Reduction:* This benchmark implements a parallel reduction algorithm.

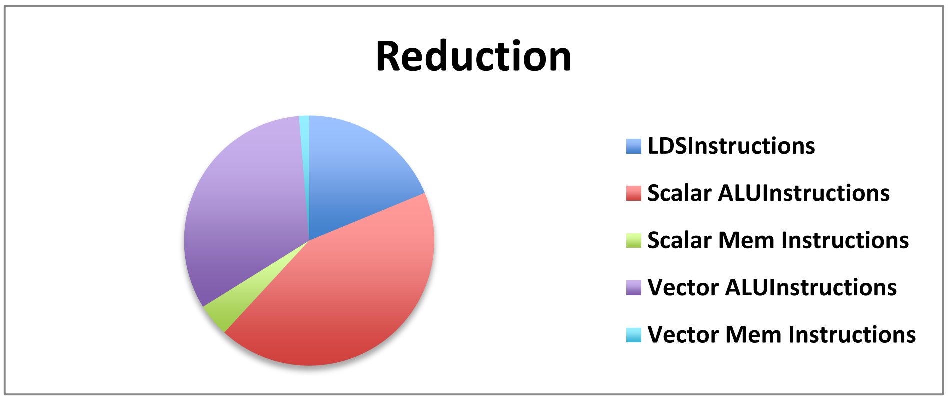


Figure 4.10. Dynamic instruction distribution by type of Reduction.

- *Hotspot:* This benchmark, gotten from the Rodinia benchmark suite, provides an implementation of Hotspot. HotSpot is a widely used tool to estimate the temperature of a processor based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations for block. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip [8] [26] [9].

- *BackPropagation:* This benchmark, gotten from the Rodinia benchmark suite, provides an implementation of Back Propagation. Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The

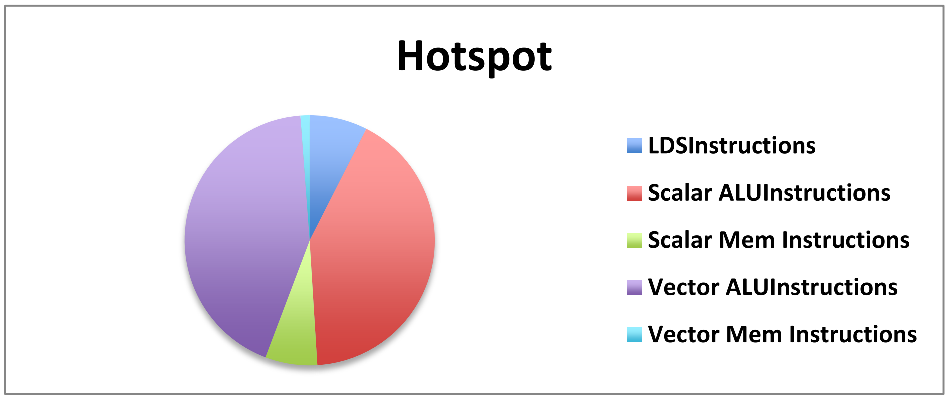


Figure 4.11. Dynamic instruction distribution by type of Hotspot.

application is comprised of two phases: the Forward Phase, in which the activations are propagated from the input to the output layer, and the Backward Phase, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. In each layer, the processing of all the nodes can be done in parallel [8] [26] [9].

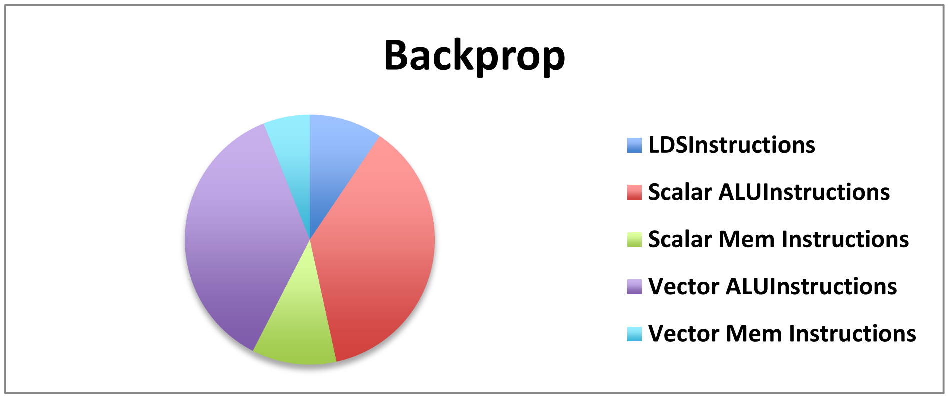


Figure 4.12. Dynamic instruction distribution by type of BackPropagation.

4.4 Performance Analysis

4.4.1 Impact on Cache hit rate

The block workgroup scheduler is proposed in section 3.4. This scheduler was designed to exploit the locality among workgroups with the goal of increasing the cache hit rate. In this

Table 4.2. Benchmark configuration (saturated)

Benchmark	**Abbr.**	**Size**	**WG count**	**WGs per CU**	**NDRange**
MatrixMultiplication	MM	1024	256	8	1
BitonicSort	Bso	2048	67584	2112	66
FastWalshTransform	FWT	262144	9216	288	18
FloydWarshall	FW	256	65536	2048	256
RadixSort	RS	524288	256	8	8
BackProp	BP	65536	8192	256	2
MatrixTranspose	MT	512	4096	128	1
BlackScholes	BSc	1048576	1024	32	1
BinomialOption	BO	1024	1024	32	1
ScanLargeArrays	Scan	524288	4113	131	5
Reduction	RD	1048576	512	16	1
Hotspot	HS	512	1849	58	1

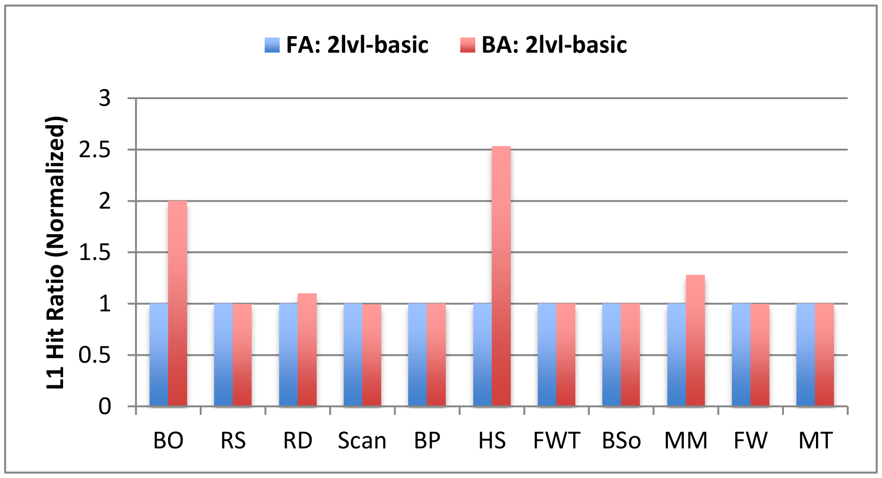
Table 4.3. Benchmark configuration (unsaturated)

Benchmark	**Abbr.**	**Size**	**WG count**	**WGs per CU**	**NDRange**
MatrixMultiplication	MM	512	64	2	1
BitonicSort	Bso	16	80	2,3, 10	10
FastWalshTransform	FWT	4096	96	3, 12	8
FloydWarshall	FW	8	64	2, 8	8
RadixSort	RS	13072	64	2, 8	8
BackProp	BP	512	64	2	2
MatrixTranspose	MT	128	64	2	1
BinomialOption	BO	64	64	2	1
ScanLargeArrays	Scan	8192	65	2,3	3
Reduction	RD	131072	64	2	1
Hotspot	HS	128	121	4	1

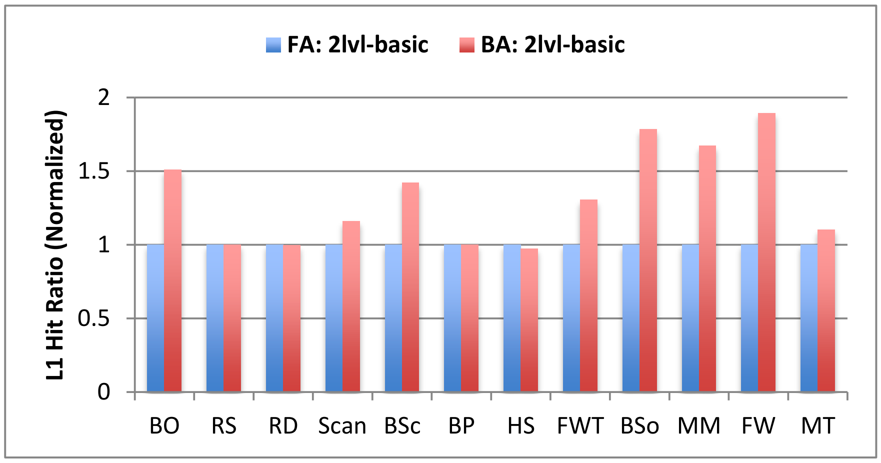
section, we investigate the impact of the proposed block workgroup scheduler on L1 cache hit rate in each compute unit. The block workgroup scheduler is implemented on top of each proposed wavefront scheduler and its impact on cache hit rate is analyzed.

i) Two-level wavefront scheduler + block workgroup scheduler

Figure 4.13 shows the cache hit rate for this configuration. For the saturated case, an average cache hit rate is increased by 32% from baseline configuration. For the unsaturated case, an average of 26.4% increase in cache hit ratio across all benchmarks is measured. The reason for this increased cache hit ratio is due to better locality in the cache. This locality is preserved because the threads are assigned in a contiguous manner to the compute units. We notice a higher hit rate in the saturated case because there are more threads which leads to more memory accesses and more opportunities for finding requested data in the cache memory. We also notice that the *hotspot* kernel had an increase in cache hit rate for the unsaturated case but not the saturated case. We suspect this is due to decrease in locality as the global work size increases. This decrease in locality is most likely due to increase in the strides for the memory access patterns.



(a) Unsaturated

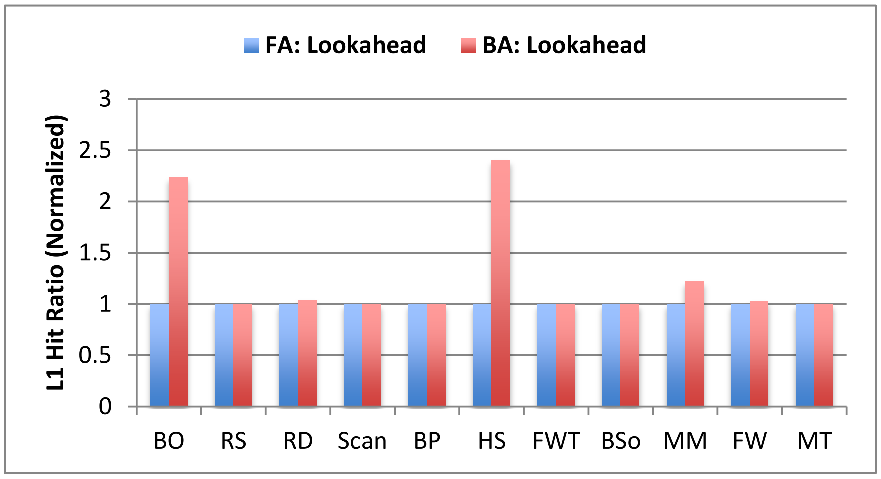


(b) Saturated

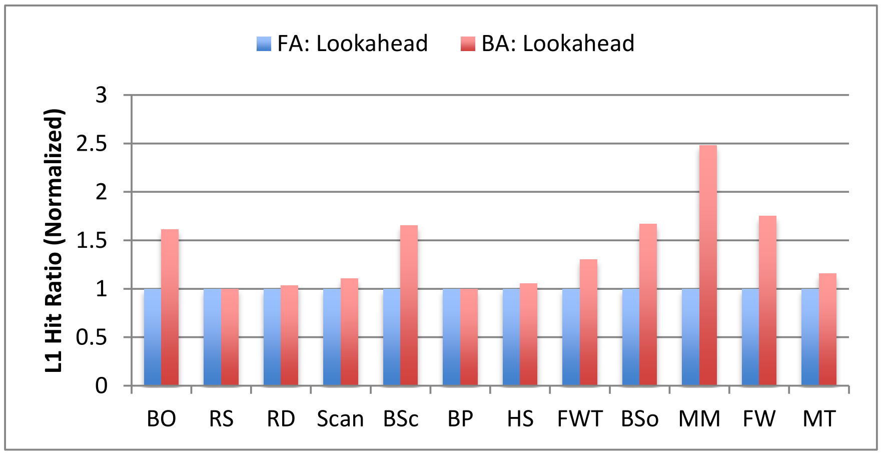
Figure 4.13. Cache hit ratio for a two-level+block workgroup scheduler

ii) Lookahead wavefront scheduler + block workgroup scheduler

Figure 4.14 shows the cache hit rate for this configuration. For the saturated case, we observe 40.3% increase in the hit rate when compared to the base Lookahead scheduler. For the unsaturated case, we observe an average of 26.6% increase in cache hit rate.



(a) Unsaturated

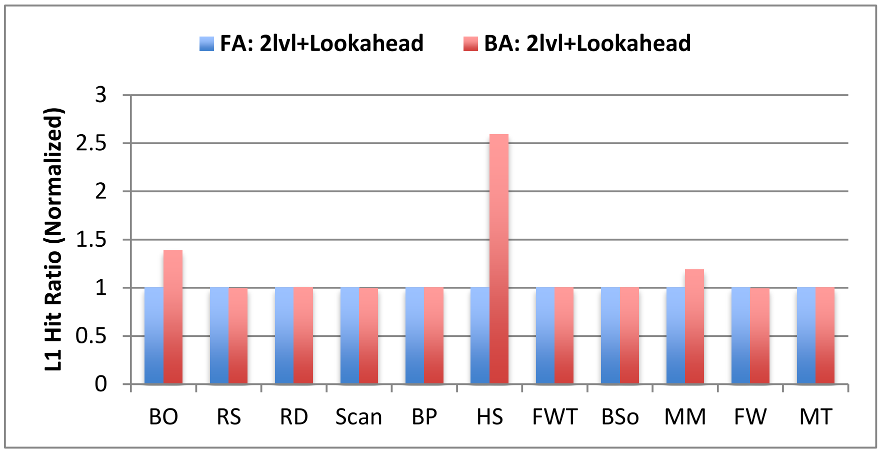


(b) Saturated

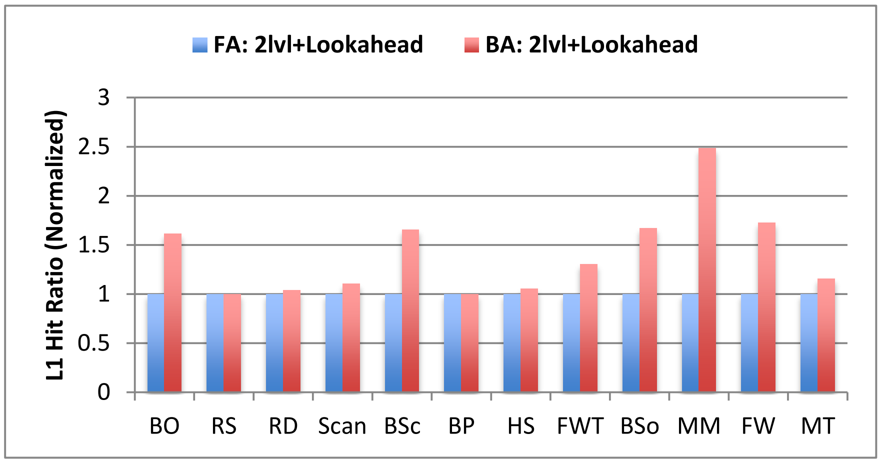
Figure 4.14. Cache hit ratio for a lookahead+block workgroup scheduler

iii) Lookahead + Two-level wavefront scheduler + block workgroup scheduler

Figure 4.15 shows the hit rates for this configuration. For the saturated case, we observe 40.1% increase in the hit rate when compared to the base two-level + Lookahead scheduler. For the unsaturated case, we measure an average of 19.6% increase in cache hit rate.



(a) Unsaturated



(b) Saturated

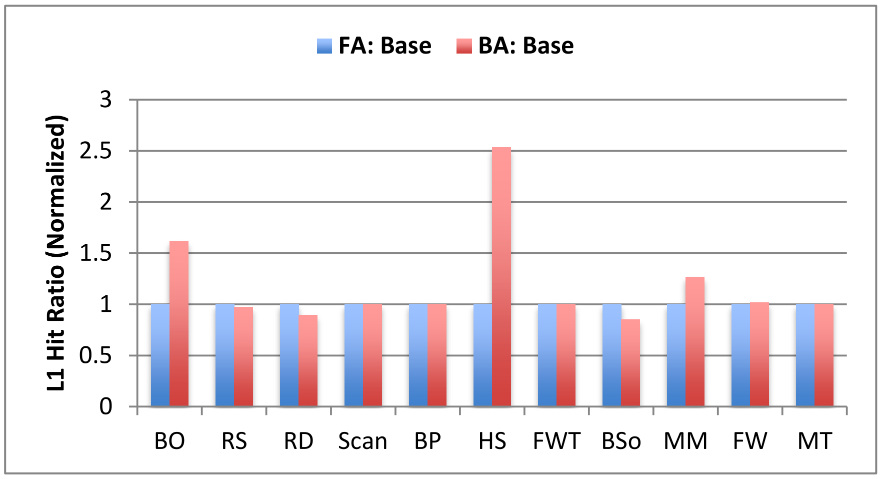
Figure 4.15. Cache hit ratio for a two-level + lookahead wavefront scheduler

iv) Base round-robin wavefront scheduler + Block workgroup scheduler

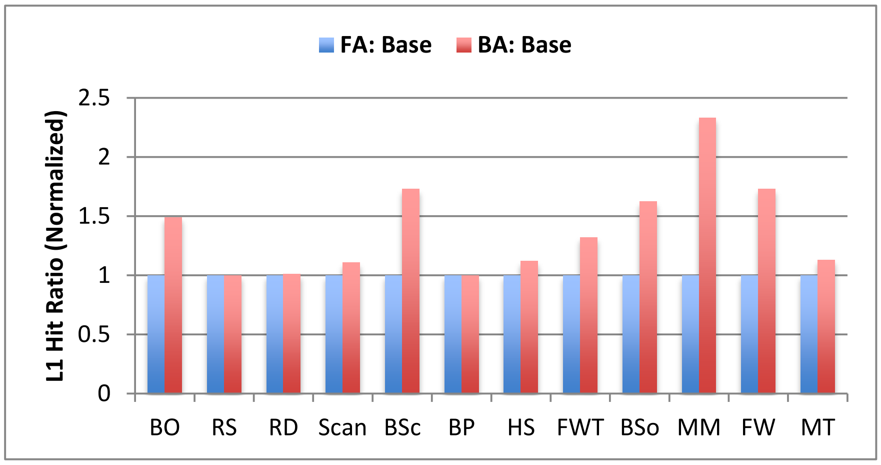
Figure 4.16 shows the cache hit rate for this configuration. For the saturated case, we observe an average of 38% increase in cache hit rate. In the unsaturated case we observe an average of 19.5% increase in the hit ratio across all benchmarks.

4.4.2 Impact on issue unit utilization

Here we show how each wavefront scheduler affects the average idle issue unit cycles in the GPU. We also show how the block workgroup scheduler affects idle issue cycles.



(a) Unsaturated

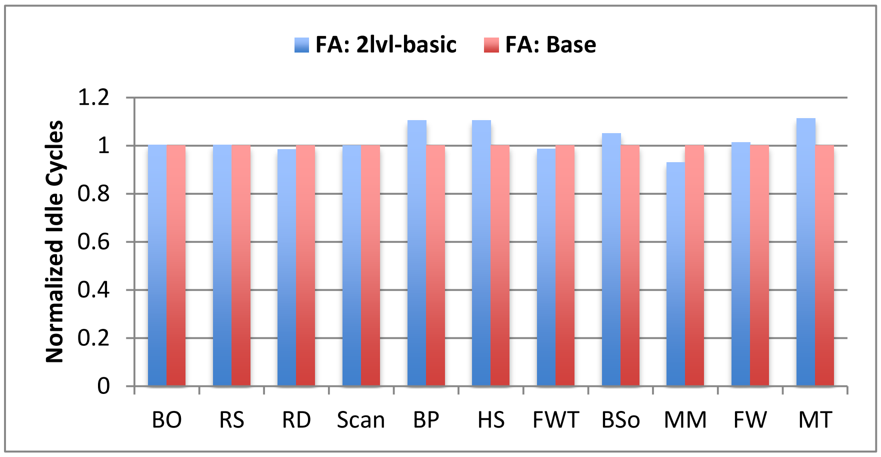


(b) Saturated

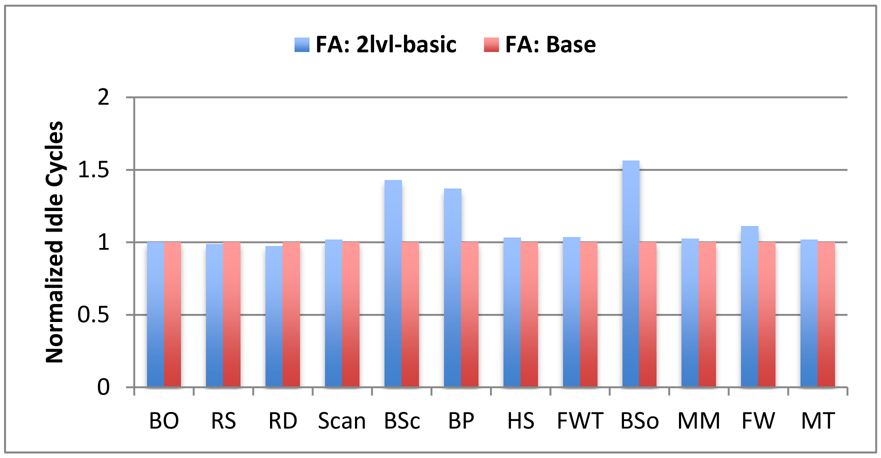
Figure 4.16. Cache hit ratio for a round-robin wavefront scheduler + block workgroup scheduler

i) Two-level scheduler

In this two-level scheduler implementation, we use a fetch group size of 10 (equal to the fetch buffer size). Figure 4.17 shows the normalized idle issue cycles for the profiled benchmarks. In the saturated case, the BlackScholes, BitonicSort, FloydWarshall and BackPropagation benchmarks show increased idle cycles in the issue unit. The other benchmarks show little or no change from the baseline round robin scheduler. For the unsaturated case, we notice a 7% reduction in the idle issue cycles for the *MatrixMultiplication* benchmark. The Back-Propagation, Hotspot, BitonicSort and MatrixTranspose benchmarks show increase in idle issue cycles. The rest of the benchmarks show little or no change from the baseline scheduler. This increase in issue cycles is because this scheduler was able to find only a small number of instructions per cycle for scheduling compared to the baseline round-robin scheduler and as a result, unable to take much advantage of the instruction level parallelism (ILP) in the CU.



(a) Unsaturated

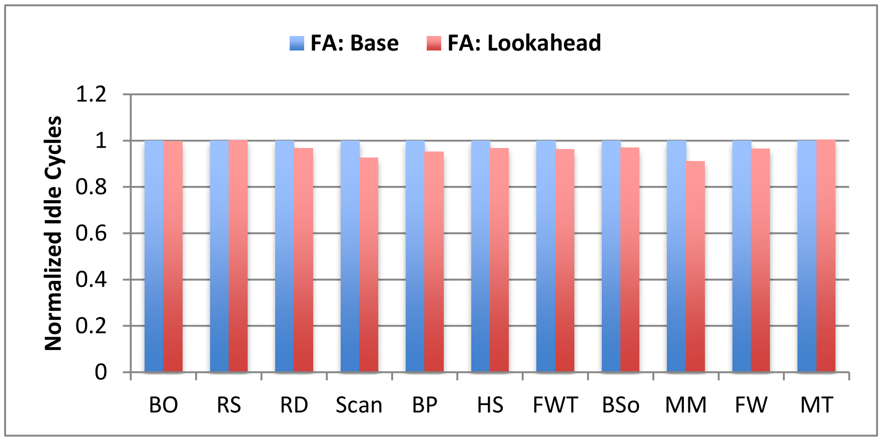


(b) Saturated

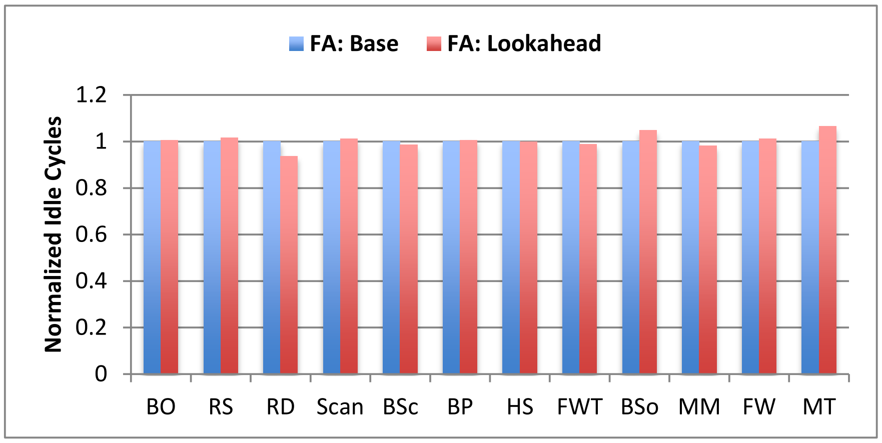
Figure 4.17. Idle cycles for a two-level wavefront scheduler

ii) Lookahead scheduler

Fig 4.18 shows the idle issue cycles measured for the Lookahead scheduler. In the saturated case, compared to the baseline round-robin wavefront scheduler, we noticed a reduction in idle issue cycles from 1% to 6.4% for half of the benchmarks. For the other half, we noticed an increase in idle cycles ranging from 1% to 6%. On average, there was no change in the idle issue cycles compared to the baseline wavefront scheduler for the saturated case. For benchmarks with increased idle cycles, our experiments showed that this was due to the reduction in cache hit rate. For the unsaturated case, we measure a best case reduction of up to 10% in idle cycles for all the benchmarks. This is because the lookahead scheduler reduces the penalty of having an empty fetch buffer by also searching occupied fetch buffers for ready wavefronts in the same clock cycle and as such, better exploits the available ILP present in the compute unit.



(a) Unsaturated

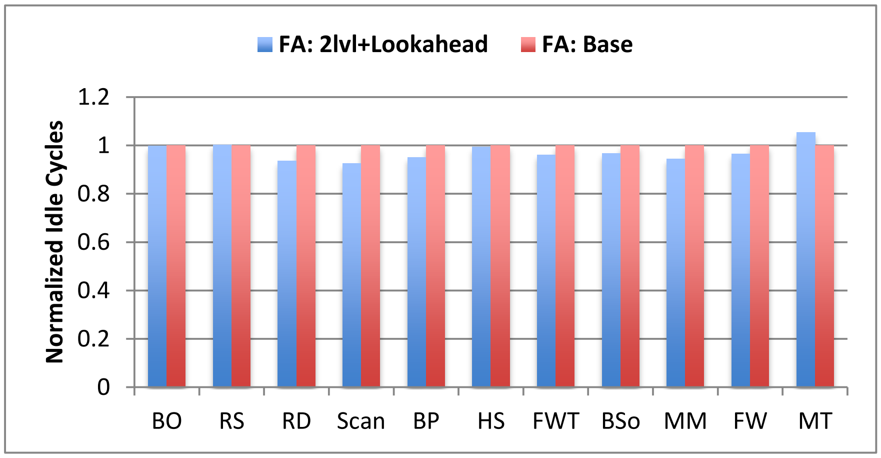


(b) Saturated

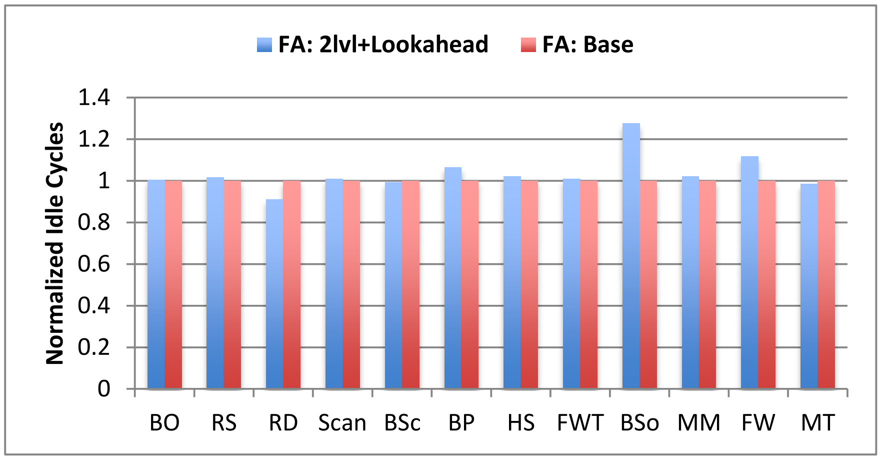
Figure 4.18. Idle issue cycles for a lookahead wavefront scheduler

iii) Lookahead + Two-level scheduler

Fig 4.19 shows the performance of the Lookahead + Two-Level scheduler compared to the baseline. For the saturated case, the average idle issue cycles was equal to that of the baseline scheduler for the profiled benchmarks. For the unsaturated case, we measure a best case reduced idle issue unit cycles of up to 8% for all except one benchmark. For the MatrixTranspose benchmark, the number of idle issue cycles is increased by 5%.



(a) Unsaturated

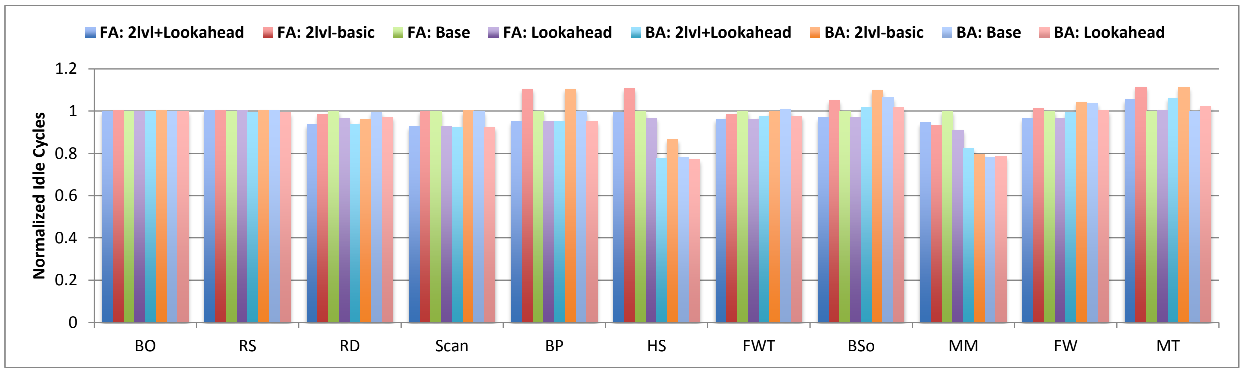


(b) Saturated

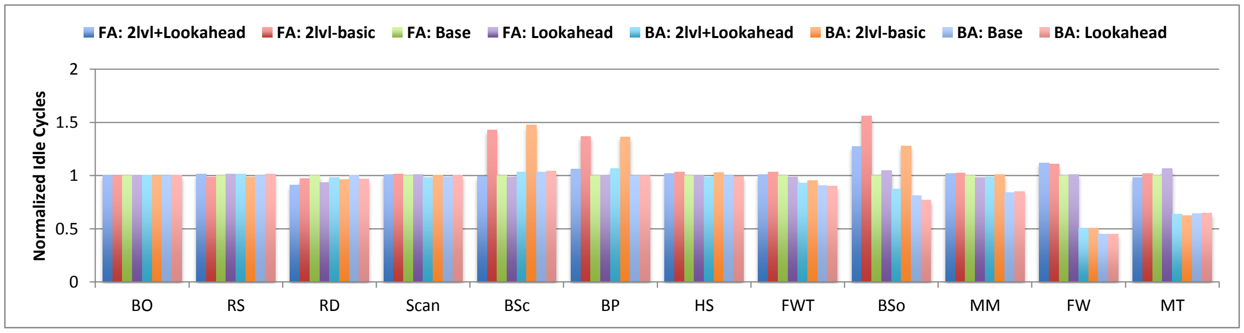
Figure 4.19. Idle cycles for a two-level + lookahead wavefront scheduler

iv) Block workgroup scheduler

Figure 4.20 shows the idle issue unit cycles measured when the block workgroup scheduler is combined with each wavefront scheduler. We compare the results to the baseline first-available workgroup scheduler. For the saturated case, we see a best case reduction of 36% in idle issue cycles. For the unsaturated case, we see a best case reduction of 6% in idle cycles. This performance delta between the saturated and unsaturated cases is due to the massive number of threads found in the saturated case. This massive number of threads gives more opportunity for exploitation of the *preserved locality*[3] in the cache due to block workgroup allocation.



(a) Unsaturated



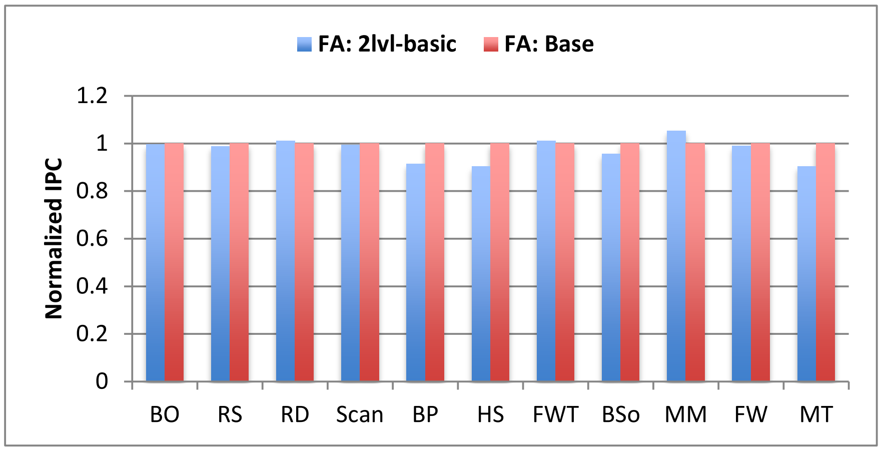
(b) Saturated

Figure 4.20. Idle cycles for a block workgroup scheduler

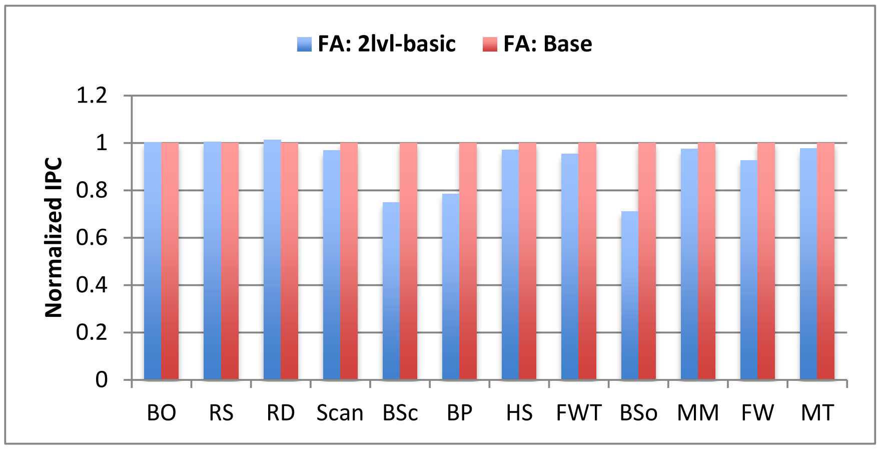
[3]This preserved locality was discussed in section 4.4.1

4.4.3 Impact on IPC

In this section, we present the impact of the proposed schedulers on the overall Instructions Per Cycle (IPC). We use the normalized IPC as a performance metric in the following charts. We discuss the performance of the Two-level, Lookahead, Two-level + Lookahead wavefront schedulers and the block workgroup scheduler.



(a) Unsaturated



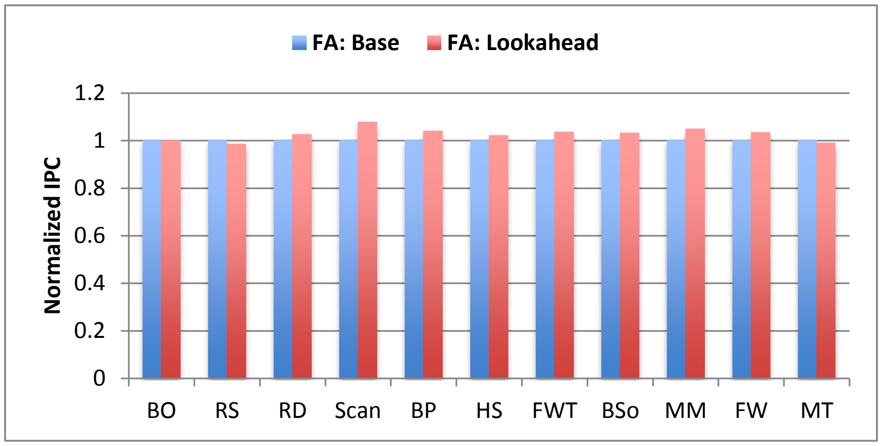
(b) Saturated

Figure 4.21. Measured IPC for a two-level wavefront scheduler

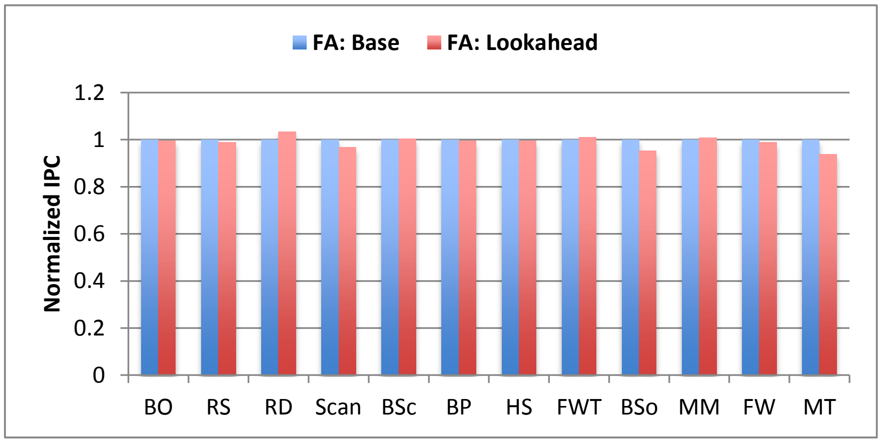
i) Two-level scheduler

Figure 4.21 shows the two-level scheduler's impact on IPC. For the saturated case, we notice a performance decrease of 8%. For the unsaturated case, we measure a performance decrease of

3.6%. Using this scheduler, most benchmarks performed the same or worse than the baseline round-robin wavefront scheduler. Again, this is due to the two-level wavefront scheduler's poor exploitation of the instruction-level parallelism of the GCN compute unit.



(a) Unsaturated

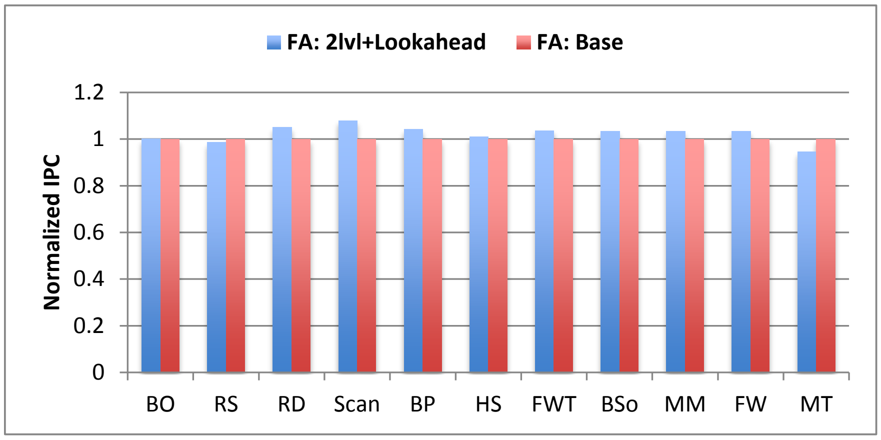


(b) Saturated

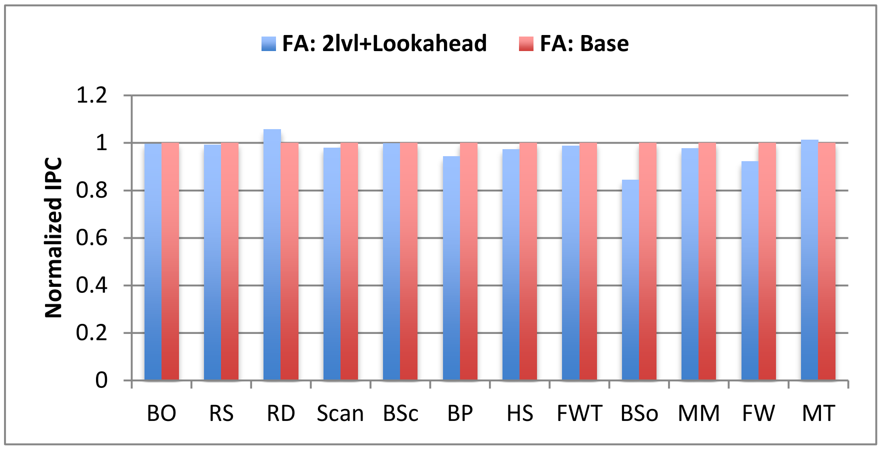
Figure 4.22. Measured IPC for a Lookahead wavefront scheduler

ii) Lookahead scheduler

For the unsaturated case, we get a best case increase in IPC of 8% on all except two benchmarks. For the saturated case, we measure the same performance on average across all the benchmarks when compared to the baseline round-robin scheduler. The reason for the improved performance of this scheduler in the unsaturated case is due to the mitigation of the penalty incurred by empty fetch buffers in the CU. Figure 4.22 shows the performance of this scheduler.



(a) Unsaturated

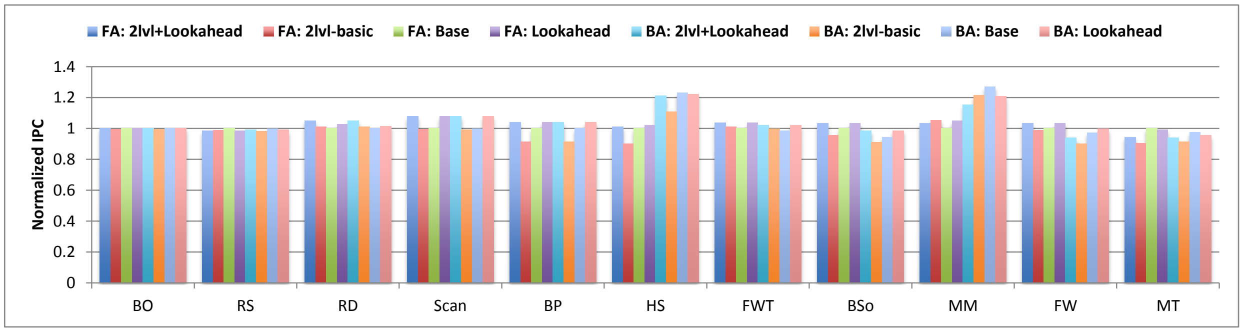


(b) Saturated

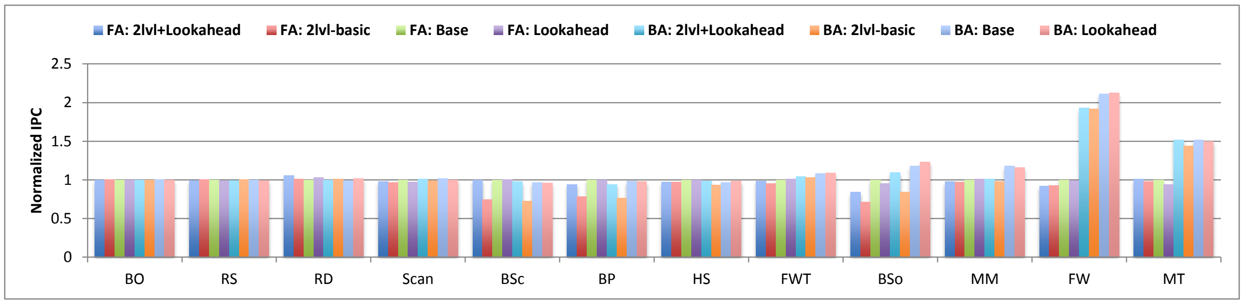
Figure 4.23. Performance for a Lookahead + two-level wavefront scheduler

iii) Lookahead + Two-level scheduler

Figure 4.23 shows the performance of this scheduler. For unsaturated case, we measure a best case increase of 8%. For saturated case, we measure a performance decrease of 2%. Our tests show that this degraded performance was due to the poor performance of the two-level portion of this configuration.



(a) Unsaturated



(b) Saturated

Figure 4.24. Performance for a block workgroup scheduler

iv) Block workgroup scheduler

Here we present the impact on IPC obtained by combining the block workgroup scheduler with each wavefront scheduler. For the saturated case, we measure an average performance increase of 16.5% across all the wavefront scheduler implementations. We measure an average of 16.7%, 18.4%, 14.5% and 16.3% for the Base, Lookahead, Two-level and Two-level+Lookahead schedulers respectively.

For the unsaturated case, we measure an average performance increase of 2.3% across all the wavefront schedulers. We measure an average of 3.5%, 2%, 2.2% and 1.5% for the Base, Lookahead, Two-level and Two-level+Lookahead schedulers respectively. Figure 4.24 shows the performance of a block workgroup scheduler.

From our experiments, we notice that the performance of the block workgroup scheduler is impacted by the cache hit rate and the dynamic instruction mix in the kernel. Though the BinomialOption, Scan, Reduction and BlackScholes kernels have much improved cache hit rate by using block workgroup allocation, we see that the overall performance is not impacted. This is because these kernels have a very small number of vector memory instructions and as a result, the improved cache hit rate has little to no impact on overall performance. The Backprop and radixsort kernels have a significant amount of vector memory instructions, but overall performance is not impacted due to no increase in cache hit ratio. Tracing the RadixSort kernel, we see that this cache behavior is as a result of the long strided memory accesses in the kernel. These long strides reduce the amount of locality present in the program and as a result assigning contiguous workgroups to the same CU does not improve the cache performance. The Hotspot kernel has a small amount of vector memory instructions and no improved cache hit rate (in the saturated case). As expected, overall performance is not impacted. The FastWalshTransform, BitonicSort, MatrixMultiplication, FloydWarshall and MatrixTranspose all see improved performance using this workgroup scheduler. This is because their kernels have a significant number of vector memory instructions combined with an improved cache hit rate due to regular memory access patterns.

CHAPTER 5

CONCLUSION

In this thesis, we aimed at understanding the performance impact of thread (i.e., wavefront and workgroup) scheduling on modern GPUs. To this end, we implemented different thread schedulers at wavefront and workgroup granularities. The proposed schedulers at wavefront level are Two-level, Lookahead and Two-level + Lookahead schedulers. A Block Workgroup scheduler that schedules threads at a workgroup granularity was also implemented. The performance of these schedulers were compared to that of the default round-robin wavefront and first-available workgroup schedulers. Through extensive experiments and evaluation, we observed the followings.

- Checking all fetch buffers for instruction issue is helpful in hardware utilization for unsaturated cases (i.e., small input).
- Assigning contiguous workgroups to the same CU has a huge impact on cache hit rate as well as overall performance.
- The impact of block workgroup allocation is most beneficial in benchmarks with regular memory access patterns
- The impact of block workgroup allocation is most beneficial in memory intensive benchmarks.

- Using a two-level wavefront scheduler degrades performance in an architecture like GCN. We anticipate it would work better with architectures that fetch instructions from a single pool of wavefronts.

Our experiments and evaluations demonstrate that on massively multithreaded GPUs, hardware thread scheduling plays a very important role in performance, and care needs to be taken to choose the right one for the target GPU architecture.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Quickstarts to opencl. http://coolworkspace.wordpress.com/author/coolworkspace/, 2014. Accessed: 2014-04.

[2] Advanced Micro Devices. *AMD Graphics Core Next (GCN) Architecture*, 2012.

[3] Advanced Micro Devices. *Reference Guide: Southern Islands Instruction Set Architecture*, 1.1 edition, December 2012.

[4] Alaa R Alameldeen and David A Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006.

[5] AMD. Amd sdk sample browser. http://developer.amd.com/app-sdk/codelisting.php?q=Accelerated%20Parallel%20Processing, 2014. Accessed: 2014-04.

[6] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[7] Texas Advanced Computing Center. 8 things you should know about gpgpu technology. https://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf, 2014. Accessed: 2014-04.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE Computer Society, 2009.

[9] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Dec 2010.

[10] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. Guided region-based gpu scheduling: Utilizing multi-thread parallelism to hide memory latency. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 441–451, May 2013.

[11] Mark Gebhart, Daniel R Johnson, David Tarjan, Stephen W Keckler, William J Dally, Erik Lindholm, and Kevin Skadron. Energy-efficient mechanisms for managing thread

context in throughput processors. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 235–246. ACM, 2011.

[12] Mark Harris. Parallel prefix sum (scan) with cuda, 2007.

[13] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2006.

[14] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 332–343. ACM, 2013.

[15] Nagesh B Lakshminarayana and Hyesoon Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[16] Kenneth Lee, Heshan Lin, and Wu-chun Feng. Performance characterization of data-intensive kernels on amd fusion architectures. *Computer Science - Research and Development*, 28(2-3):175–184, 2013.

[17] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. IEEE, 2014.

[18] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, 2003.

[19] Peter J Lu, Hidekazu Oki, Catherine A Frey, Gregory E Chamitoff, Leroy Chiao, Edward M Fincke, C Michael Foale, Sandra H Magnus, William S McArthur Jr, Daniel M Tani, et al. Orders-of-magnitude performance increases in gpu-accelerated correlation of images from the international space station. *Journal of Real-Time Image Processing*, 5(3):179–193, 2010.

[20] Ben Lund and Justin W Smith. A multi-stage cuda kernel for floyd-warshall. *arXiv preprint arXiv:1001.4108*, 2010.

[21] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317. ACM, 2011.

[22] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[23] NVIDIA. What is gpu computing? `http://www.nvidia.com/object/what-is-gpu-computing.html`, 2014. Accessed: 2014-02.

[24] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[25] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. In *Parallel Processing and Applied Mathematics*, pages 403–410. Springer, 2010.

[26] Rodinia. Rodinia:accelerating compute-intensive applications with accelerators. `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page`, 2014. Accessed: 2014-04.

[27] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[28] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in cuda. *Nvidia CUDA SDK Application Note*, 2009.

[29] Matthew Scarpino. *OpenCL in Action: how to accelerate graphics and computation*. Manning, 2012.

[30] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-m Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[31] CK Tang. Cache system design in the tightly coupled multiprocessor system. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 749–753. ACM, 1976.

[32] Tao Tang, Xuejun Yang, and Yisong Lin. Cache miss analysis for gpu programs based on stack distance profile. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 623–634. IEEE, 2011.

[33] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[34] Jacobus Antoon van Meel, Axel Arnold, Daan Frenkel, SF Portegies Zwart, and Robert G Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

VITA

Orevaoghene was born in Nigeria. After graduation from highschool, he pursued udergraduate study in Electrical Engineering at the University of Mississippi. Oreva accepted a graduate teaching assistantship to pursue a masters degree in Computer Science at the University of Mississippi. During his graduate studies, he interned twice with Intel Corporation and will work full-time with Intel upon graduation.