

2012

A Hardware-Software Integrated Solution for Improved Single-Instruction Multi-Thread Processor Efficiency

Michael Steffen
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/etd>

 Part of the [Computer Engineering Commons](#)

Recommended Citation

Steffen, Michael, "A Hardware-Software Integrated Solution for Improved Single-Instruction Multi-Thread Processor Efficiency" (2012). *Graduate Theses and Dissertations*. 12639.
<https://lib.dr.iastate.edu/etd/12639>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

**A hardware-software integrated solution for improved single-instruction
multi-thread processor efficiency**

by

Michael Anthony Steffen

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Joseph A. Zambreno, Major Professor

Srinivas Aluru

Morris Chang

Akhilesh Tyagi

Zhao Zhang

Iowa State University

Ames, Iowa

2012

Copyright © Michael Anthony Steffen, 2012. All rights reserved.

DEDICATION

This thesis is dedicated to my parents
for their continuous encouragement and support in my education

and

also dedicated to my wife
for her sacrifice and help that allowed me to complete this work

TABLE OF CONTENTS

LIST OF TABLES	v
LIST OF FIGURES	vi
ACKNOWLEDGEMENTS	xii
ABSTRACT	xiii
CHAPTER 1. Introduction	1
CHAPTER 2. Streaming Processor Architecture	5
2.1 SIMT Processor Architecture	5
2.2 Thread Control Flow Divergence	7
CHAPTER 3. Literature Review	10
3.1 GPGPU-Computing	10
3.2 Warp Control Flow	11
CHAPTER 4. Hyper-threaded SIMT Cores	15
4.1 Warp Divergence	16
4.1.1 Parallel Execution of Diverged Warps	18
4.2 SIMT Hyper-Threading Architecture	19
4.2.1 Stack Manipulation Algorithms	21
CHAPTER 5. Dynamic μ-Kernel Architecture	26
5.1 Memory Organization	27
5.1.1 Thread Usage of Spawn Memory	28
5.1.2 Partial Warp Formation	29
5.2 Spawn Instruction	29

5.3	Warp Formation	29
5.4	Scheduling	31
5.5	Programming Model for Dynamic μ -Kernels	33
CHAPTER 6. CUDA Supported μ-Kernel		36
6.1	CUDA Thread Hierarchy	36
6.2	Improved Thread Spawning	37
6.2.1	Run-time Performance Model Execution	37
6.2.2	Divergence Instruction Hardware	40
6.3	Performance Model	41
CHAPTER 7. Experimental Setup		44
7.1	SIMT Hyper-threading	44
7.2	μ -kernels Setup	44
7.2.1	Benchmark Kernels	46
7.2.2	Benchmark Scenes	47
7.3	CUDA Supported μ -Kernels	48
CHAPTER 8. Experimental Results		50
8.1	SIMT Hyper-threading	50
8.2	Dynamic Micro-Kernel Results	52
8.3	CUDA Supported Micro-kernels	57
CHAPTER 9. Future Work		62
9.0.1	Spawn Memory Caching	62
9.0.2	Register and Shared Memory Utilization	62
9.1	Programmable Thread Grouping	63
9.2	Programming Model for Spawning Threads	64
CHAPTER 10. Conclusions		65
APPENDIX A. Hardware Accelerated Ray Tracing Data Structure		67
APPENDIX B. Teaching Graphics Processing and Architecture		91

BIBLIOGRAPHY 110

LIST OF TABLES

7.1	Hardware Configuration of the Simulator	45
7.2	Simulated Benchmarks.	45
7.3	Configuration used for simulation.	46
7.4	Kernel processor resource requirements per thread.	47
7.5	Benchmark scenes with object count and tree data structure parameters.	48
7.6	Hardware Configuration of the Simulator	48
8.1	Memory bandwidth requirements for drawing a single image without caching. Values are calculated from the number of tree traversal operations and intersection tests.	58
A.1	GrUG performance parameters for each fixed hardware pipeline stage.	80
A.2	Benchmark scenes with triangle count and tree data structure parameters.	85

LIST OF FIGURES

2.1	Single streaming multiprocessor architecture composed of multiple stream processors.	6
2.2	PDOM branching efficiency for a single warp performing a looping operation.	7
2.3	Divergence breakdown for warps using traditional SIMT branching methods for the <code>conference</code> benchmark. Higher values in the key represent more threads that are active within a warp. This figure was created using AerialVision [5].	9
3.1	NVIDIA's GPU Theoretical GFLOP performance compared to Intel CPU's. This figure is from NVIDIA's CUDA C Programming Guide [50].	11
4.1	Single-Instruction Multiple-Thread Architecture for processing a warp. a) No thread Divergence. b) Thread divergence. c) Our 2 way hyper-threading architecture.	16
4.2	Different categories of warp divergence. (a) results in two different control flow paths before all threads reconverge at 'D'. (b) Only a single diverging control flow is required before threads reconverge at 'G'. (c) A looping operation where one iteration has 3 threads active and a later iteration has 1 thread. Similar to (b), only one diverging control flow is created before reconvergence at 'J'.	17

4.3	Example application containing multiple diverging loops where threads run a varying number of iterations per loops. In PDOM, the second loop can only be run after all threads finish the first loop. Using Hyper-threading, threads that finish the first loop can begin executing the second loop before all threads finish the first loop.	18
4.4	Warp meta-data fields for both PDOM and hyper-threading warps. Conventional SIMT warps only requiring knowing a single instruction PC and what threads out of the warp requiring executing this PC. For hyper-threaded warps, multiple instruction PCs are required in addition to expanding the active thread mask to indicate which threads are active and which instruction PC is required.	20
4.5	SIMT 2 way hyper-threading architecture resulting in two virtual processors.	21
4.6	Sample stack modification for allowing threads at a PDOM point to work ahead. The PDOM stack entry (depth 0) is split into three new stack entries: 1) For the next PDOM to reconverge all threads back together. 2) Stack entry for threads working ahead. 3) Stack entry to progress threads currently not at PDOM point to progress to the next PDOM point.	23
4.7	Warp stack layout used for spatial subdivision stack management algorithm. This layout is designed to mimic smaller warps using hyper-threading.	25
5.1	Dynamic thread creation hardware overview. New threads created by the SPs are placed into new warps waiting in the partial warp pool. Once enough threads have been created to complete a warp, the warp can replace an existing warp that has finished.	27

5.2	Architecture for the spawn instruction. Dynamically created threads identify existing threads that will follow the same control path using a look-up table. Once the warp has been identified for the new threads, the memory pointers are stored in memory for later use.	30
5.3	Spawn memory layout for threads accessing parent thread data using 4 threads per-warp and 8 bytes of storage between threads. Child threads use their special register to access the warp formation data. The warp formation data is a memory pointer to the parent thread data.	35
6.1	Potential processor efficiency lose from using Micro-Kernels. The nested branching operation at C uses the spawn branching method. Thread reconverging at block F see less efficiency then using PDOM branching at C since their are no threads to reconverge with.	38
6.2	Implementation for the diverge instruction and updating parameters used by the performance model.	41
8.1	Limitations on the performance gain using hyper-threading on warps.	51
8.2	2-way hyper-threading results.	52
8.3	Effects on performance with increasing hyper-threading virtual processor count.	53
8.4	Comparing performance results of using hyper-threading and smaller warp sizes.	54
8.5	Hyper-threading results using existing SIMT memory systems	55
8.6	Divergence breakdown for warps using μ -kernels for the conference benchmark. Warps are able to keep more threads active by creating new warps at critical branching points.	56
8.7	Performance results for all benchmarks using different branching and scheduling methods.	57

8.8	Divergence breakdown for warps using dynamic thread creation with bank conflicts for the conference benchmark. Warps still maintain more active threads over traditional branching methods, however, additional pipeline stalls are introduced by bank conflicts.	59
8.9	Branching performance for the conference benchmark. Theoretical results were simulated with an ideal memory system.	59
8.10	Performance for all warps diverging with different percentage of threads going to the two different branches. As we have more threads executing the shorter of the path, we expect higher performance since our method will perform fewer instructions.	60
8.11	Performance for 50% of threads diverging in a warp with different percentage of warps diverging in a block. As we go to lower number of warps diverging we have fewer threads to form new warps, PDOM performance starts to be the better branching method.	60
8.12	Performance results for 50% of threads diverging in a warp with different number of instructions after reconvergence point. As the number of instructions after reconvergence increases above three times the branching instructions, PDOM becomes more efficient when <i>reconverge</i> instructions are not used.	61
A.1	GrUG data structure composed of a uniform grid for the top layer and a lower layer that maps to a HS3 tree structure.	71
A.2	Hash function starting with X,Y,Z coordinates in integer format and producing the data structure that contains geometry information for intersection testing.	73
A.3	Variables needed for DDA traversal.	75
A.4	Architecture of Grouped Uniform Grid	77
A.5	Integration of the GrUG pipeline into a multi-core graphics processor and the fixed hardware stages for the GrUG pipeline.	79
A.6	Architecture of GrUG hash function for one axis using a 512 grid.	81

A.7	Benchmark data for each scene and the total memory required by the data structure.	82
A.8	The memory requirements of GrUG compared to <i>kd</i> -tree.	83
A.9	Number of memory reads required for traversing GrUG and <i>kd</i> -tree.	84
A.10	Total number of instructions executed for traversal of visible rays. <i>kd</i> -tree results are compared against GrUG using grid sizes of 512, 256 and 128.	86
A.11	Number of completed rays for the first 300,000 clock cycles of the Conference benchmark scene.	87
A.12	Memory requirements for <i>kd</i> -tree and GrUG.	88
A.13	Ratio of the performance increase divided by the memory overhead for different benchmarks and grid sizes. Performance increase and memory overhead are based on the <i>kd</i> -tree algorithm resulting in a value of 1.0 for the <i>kd</i> -tree.	89
A.14	Memory bandwidth per frame required to render visible rays without caching.	90
B.1	FPGA architecture framework provided to students. Students were responsible for implementing code in the graphics pipe sub-unit.	94
B.2	The protocol for sending data from the workstation to the FPGA. This format represents the first 32-bit message in a packet.	96
B.3	Format of how vertex attributes are sent to the FPGA and then stored in memory. The index queue stores address for the vertex and color queue for each vertex.	99
B.4	Sample student OpenGL application that draws a 2D video game using pixels.	100
B.5	Transformation process for converting 3D OpenGL coordinates to screen coordinates.	101
B.6	A rasterization zig-zag pattern implemented by students to identify all pixels inside of a triangle.	103

- B.7 OpenGL application used to test all 8 depth functions. Each horizontal bar uses a different depth function test. The black color is a plane at depth 0 and the red and blue colors are a sin wave with depths from -1 to 1. 106
- B.8 Texture coordinates and how they map to the texture image memory address. 107

ACKNOWLEDGEMENTS

First, thanks to my advisor, Professor Joseph Zambreno for his guidance in my research and professional development as well as for additional opportunities to pursue my interests in engineering education. Thanks also to Professor Jeffrey Will at Valparaiso University for his encouragement and mentorship when my interest in computer engineering first sparked. I also thank Professor Srinivas Aluru, Professor Morris Chang, Professor Akhilesh Tyagi and Professor Zhao Zhang for serving on my Preliminary and Final Examination committee.

This work was partly funded by the National Science Foundation (NSF) Graduate Research Fellowship (GRF).

ABSTRACT

This thesis proposes using an integrated hardware-software solution for improving Single-Instruction Multiple-Thread branching efficiency. Unlike current SIMT hardware branching architectures, this hardware-software solution allows programmers the ability to fine tune branching behavior for their application or allow the compiler to implement a generic software solution. To support a wide range of SIMT applications with different control flow properties, three branching methods are implemented in hardware with configurable software instructions. The three branching methods are the contemporary Immediate Post-Dominator Re-convergence that is currently implemented in SIMT processors, a proposed Hyper-threaded SIMT processor for maintaining statically allocated thread warps and a proposed Dynamic μ -Kernels that modified thread warps during run-time execution. Each of the implemented branching methods have their strengths and weaknesses and result in different performance improvements depending on the application. SIMT hyper-threading turns a single SIMT processor core into multiple virtual processors. These virtual processors run divergent control flow paths in parallel with threads from the same warp. Controlling how the virtual processor cores are created is done using a per-warp stack that is managed through software instructions. Dynamic μ -Kernels create new threads at run-time to execute divergent control flow paths instead of using branching instructions. A spawn instruction is used to create threads at run-time and once created are placed into new warps with similar threads following the same control flow path.

This thesis's integrated hardware-software branching architectures are evaluated using multiple realistic benchmarks with varying control flow divergence. Synthetic benchmarks are also used for evaluation and are designed to test specific branching conditions and isolate common branching behaviors. Each of the hardware implemented branching solutions are tested in isolation using different software algorithms. Results show improved performance for divergent applications and using different software algorithms will affect performance.

CHAPTER 1. Introduction

Single-Instruction Multiple-Thread (SIMT) processor architectures are used for large data-parallel applications since they are designed to be scalable to very high processor core counts. Thread level parallelism (TLP) is also commonly associated with SIMT to support even larger quantities of parallel threads to help hide individual thread latencies. As an example, NVIDIA's GeForce GTX 590 [46] SIMT processor has 1024 thread processor cores and supports TLP of 32-threads per thread-core. To support these high core counts, SIMT architectures impose additional structure on multi-threaded applications and introduce some simplifications to conventional processor datapaths that can negatively impact performance. These limitations on program structure often require new algorithms to be developed for improving SIMT performance; making the porting process for existing algorithms to SIMT difficult [36]. Similarly, while many algorithms have been developed for SIMT processors, (see the over 1,000 applications and papers reported by the NVIDIA CUDA Community Showcase [49]), performance is often less than the expected Amdahl's speedup, mainly for architectural reasons.

To manage the large number of threads (32,768 for the GeForce GTX 590), threads are organized into blocks dictated by the programming model and an entire block is scheduled onto one of the SIMT core processors. When threads are assigned to a SIMT core they are statically grouped into warps. SIMT cores use warps as the scheduling entry instead of individual threads. When a warp is scheduled, all threads inside the warp are executed in their own processing lane but share scheduling and instruction fetching hardware. The GeForce GTX 590 has 32 SIMT core processors each configured to support a warp size of 32 threads.

Individual threads are allowed to execute control flow instructions, even for threads inside the same warp. Control flow divergence can then occur when threads inside the same warp require more than one control path, resulting in different instructions to be executed. Since

only a single instruction can be fetched for all threads in a warp, all required control flow paths are executed and threads not requiring the current path being executed are disabled (decreasing the IPC). Reconvergence algorithms are implemented to minimize the time spent in diverging control flow paths. However, processor utilization still decreases as not all of the processing lanes can be utilized due to disabled threads.

Consider, for example, physically-based global rendering algorithms, that seek to produce highly realistic images through the modeling of the physics of light transport [53]. Conceptually, these global rendering algorithms map nicely to wide SIMT hardware, since individual pixels can be represented by a single thread, resulting in thousands of individual threads with no inter-thread data dependencies. These threads can then be mapped to one of the many parallel cores that can switch between threads at minimal cost [9, 55]. In practice, however, global rendering algorithms require large amounts of memory bandwidth as well as complex scalar thread flow (i.e. branching). While fixed hardware solutions such as Appendix A can improve performance, these implementations are not generic enough for other types of algorithms. Results from SIMT execution on NVIDIA GPUs show that performance is typically limited by the complex control flow and not the memory bandwidth requirements [1]. Global rendering algorithms are just one of many types of algorithms that have difficulty achieving significant performance increases on SIMT architectures and any application requiring control flow instructions will see some degree of performance degradation.

To reduce the performance loss from branching, this thesis proposes using integrated hardware/software branching methods that allow developers to fine tune branching behavior for the application. By including a software component to branching behavior, developers have full control over the instructions being used for each branch in their application or can allow a compiler to implement generic branching solutions. To be able to use hardware/software branching methods, this thesis also introduces two software configurable branching behaviors each designed for improving different workloads of branching applications.

The first proposed software/hardware branching method is hyper-threaded SIMT processors. Hyper-threading [40] creates multiple virtual parallel processors out of a single physical processor core. While conventional hyper-threading uses different execution data paths to

mimic more processors, this approach divides the individual thread processing lanes. This creates multiple SIMT processors of different warp-sizes from a single SIMT core. Using a single warp to schedule parallel threads for hyper-threading, no conflicts will occur between processing lanes when forming SIMT hyper-threading processors. Conventional hyper-threading using the SIMT thread processor's execution pipelines can also be implemented along with this proposed hyper-threading architecture.

To determine how to divide the individual thread lane processors for hyper-threading, the per-warp stack reconvergence table is extended. We present three algorithms for managing the stack entries, as this could be customized for specific applications. The first algorithm presented is used to schedule stack entries without thread conflicts with other stack entries. With this method, only diverging control flow produced from if-else or case statements will utilize the hyper-threading architecture. The second approach spatially divides up all thread lane processors into even groups based on the number of different instructions the core is able to fetch. This method is similar to reducing the warp size while keeping the processor core count consistent. The only difference is that the warp size is held constant and for a warp to finish all spatially divided thread lanes must finish all threads before a warp is concluded. The final approach attempts to maximize the thread lane utilization by picking two instructions in the warp stack that will result in the best performance.

The second proposed software/hardware branching method is to allow threads to dynamically spawn new threads at runtime. The locations in a kernel where a branching statement can lead to a significant code divergence is where new threads are created dynamically during runtime. The parent thread then exits allowing the child thread to continue processing the same work as the parent would of before spawning. Threads created at runtime are then placed into new warps, where all threads in a new warp will begin executing the same PC. SIMT processor efficiency is improved by grouping threads that would of run diverged inside a warp into new thread warps that will have no initial thread divergence.

The remainder of this thesis is organized as follows. Chapter 2 introduces SIMT architecture and describes the current performance loss from thread control flow divergence. Chapter 3 provides additional background information on SIMT processing, with a review on related work

targeting SIMT branching performance. In Chapter 4 our first software/hardware branching method using hyper-threaded SIMT cores is described. Chapter 5 introduces the second branching method of spawning dynamic threads. In Chapter 6 spawning dynamic threads is extended to support NVIDIA CUDA resources such as thread synchronization and other block resources. Chapter 7 outlines our experimental setup with results being discussed in Chapter 8. Chapter 9 outlines possible directions of future work related to a software/hardware branching method and Chapter 10 concludes this thesis. Appendix A contains performance improvements targeted for ray-tracing applications on SIMT processors. Appendix B demonstrates how to teach graphics processing architecture.

CHAPTER 2. Streaming Processor Architecture

SIMT processors are designed for executing large numbers of parallel threads that are defined at the start of execution and run the same application kernel [57]. The overall runtime for all threads (from first thread launch to the last thread finishing) is the critical performance metric, rather than individual thread runtime. With this metric in mind, the architecture is tailored to high-throughput parallel processing with scheduling policies that focus on keeping processor ALUs active. Parallel processing is accomplished by having a large number of lightweight cores that lack branch prediction, out of order scheduling, and other scalar thread acceleration hardware. Individual cores are kept from stalling by constantly switching between all available threads. While switching between threads increases the individual runtime for a thread, it allows for high latency instructions that would stall a processor to be tolerated by executing instructions from other active threads.

2.1 SIMT Processor Architecture

For area and energy efficiency, SIMT processors are grouped together to share certain hardware components, such as register files and instruction fetch units [35]. SIMT architectures commonly define processors using a hierarchy. At the top level, the architecture is composed of an array of processors referred to as Streaming Multiprocessor (SMs) [48]. All SMs have access to multiple memory controllers through a networked interconnect to allow for highly banked parallel memory operations. SMs operate in isolation and communication is not supported amongst SMs. As a result, two threads assigned to different SMs have no synchronization support, whereas threads assigned to the same SM have some limited synchronization capabilities.

The second level of the SIMT processor hierarchy is inside each SM component. SMs

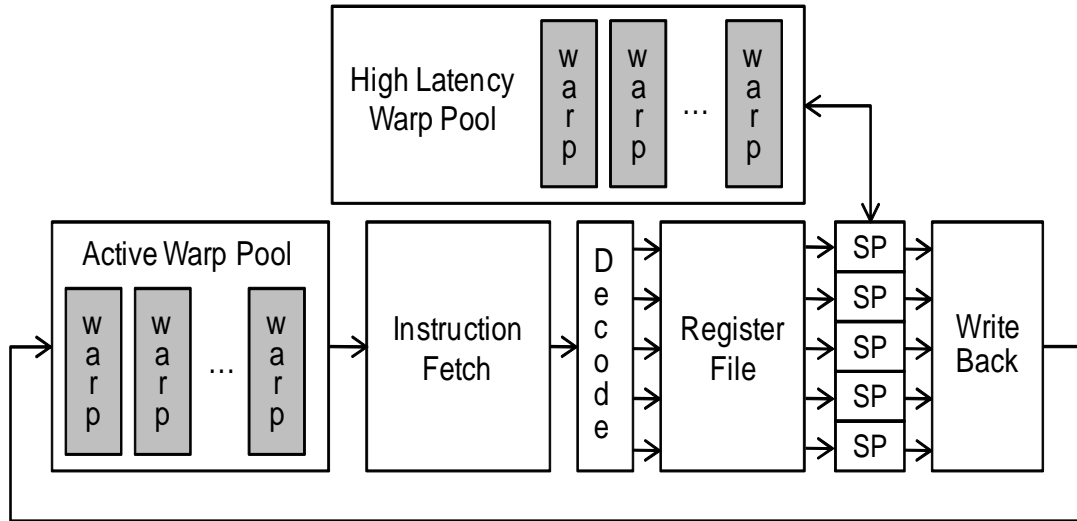


Figure 2.1: Single streaming multiprocessor architecture composed of multiple stream processors.

are composed of multiple Stream Processors (SPs) [48] that execute scalar threads. SPs are primarily ALUs and do not have individual register files or instruction fetch units; instead, SPs receive instructions and data from a single shared instruction fetch unit and register file. The SM register file is highly banked to allow for multiple simultaneous accesses by SPs. Each SM also contains two thread queues to manage the threads assigned to it (see Figure 2.1).

For all SPs inside of an SM to share a instruction fetch unit efficiently, threads running within a processor group must execute the same instruction, and register names are manipulated to access different data on all the SPs. Individual threads are then grouped into warps [48] at application launch and remain together throughout their lifetime. The number of threads in a warp can be a multiple of the number of SPs in a SM. All threads in a warp then execute in lock-step, requiring only one instruction fetch for all threads. Warps are the granularity used for scheduling inside an SM, where the individual threads of a warp are assigned an individual SP.

Unlike conventional scalar thread processing, SPs execute one instruction from a warp and can then switch to another warp on the next cycle. Fast switching is done by using a scheduling thread queue (with each element organized as warps) and a large register file. The register file is large enough to accommodate all threads assigned to an SM. The number of registers that can be used per thread is flexible, but can be the limiting factor in terms of the number of

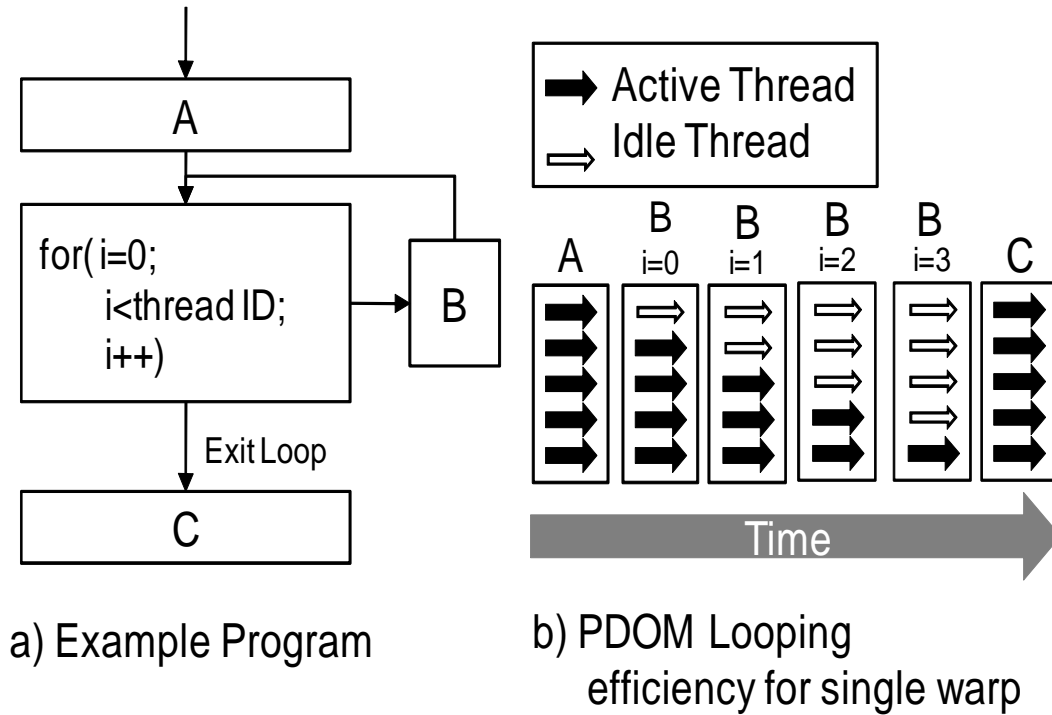


Figure 2.2: PDOM branching efficiency for a single warp performing a looping operation.

threads assigned per SM (the size of the thread queue is also another possible limiting factor). A warp is fetched from the thread queue after execution of the current warp instruction. Once a warp has finished data write-back for the fetched instruction, the warp is placed back in the thread queue. If the current executed instruction requires a high-latency operation before write-back, the warp is placed into another thread queue. Once the instruction finishes, the warp is moved back into the scheduling thread queue.

2.2 Thread Control Flow Divergence

To allow for branching instructions, threads within a warp must be allowed to follow different control flow paths while using a single instruction fetch unit. Consequently, all possible control paths in a warp are executed sequentially and threads not requiring the current control path do not commit the results of those instructions. To minimize the performance loss of SPs sitting idle, reconvergence algorithms, such as Immediate Post-Dominator (PDOM) [28], are implemented to schedule all control paths with minimal processor idle time. Figure 2.2 shows

an example for a simple data dependent looping operation (similar to the one presented by Fung et al. in [23] for instruction branching), and how PDOM results in multiple idle streaming processors. The example application in Figure 2.2a only has two control paths for the loop (running B again or executing C) and a single thread convergence location (C). PDOM first executes the first control path for B until there are no more threads requiring this path (Figure 2.2b). The next control path is executing C, where all threads would be enabled, since C is the convergence location for all the threads in the warp. If the runtime for B is much larger than for both A and C, the looping operation would only be 50% efficient since only half of the SPs would be used on average.

Consequently, applications that require complex control flow or long-running diverging branches can have decreased processor efficiency, due to the idle SPs that are completing all control flow paths before the warp can converge from the diverging paths. Figure 2.3, plotted using AerialVision [5], illustrates how many streaming processors are running idle per clock for executing a highly divergent ray-tracing application. This plot categorizes a warp into 10 different categories based on the number of threads in a warps that are not idle. Category W29:32 is the number of warps that have 29 to 32 active threads in the warp. Category W1:4 indicate that there are only 1 to 4 active threads and the remanding are idle due to branching.

Two types of control flow instructions are implemented in SIMT architectures: predicated instructions and jump instructions. Predicated instructions [12] are only executed if the predication register is set. With predication, no stack modifications are required (unless a predicated jump instruction is executed) since all threads advance to the same next PC whether or not they executed the predicated instruction. This work does not attempt to improve performance for predicated instructions since predicated instructions are less common and larger predicated code blocks could be re-written to use jump instructions. Jump instructions alone are no concern since all threads executing a jump instruction move to the jump target resulting in no thread control flow divergence. Thread divergence in a warp only occur when a predicated jump instruction is executed. In this scenario, each thread can have different predicate register values that can result in some of the threads executing the jump and others skipping the jump instruction due to predication.

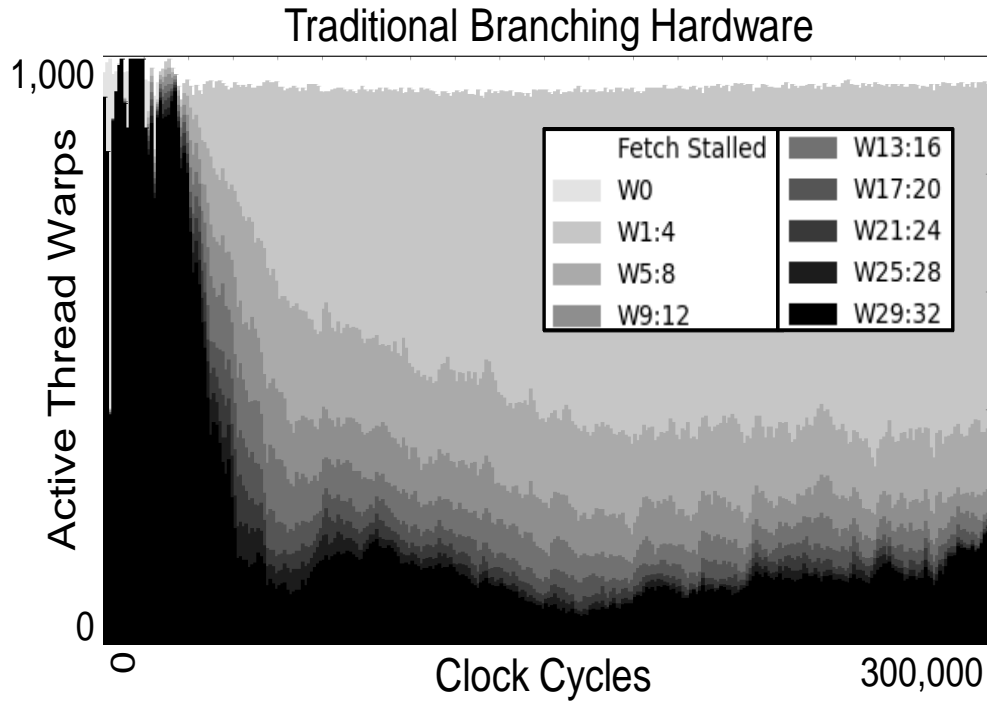


Figure 2.3: Divergence breakdown for warps using traditional SIMT branching methods for the `conference` benchmark. Higher values in the key represent more threads that are active within a warp. This figure was created using AerialVision [5].

Improving the under-utilization of SPs can have a dramatic effect on the performance of an application, without the need of additional processing power. As previously mentioned, in our observation of branching performance, scalar thread branching only decreases performance when threads in a warp follow different control paths. If all threads in a warp follow the same control flow, there is no performance loss. Additionally, two different warps can have diverging control flow with no performance loss, presuming all threads in the warp follow the same path.

CHAPTER 3. Literature Review

3.1 GPGPU-Computing

General-Purpose Computing on Graphics Processing Units (GPGPU) allowed parallel applications to be executed on graphics processors that often had a higher GFLOPS than modern time CPUs (See Figure 3.1). While early 3D graphics processing hardware were implemented entirely in fixed logic (such as the one described in Appendix B), graphics design continue improving performance by leveraging the abundant amount of parallelism in graphics algorithms. As processing technologies continued to shrink, additional parallel processing logic is added to GPUs. To improve the visual quality of rendered images, programmable shaders were eventually added to the rendering pipeline [37]. As the GPU programmable shaders continued to advance, several high level programming languages were created for the intent of being used for rendering computer graphics [21, 33]. As the GFLOP difference between CPUs and GPUs continued to grow (see Figure 3.1) and the GPU pipeline continually added programmable capabilities, developers started porting general-purpose computations to the graphics rendering pipeline [51]. Initially GPGPU developers were required to map their general purpose applications to one of the common graphics rendering APIs, either OpenGL [62] or DirectX. Later, several APIs such as BrookGPU [13, 14] and Lib Sh [31] were developed that allowed GPGPU developers to port their applications to these APIs designed to run on the GPU pipeline but not requiring the graphics APIs. Today GPU vendors support multiple APIs designed for both graphics (OpenGL and DirectX) and GPGPU computing (NVIDIA CUDA [48], ATI Streams [6] and OpenCL [42]).

While GPU computing is increasing in popularity, graphics processing still influences the processing architecture for general purpose computing. One such influence is how code control

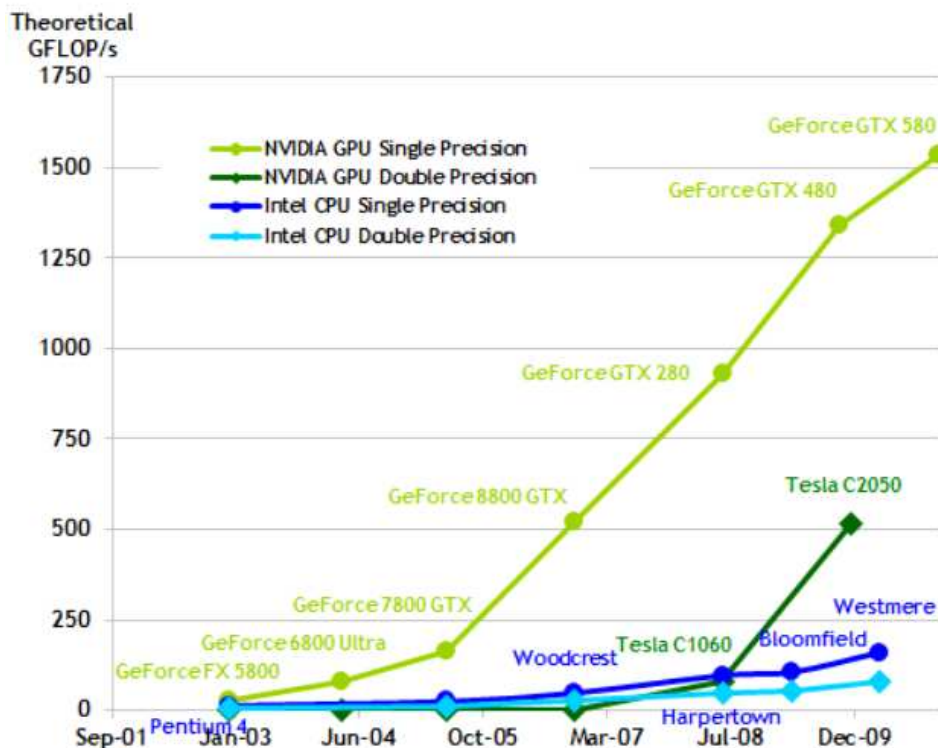


Figure 3.1: NVIDIA's GPU Theoretical GFLOP performance compared to Intel CPU's. This figure is from NVIDIA's CUDA C Programming Guide [50].

flow is processed. Rendering shaders that run on the GPU programmable shaders started off with only a few instructions that did not require much control flow (initial shaders supported none). As rendering shaders grew in complexity, branching support was improved, however today's shaders still contain less complex control flow than GPU computing applications. Consequentially, GPU computing uses the same branching methods as GPU shaders, where the branching performance is not a large concern in graphics shaders.

3.2 Warp Control Flow

Improving wide SIMT processor efficiency for complex control flow has been approached from both hardware and software perspectives. Persistent threads [1] is a software scheduling algorithm specifically for ray tracing applications. This approach uses just enough threads to keep the machine full, and allows warps to focus on a single section of the entire algorithm. The entire algorithm is represented by multiple warps and uses memory arrays called work

queues to pass data between warps. Warps are then executed in a loop operation for reading work from a queue, performing computations, and then writing back to another queue or device memory. Once the work queues are all empty, the threads then exit. Divergent control flow is reduced by allowing threads in a warp to write to different work queues. Since this is entirely a software solution, this algorithm can be applied to current and future hardware that supports the required memory transaction instructions. To prevent concurrency errors during work queue translations from the large number of parallel threads running at a time, atomic instructions are required. Atomic instructions result in higher instruction latencies to serialize the instructions operating on the same data. Scheduling of warps is also left up to the developer, resulting in complex scheduling code for workload balancing, or simple methods that can result in an unbalanced distribution.

Hardware support for SIMT branching has grown in complexity as machines continue to advance to support wider application scopes. Early SIMT machines supported branching using mode bits [12] (also called predicated masks). Mode bits would disable the results for specific threads from being written back, effectively disabling the processor. This method results in every instruction being executed and processors are turned off accordingly. Modern day processors still implement mode bits for short branching instructions; however it does not support diverging control flow.

Dynamic warp formation [23] allows for warps to be modified to contain different threads during runtime. Processor cores have multiple warps assigned at application launch and only a few can be executed through the pipeline at a given time. As warps exit the pipeline back into the warp pool, threads are separated based on their next PC and placed into new warps. Warp metadata is then expanded to keep track of which threads are in a warp, so that register translation methods can still function correctly. Threads can be organized into warps using two methods. The first requires that threads cannot change the SP to which it was originally assigned. This method needs minimal hardware support, but limits thread flexibility. The second method adds a cross-bar network to all SPs to allow register values to be passed. While this adds hardware complexity, threads can be assigned to any warp that has a thread opening. Modifying warps at this level has the advantage of not requiring any code modifi-

cations. Thread Block Compaction [22] further expands Dynamic Warp Formation by only re-organizing threads into warps at diverging branch locations. Thread Block Compaction also uses the PDOM location data to re-organize threads back into the original warps for improved memory operations.

Dynamic Warp Subdivision [41] allows warps that contain stack entries that could be executed in parallel to temporarily separate into two warps. Each warp is composed of a single control flow path that is scheduled independently of the other warps (where with PDOM, each control flow path is scheduled sequentially). Subdivided warps are re-converged at the PDOM locations to maintain optimal processor utilization. Since subdivided warps are executed on the same SM, performance improvements can only result for applications where the scheduler runs out of available warps to schedule (either due to long latency instructions or limited work) and for applications that result in diverging branches that can run in parallel.

Re-convergence at thread frontiers [18] attempts to re-converge divergent threads in a warp before the Immediate Post-Dominator Locations. Unstructured control flow code can often result in divergent threads running the same basic block instructions before their Immediate Post-Dominator point. Re-convergence at thread frontier allows for these threads to be re-converged at these basic blocks, by prioritizing the scheduling of basic blocks and identifying control flow paths that could potentially intersect before the PDOM location. Re-converging threads before the PDOM location reduces the total number of executed instructions and scheduled warps.

Decreasing warp size [36] while maintaining the number of thread lane processors also improves processor utilization for highly divergent applications. This method potentially improves performance for applications consisting of large amounts of thread divergence since the number of threads in a warp is smaller resulting in fewer threads potentially diverging. Applications with small amounts of divergent control flow will see little performance gains and hardware implementation to maintain the same IPC results in additional warp scheduling hardware.

Large Warps and Two-Level Warp Scheduling [43] improves branching performance by increasing the number of threads in a warp such that it is significantly larger than the number of processing SPs. Every time a warp is scheduled, smaller sub-warps are formed based on the

thread's control flow path allowing for threads to be grouped for improved processor utilization.

Programming models for GPUs have focused on graphics rendering [62] and general-purpose computations [48, 6]. Graphics rendering programming models use the concept of a pipeline. While different pipeline stages can be programmable using custom kernels, additional pipeline stages cannot be added and the data movement between stages is fixed. Furthermore, implementation of the fixed logic hardware components is proprietary and fixed. Programmable graphics pipelines is supported in GRAMPS [67], where user-defined pipeline stages (implemented using kernels) pass data between stages using queues. However, underlying optimizations for Wide SIMT control flow are not addressed.

CHAPTER 4. Hyper-threaded SIMT Cores

The first proposed software/hardware branching solution focuses on extending the capabilities of the Immediate Post-Dominator (PDOM) branching solution to allow for SIMT Hyper-threading architecture. Hyper-threading allows for parallel execution of different control paths from a single warp. Application developers can configure how the parallel execution of different control paths are performed through modifying data fields in the warp meta-data. Since this method is an expansion to PDOM, static allocation of threads into warps is maintained, making it a good target for diverging branches that are not long or are deeply nested in the control flow graph.

Hyper-threading [40] creates multiple parallel processors out of a single processor core. While conventional hyper-threading uses different execution data paths to mimic more processors, this approach divides the individual thread processing lanes. This creates multiple SIMT processors of varying warp-sizes from a single SIMT core. Using a single warp to schedule parallel threads for hyper-threading, no conflicts will occur between processing lanes when forming SIMT hyper-threading processors (See figure 4.1c). Conventional hyper-threading using the SIMT thread processor's execution pipelines can also be implemented along with this proposed hyper-threading architecture.

To determine how to divide the individual thread lane processors for hyper-threading, the per-warp stack reconvergence table is extended. I present three algorithms for managing the stack entries, as this could be customized for specific applications. The first presented algorithm is used to schedule stack entries with no thread conflicts with other stack entries. With this method, only diverging control flow produced from if-else or case statements will utilize the hyper-threading architecture. The second approach spatially divides all thread lane processors into even groups based on the number of instructions the core is able to fetch. This method is

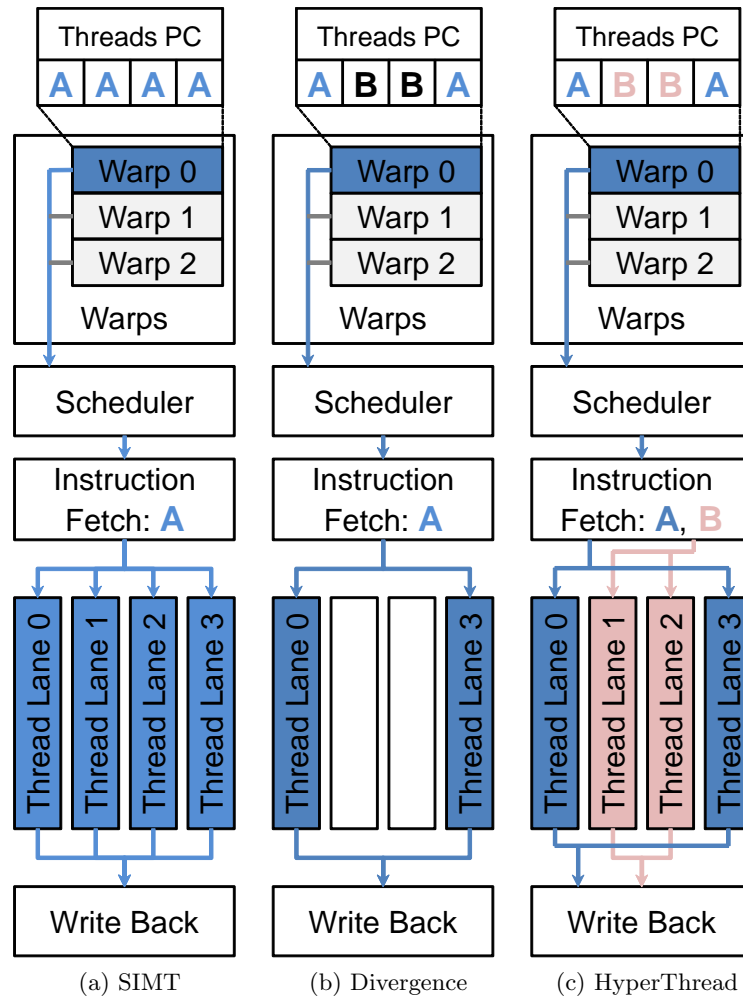


Figure 4.1: Single-Instruction Multiple-Thread Architecture for processing a warp. a) No thread Divergence. b) Thread divergence. c) Our 2 way hyper-threading architecture.

similar to reducing the warp size while keeping the processor core count the same. The only exception is the warp size is not decreased. For a warp to finish, all spatially divided thread lanes must finish before a warp is completed. The final approach attempts to maximize the thread lane utilization by picking two instructions in the warp stack that will result in the best performance.

4.1 Warp Divergence

A limitation of running threads in warps is that each thread in a given warp will only be complete when all threads in its warp are completed. If one of the threads in the warp

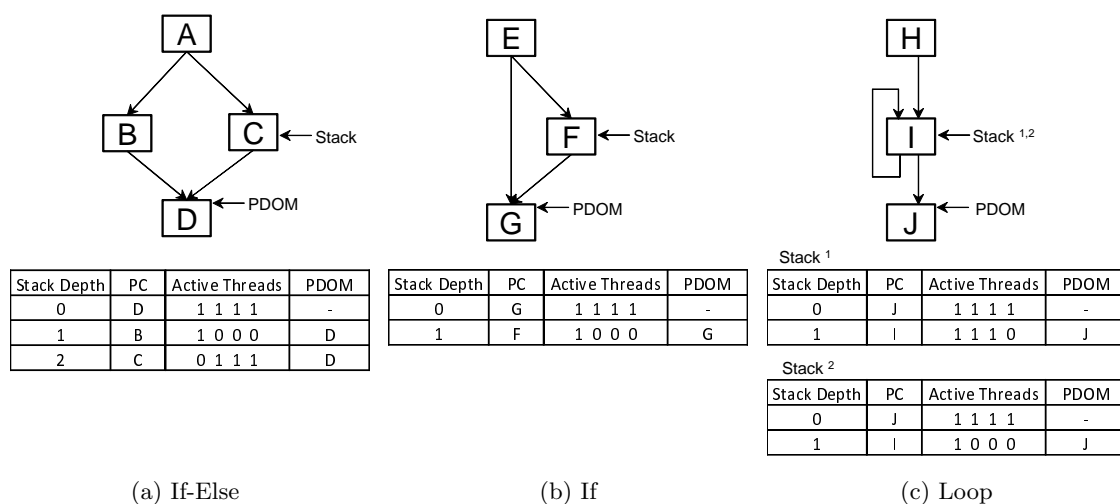


Figure 4.2: Different categories of warp divergence. (a) results in two different control flow paths before all threads reconverge at 'D'. (b) Only a single diverging control flow is required before threads reconverge at 'G'. (c) A looping operation where one iteration has 3 threads active and a later iteration has 1 thread. Similar to (b), only one diverging control flow is created before reconvergence at 'J'.

runs longer than all others, the other threads, while potentially complete, continue to consume idle execution cycles. The best performance time for all threads in a warp is equal to or greater than the time to process the longest running thread in the warp. Figure 4.2 shows different categories of predicated jump instructions that result in branch divergence and how each category effects the processor thread lane utilization. Figure 4.2a shows a divergence that results in two control flow paths before the paths reconverge (such as code generated from if-else statements). In this case, three stack entries are used where the bottom stack entry represents the reconvergence point and the other two entries represent the two control flow paths. Both control flow paths are executed resulting in every thread lane going unutilized at some point during the execution of this diverging statement. The second category is shown in figure 4.2b where one of the diverging control flow paths jumps straight to the reconvergence point such as a single if statement. In this case, only the threads that jump to the reconvergence point go idle and wait for the other control flow path to reach the reconvergence point. Only two stack entries are needed, one for the reconvergence point and the other for the single diverged control flow path. The final case is a looping category shown in figure 4.2c. Divergent branching of loops occurs when the number of loop iterations is different per thread, most commonly from

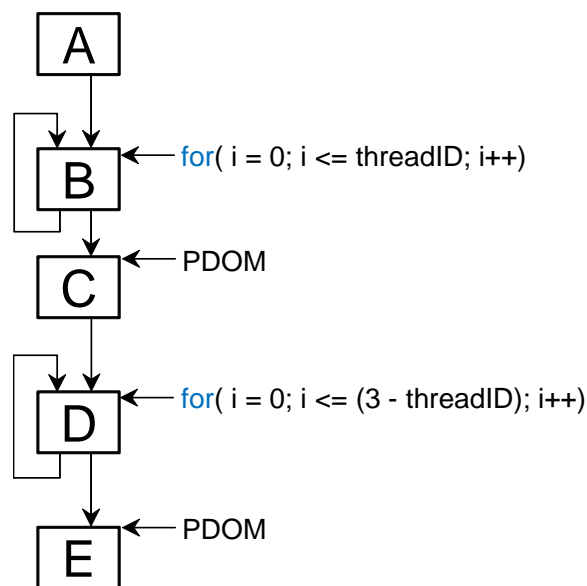


Figure 4.3: Example application containing multiple diverging loops where threads run a varying number of iterations per loops. In PDOM, the second loop can only be run after all threads finish the first loop. Using Hyper-threading, threads that finish the first loop can begin executing the second loop before all threads finish the first loop.

using thread specific data. While this case is similar to figure 4.2b where one of the branching paths results in the reconvergence point and the other is a separate control flow path, the number of times the separate control flow path is executed may vary between threads. The thread processing lane utilization can also decrease during every loop iteration. Only when the last thread exits out of the loop will all threads be able to continue.

4.1.1 Parallel Execution of Diverged Warps

Accelerating if-else diverging branches (Figure 4.2a) can be done by scheduling both diverging control flow paths in parallel. If the two control flow paths have different lengths, then both are run in parallel until they reach the reconvergence point. Once one path reaches the reconvergence point, the threads go idle until the other path reconverges. The expected performance gain for this is $\max(pathA, pathB)/(pathA + pathB)$.

Increasing performance for if statements and loop diverging branch categories cannot be done for just a single diverging point in an application. Running parallel control flow paths for these categories is not possible since one of the diverging paths is the reconvergence point

that the alternative path will still execute. To accelerate these diverging categories, multiple branching operations must be present. In figure 4.3 there are two sequential looping operations where the number of loop iterations are different for the same thread between the two loops. In conventional SIMT scheduling the time required to finish both loops will be the sum of the time that the longest thread took in loop one and the longest time a thread took in loop two. In this example the longest time in loop one is from thread ID 3 of 10 iterations and for loop two it is thread ID 0 of 10 iterations. Allowing threads to execute in parallel past their reconvergence point may be beneficial if there are additional branching statements. In our example from figure 4.3, allowing thread ID 0, that requires the longest time in loop 2, to start in loop 2 while other threads are in loop 1 is possible since thread 0 is able to exit out of loop 1 before the other threads. This results in thread 0 not having to wait for all threads to finish loop 1.

In both cases the best performance time possible for a warp is still limited to the time required to process the longest running thread. These parallel execution methods for diverging warps are designed to get warp processing time closer to the best case warp performance time. This is done by reducing the time that the longest running thread is idle due to branch divergences from other threads in the warp that run longer than the longest running thread in a specific branch. Warps with execution time already equal to the time required to process the longest thread will not see any performance improvement.

4.2 SIMT Hyper-Threading Architecture

Hyper-threading [40] allows for a single processor core to mimic a multi-core processor by allowing multiple threads to use different processor core components, such as a floating point unit and integer pipeline. To support multiple threads being executed in parallel, additional hardware is implemented to add capabilities to fetch multiple instructions and perform the necessary register access. In this implementation of SIMT hyper-threading, the individual thread processing lanes are the processor core components that are intermixed to create multiple virtual parallel SIMT processors from a single core. Additional hardware resources are required to allow for multiple instruction fetching from the same warp.

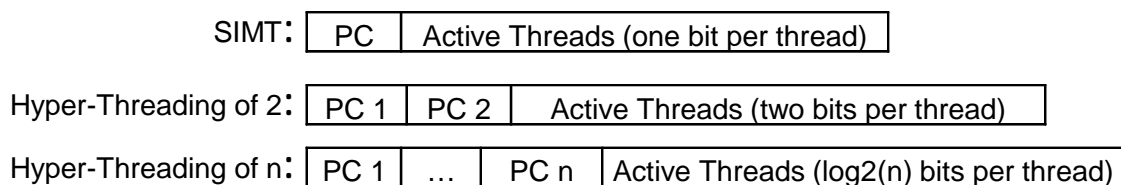


Figure 4.4: Warp meta-data fields for both PDOM and hyper-threading warps. Conventional SIMT warps only requiring knowing a single instruction PC and what threads out of the warp requiring executing this PC. For hyper-threaded warps, multiple instruction PCs are required in addition to expanding the active thread mask to indicate which threads are active and which instruction PC is required.

The TLP scheduler for selecting warps to execute is not changed however the data structure used to represent warp meta-data is expanded to allow additional PC fields as well as the active thread field such that each thread has the required bit representation to select from the available PCs. The expanded warp meta-data is shown in Figure 4.4. The number of virtual processors that hyper-threading is able to create is implementation dependent and can range from 2 up to the warp size. With increasing hyper-threading virtual processor count, the warp meta-data will grow to store all the required PCs and active thread masks.

The warp meta-data retrieved from warp scheduling contains all the required PCs to be fetched. The port size for the Instruction Cache is increased to support the maximum number of instructions that may need to be fetched. A cache miss for any of the instructions results in all remaining PCs waiting for the cache miss to be resolved. Once all instructions are fetched, the individual thread processing lanes select which of the instructions to execute using a mux with the selecting signal being the warp meta-data active thread mask as shown in Figure 4.5. Individual processing lanes executing shorter latency instructions must wait for all processing lanes to finish before the write-back pipeline stage. A waiting pool is used for storing threads that finish before others.

Setting the PCs and active thread masks field in the warp meta-data for diverging instructions is performed using custom instructions that are inserted by a just-in-time compiler. These instructions update the per-warp stack and also the PCs warp meta-data when divergence occurs. The locations for these instructions are implementation specific allowing for multiple

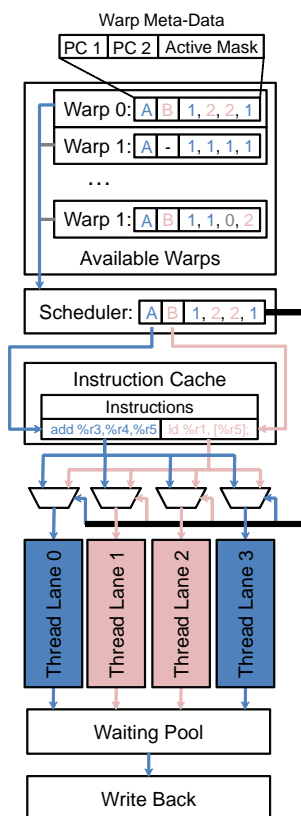


Figure 4.5: SIMT 2 way hyper-threading architecture resulting in two virtual processors.

stack based algorithms to be tested. Typically instructions are inserted at all possible points of control flow divergence and reconvergence.

4.2.1 Stack Manipulation Algorithms

In this section I present three hyper-threading per-warp stack modification algorithms. These algorithms are not necessarily intended to reflect ideal solutions, as more tailored algorithms can be generated for specific applications to further improve performance. Instead, the three presented below are designed for general applications and to study the performance effects across various methods.

4.2.1.1 Parallel Stack Entries

This stack modification algorithm uses hyper-threading to schedule only stack entries that have no thread conflicts with other entries located above it in the stack (Figure 4.2a). Only

Algorithm 1 Parallel Stack Scheduling

Require: Depth = Stack Depth
Require: Stack = Stack Location
Require: NumSch = Hyper-Threading Size
Require: MetaData = Warp Meta-Data
 1: # Schedule top stack entry
 2: MetaData.PC[1] \leftarrow stack[Depth].PC
 3: MetaData.ActiveMask \leftarrow stack[Depth].ActiveMask
 4: # Search through rest of stack for non-conflicting entries
 5: numScheduled \leftarrow 1
 6: index \leftarrow Depth-1
 7: scheduledMask \leftarrow stack[Depth].ActiveMask
 8: **while** (numScheduled \leq NumSch) AND (index \geq 0) **do**
 9: **if** (stack[index].ActiveMask & !scheduledMask) = 0 **then**
 10: # Found stack entry with no thread conflicts
 11: # Add entry to warp meta-Data for scheduling
 12: numScheduled \leftarrow numScheduled+1
 13: MetaData.PC[numScheduled] \leftarrow stack[index].PC
 14: MetaData.ActiveMask | = stack[index].ActiveMask
 15: scheduledMask | = stack[index].ActiveMask
 16: **end if**
 17: **end while**

diverged control flow paths that result in parallel paths before the PDOM point will be scheduled. Manipulating the stack entries is done using the same algorithm used for PDOM where two additional stack entries are added to the stack. After each stack modification a scheduling algorithm is called to determine which stack entries to schedule. The algorithm to determine which stack entries to schedule is shown in Algorithm 1 and first schedules the top stack entry. After that it searches through the rest of the stack to find any other entries that could also be scheduled. Bit masking operations are applied to check for non-conflicting threads in all stack entries. If a valid stack entry is found, the warp meta-data is updated to include the required information. In this scheduling method PDOM reconvergence points are still used to reconverge thread paths. Thread synchronization require no additional implementation since synchronization points requires all threads in a warp to be converged, which is taken care of using conventional PDOM.

4.2.1.2 Work Ahead

To potentially accelerate branching categories in Figure 4.2b and 4.2c, threads must be allowed to work ahead of PDOM points. The stack entry representing the work at a PDOM

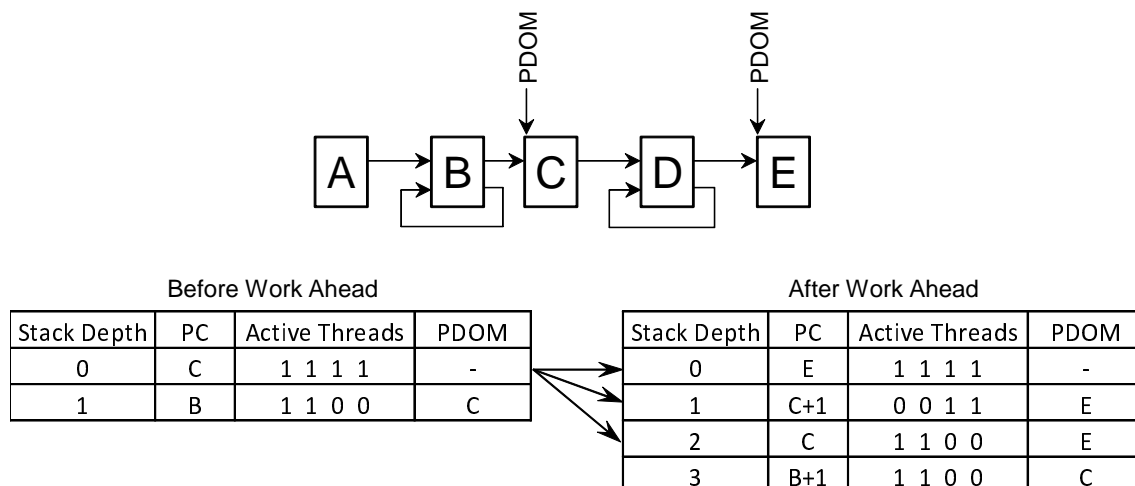


Figure 4.6: Sample stack modification for allowing threads at a PDOM point to work ahead. The PDOM stack entry (depth 0) is split into three new stack entries: 1) For the next PDOM to reconverge all threads back together. 2) Stack entry for threads working ahead. 3) Stack entry to progress threads currently not at PDOM point to progress to the next PDOM point.

point must be split into three stack entries because some of the threads in this entry are dependent on stack entries further down the stack (see figure 4.6). Determining the non-dependent threads is done using bit masking operations and traversing all stack entries between this entry and the top of the stack. The lower of the three stack entries contains the original active thread mask and the PC of the next PDOM point from the current PC. The next stack entry is for all the threads that are allowed to work ahead and has a PDOM value set to the next PDOM point. The third stack entry added is the original PDOM stack entry except the active thread mask is reduced to only the threads that could not work ahead due to other stack entry dependencies. By creating the third lower level stack entry for the next PDOM reconvergence point, we allow a method for threads working ahead to reconverge with the original stack entries in the case that original threads left behind eventually catch up to these threads.

To determine which PDOM point to work ahead for in the case there are more options than hyper-threaded processors, each PDOM point is given a priority. Priority is computed off-line during compile time and is set based on the shortest path to a looping jump instruction. In this implementation a higher priority is given to a nested looping operation than the outer

loop. While this method favors looping operations over other diverging operations, loops are given the highest likelihood of acceleration since they are performed multiple times compared to other branching methods.

Loop Lapping is a common source of performance gain for this method and occurs when a thread exits a loop and while working ahead enters back into the same looping operation. To improve the performance when loop lapping occurs, since threads that have worked ahead are in a different stack entry than the other threads that are all in the same loop, the stack modification algorithms (when triggered by a diverging instruction or threads reaching PDOM points) scans all entries to determine if multiple stack entries could be merged together in the warp meta-data. By merging them in the warp meta-data, similar stack entries only consume a single hyper-threaded processor.

4.2.1.3 Spatial Subdivision

Spatial subdivision stack modification algorithm attempts to mimic smaller warp sizes by dividing the threads into smaller equal sized groups at the first diverging point (Figure 4.7). The new group sizes are determined by the number of hyper-threading virtual cores that can be created. For example, a 2-way hyper-threading with 32 thread per-warp will divide the stack into two 16-threads per group. Each group then behaves similarly to PDOM where each of the top stack entries for each group is scheduled. SIMT hyper-threading allows for the smaller thread groups to run in parallel, however the overall runtime for the entire warp is still limited to the longest running thread in the warp. To handle thread synchronization, all threads must be reconverged. Before a thread executes a thread synchronization instruction, the warp's stack is modified such that the stack entry at level 0 has the PC for the synchronization point (the stack entry at level zero already contains the list of all active threads in the warp). All other stack entries for this group are then removed from the stack. Only when the stack entry is at level 0 is the synchronization instruction called.

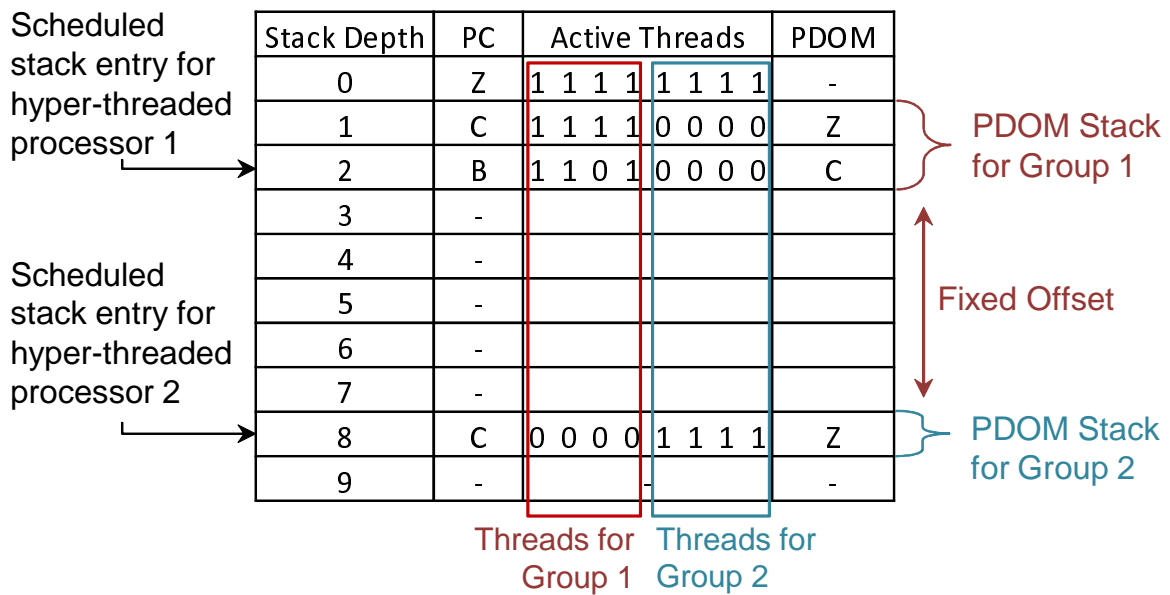


Figure 4.7: Warp stack layout used for spatial subdivision stack management algorithm. This layout is designed to mimic smaller warps using hyper-threading.

CHAPTER 5. Dynamic μ -Kernel Architecture

Our second proposed software/hardware branching solution is targeted for long diverging control flow paths. This solution breaks the static allocation of threads into warps to potentially achieve higher processor utilization than possible with the static thread allocation. While re-grouping thread allocations during runtime results in a slight processing overhead (making it not feasible for short branches), certain branching behaviors can see a large improvement in processor efficiency.

Our dynamic μ -kernel implementation is broken down into two components. The first part allows SPs to create new processing threads. The second part takes new threads and creates new warps that will not result in large divergent control paths. The process of creating threads at runtime and forming new warps is outlined in Figure 5.1. Threads are able to initiate the creation of threads inside an SP using a new instruction which we call *spawn*. The SM then groups the new threads with previously created threads that share the same targeted μ -kernel in the partial warp pool. Once enough threads are grouped to form a new warp, the warp is placed into a new warp FIFO and waits to be scheduled for execution. When a currently scheduled warp finishes, a new warp from the warp FIFO is issued using the same resources as the finished warp and placed into the active warp pool.

In addition to the SM hardware for issuing new threads, data must be passed from the parent thread to the newly spawned child thread that is intended to continue the work of the parent thread. Data required for the spawned threads cannot be passed using registers since the new thread is likely to be assigned a different SP than its parent. Depending on the applications requirement of persistent memory between μ -kernels, register states can be saved to either on-chip shared memory or off-chip global memory. When a thread is created at runtime, a memory pointer is provided to the thread allowing the thread to access its associated

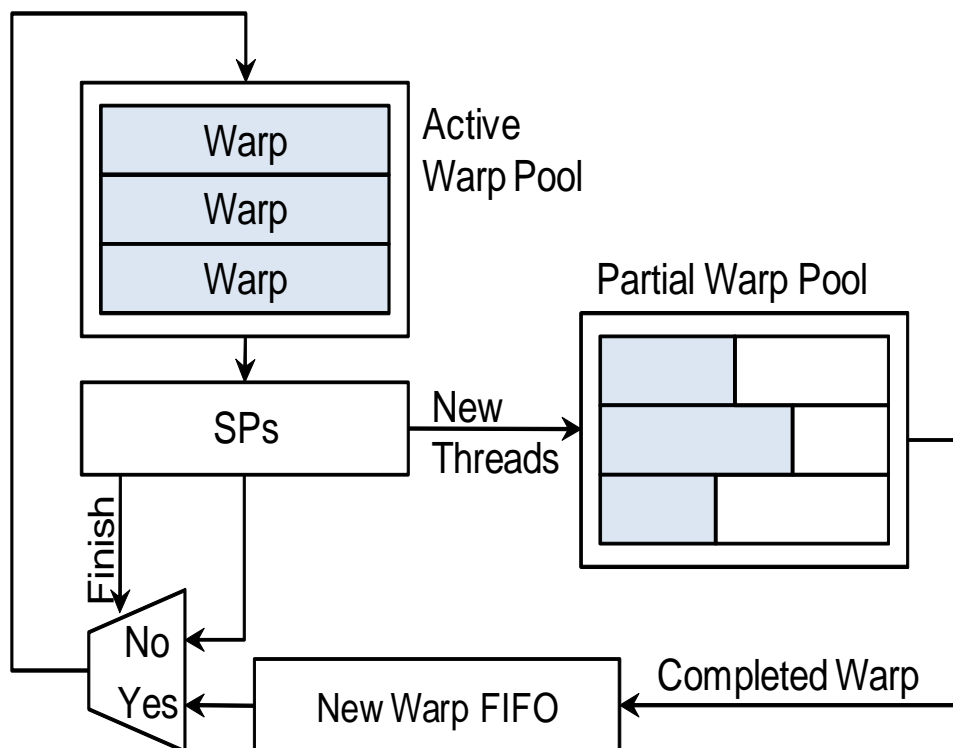


Figure 5.1: Dynamic thread creation hardware overview. New threads created by the SPs are placed into new warps waiting in the partial warp pool. Once enough threads have been created to complete a warp, the warp can replace an existing warp that has finished.

data.

5.1 Memory Organization

Similar to NVIDIA CUDA [48] supported architectures, the available memory spaces for a thread consists of registers, shared memory, local thread memory, global device memory, constant memory, texture memory, and a new memory space called *spawn memory*. Registers and shared memory are located on-chip and are tied to an SM. Local thread memory is stored in off-chip device memory. Local thread memory is reserved for register overflow to reduce register counts for kernels and also for any intermediate data storage that is too large for on-chip memory. Constant memory and texture memory also use device memory and are used similar to local memory except that they are read-only and can be cached. Global device memory is off-chip and is shared across all SMs, and can also be allocated and accessed by a host processor.

Spawn memory may be implemented in on-chip memory inside an SM or device memory. The memory space is used for two purposes: storing the data to be passed between threads and storing partial warps during warp formation (see Chapter 5.3). This memory space is allocated at kernel launch time since the size requirements can be computed off-line and are constant throughout application execution.

5.1.1 Thread Usage of Spawn Memory

The first section of this memory space is for storing data to be passed between threads. The allocation size is computed by the size of the data structure used for passed data between threads, multiplied by the number of threads that can be assigned to an SM. Since the size of the data structure may vary depending on what μ -kernel is being called, the largest data structure is used for the computation. This also requires that the μ -kernels must be defined before the application is executed. Individual threads in an SM are then assigned a region of this memory space.

The method that threads access their spawn memory space depends on how the thread is created and when it is scheduled. However, all threads use a special register called *spawnMemAddr* to determine the spawn memory address. Threads initially created and scheduled at application launch have their *spawnMemAddr* set to a unique address inside the spawn memory using the equation $SpawnMemoryBaseAddress + threadID * sizeof(dataToBePassedBetweenThreads)$. Dynamically created threads are provided a memory pointer in the *spawnMemAddr* when scheduled that is used to obtain the appropriate spawn memory address (see Chapter 5.4). Threads that were created at application launch but were not able to be scheduled due to resource availability require a spawn memory address that has been freed by a thread, made available when a thread exits from the last μ -kernel.

Data in the spawn memory space is what is passed between a parent and child thread. If a thread is creating a child, the parent thread will store its current state in the spawn memory before calling the spawn instruction. If the thread is a child, the spawn memory space is used to retrieve data from its parent thread. Child threads can reuse the same spawn memory address to pass data to its future children.

5.1.2 Partial Warp Formation

The second half of the spawn memory space is for storing dynamic threads during warp formation. The hardware components for creating new warps require consecutive memory addresses to store the meta-data of individual threads belonging to a new warp. The number of consecutive memory address is equal to the number of threads in a warp. The minimal size required for this memory is a function of the number of threads that can be assigned to an SM, the number of threads per-warp and the number of μ -kernels ($size = NumThreads + (SpawnLocations - 1) * WarpSize$). The size allocated in memory is doubled to prevent new threads meta-data from clobbering active threads. The spawn memory used for passing data between threads does not need to be doubled, since registers can be used to save original data when reusing the memory space for creating a child thread.

5.2 Spawn Instruction

Spawn instructions take two parameters: an assembly code label (converted by the assembler to the Program Counter (PC) value), used to indicate the new thread's μ -kernel and a register that contains the memory pointer to the thread's spawn memory space. The spawn function performs two key functions. First, it updates the warp creation hardware used to assign the new threads into warps. Second, it performs a memory write operation that is required by the warp creation hardware to save the thread's meta-data.

5.3 Warp Formation

Figure 5.2 shows the architecture for the spawn instruction for an SM. The first operation performed is thread classification where new threads are placed into warps. The PC value of the spawn instruction will be the same value for all threads executing the spawn instruction. The value is the same for all instructions since the PC value is statically compiled into the instruction and all instructions in a warp execute in lock-step. The PC is used in a look-up table (LUT) to determine the address in the spawn memory space where similar threads are being grouped into new warps. The functionality of the LUT is identical to the dynamic warp

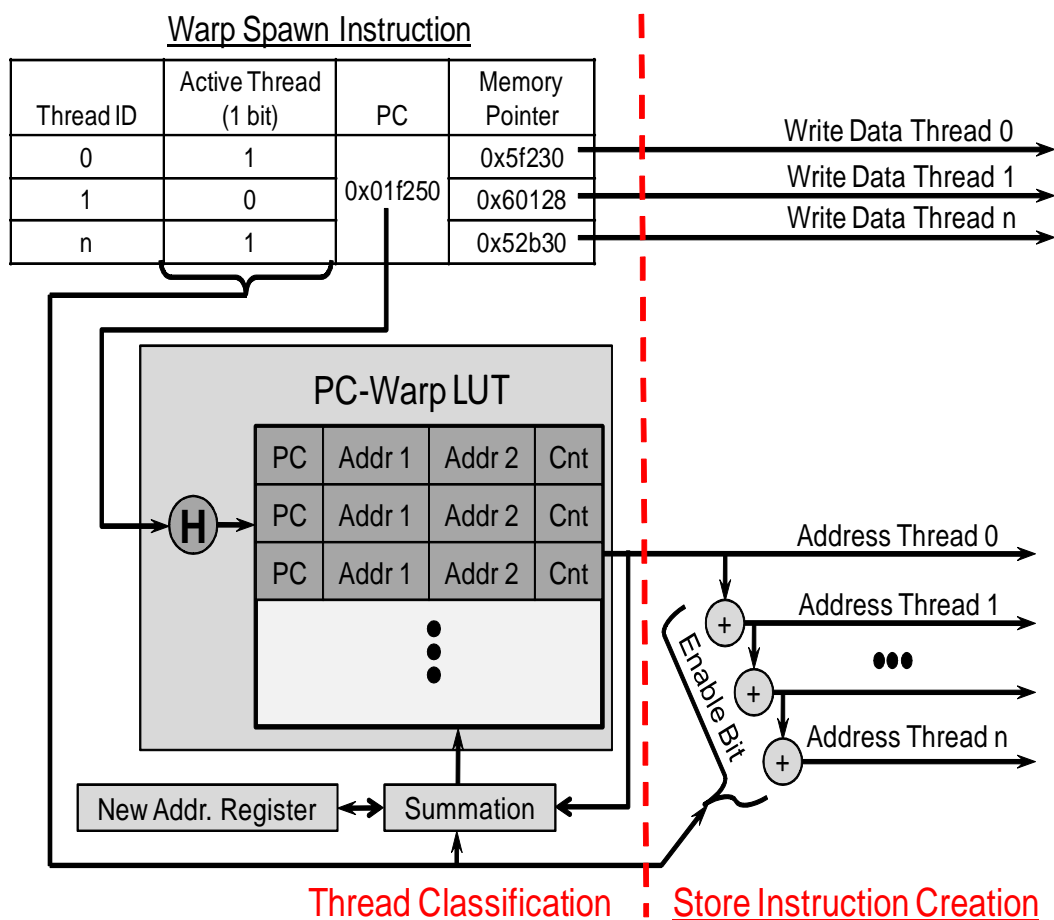


Figure 5.2: Architecture for the spawn instruction. Dynamically created threads identify existing threads that will follow the same control path using a look-up table. Once the warp has been identified for the new threads, the memory pointers are stored in memory for later use.

formation concept presented in [23], and provides the mechanism for taking new threads PC and providing an index to start forming new warps. The LUT is an on-chip memory organized as a fully associated cache where the number of entries is equal to the number of supported μ -kernels. The content of our LUT is different from that of [23]: two memory addresses and one counter variable are stored for each line in the LUT. The counter keeps track of the number of threads already contained in the partially created warp. The first memory address is the spawn memory address where the current warp is being created. The second memory address is the overflow address used for creating the next warp for the same PC. A single spawn instruction may result in more threads being created for a μ -kernel than the current warp being formed has room for. In this case the second memory address is required to create a second warp.

After the memory read from the LUT, the resulting first memory address and counter are both incremented based on the number of new threads (summation hardware). The number of threads is the sum of all active threads in the warp that executed the spawn instruction. Once incremented, both results are stored back into the LUT. If the counter is incremented over the size of a warp, the second memory address is incremented based on the overflow from the first memory address and replaces the first memory address. The second memory address is then set to the next available free memory address stored in a register. If the first memory address does overflow, this signals that a new warp has been created and is pushed into the new warp FIFO. The value pushed into the FIFO is the first memory address from the LUT that points to the spawn memory space containing the last thread in the finished warp.

The second operation is creating a store instruction to save the new threads meta-data into memory. The two addresses read from the LUT are used to compute a unique address for all threads executing the spawn instruction. Since not all threads of a warp are active, memory addresses used for the store instruction are not computed for all threads. The memory addresses from the LUT are pipelined to each thread channel and incremented if that thread channel is enabled. The result is that all active thread channels have a memory address that is both unique and sequential. Threads in the warp that are not executing the spawn instruction result in a duplicated memory address since the addition operation was disabled and the value is pipelined through the thread's channel to feed additional thread channels. Since the threads are not active, they will not perform the store instruction and ignore the duplicated memory address. In the event that a warp is filled before the end of all active threads, the second memory address is used to start forming a new warp. After a memory address has been computed for each thread, a memory store transaction is generated for storing the memory pointer to the new threads data being passed by the parent thread to the second half of the spawn memory.

5.4 Scheduling

Once a warp has been defined in memory, the scheduler attempts to schedule the warp for execution. Similar to scheduling threads defined at kernel launch, SM resources (such as registers and the number of threads that can be stored in the SM thread queue) need to be

available before beginning execution of a warp. To reduce the size of the dynamic warp FIFO, dynamic threads are given priority for scheduling over unscheduled threads defined at kernel launch. If not all SM resources are used during kernel launch, dynamically created threads can use the remaining available resources to be scheduled for execution. When currently executing warps exit, their resources are freed, which makes scheduling possible for additional warps.

Since spawn instructions can invoke multiple μ -kernels, dynamically created threads may require different amounts of SM resources. To allow for easier scheduling, all threads use the maximum resource per category required by each of the μ -kernels. This allows for any warp to replace an existing warp; the scheduler does not need to keep track of different warp resources. The tradeoff for this method is a decrease in the number of threads that can be actively executing if μ -kernels are not balanced.

To give each thread in a warp access to its data stored in spawn memory, the *spawnMemAddr* special register is set to the memory address pointing to the data in spawn memory. The spawn memory pointer is stored in memory by the thread grouping hardware during the spawn instruction. The memory address provided to new threads in *spawnMemAddr* is computed from the memory address from the LUT. The individual thread values are computed by taking the address from the LUT that was used to store the last thread in a warp meta-data to memory, and subtracting the thread ID inside the warp. Using this method, the size of the on-chip FIFO is reduced by a factor equal to the number of threads in a warp, since the individual thread address can be computed for a single value for the entire warp.

Scheduling new warps only when the warp is completely filled can cause threads to stall during thread formation due to a lack of remaining threads to finish the new warp. To prevent this problem, partial warps can be forced out of the thread pool and scheduled as incomplete warps. Threads are forced out only when the scheduler runs out of available warps to schedule. This happens only near the end of the application when all warps defined at launch time have finished and no new warps remain in the dynamic warp FIFO. The warp that is forced out is selected by the PC address of the μ -kernel, starting with the lowest PC address.

5.5 Programming Model for Dynamic μ -Kernels

Memory allocation for the spawn memory space occurs before threads are scheduled onto the SMs. The size of this memory depends on two parameters. The first parameter is the total number of threads able to be launched given the resource requirements. The second parameter is the amount of memory required to be passed between threads. Off-line, the compiler computes these two parameters, then uses them to determine the size of the spawn memory. When the sum of the spawn memory and shared memory (also computed off-line) exceeds the amount of on-chip memory, the hardware allocates the spawn memory to off-chip memory.

An assembly template for a μ -kernel is shown in Algorithm 2. Software instructions store and load the thread states. Before spawning a new thread, threads must save active data to memory (lines 14 and 15). Each thread has access to its spawn memory space through the *spawnMemAddr* special register created at thread launch (line 3). After the software instructions save the state of the thread to memory, the program calls the spawn instruction (line 17 or 19). The spawn instruction requires two arguments, a program counter for the μ -kernel to be executed and the spawn memory pointer. While the parent thread is free to continue executing application code, the thread must not modify the spawn memory space, as the usage of the spawn memory by the child thread cannot be determined by the parent thread and modifications by the parent thread could result in concurrency errors. To load the state for a dynamically created thread, the spawn memory address must be retrieved.

Figure 5.3 shows the memory layout used by threads to access spawn memory data. Dynamic threads are provided a memory address in *spawnMemAddr*. This address points to the warp formation section of the spawn memory space that was being written to when the thread was first created. The value stored in the warp formation memory is the memory pointer used in the spawn instruction that created the thread and points to the thread usage data (lines 3 through 5). Once the spawn memory address is retrieved, the state can be loaded into the thread's registers (lines 7 and 8). The contents of the spawn memory space is accessible for the lifetime of the thread. The spawn memory space is reused to spawn a new thread by writing data back into the memory space for the child thread (lines 14 and 15).

Algorithm 2 Sample μ -kernel Assembly Code

```

1: microKernel_label:
2:  # Get memory pointer to threads spawn memory
3:  mov      rd1, SREG.spawnMemAddr;
4:  ld.spawnMem  r1, [rd1+0];
5:  mov      rd1, r1;
6:  # Load thread state from spawn memory
7:  ld.spawnMem  f1, [rd1+0];
8:  ld.spawnMem  f2, [rd1+4];
9:
10: # Run micro-kernel code
11: # Sets P0 to select between two micro-kernels
12:
13: # Save thread state back to spawn memory
14: st.spawnMem  [rd1+0], f1;
15: st.spawnMem  [rd1+4], f2;
16: # Create a new thread
17: @p0 spawn $microKernel_option_1, rd1;
18: @p0 exit;
19: spawn $microKernel_option_2, rd1;
20: exit;

```

Requiring the current state of a thread to be saved to memory results in overhead for performing μ -kernel execution. Performing multiple load/store operations at the beginning and end of each μ -kernel results in additional instruction execution and increased instruction latency. The number of operations required is application specific and determined by the amount of memory required to save the state and the memory data layout. Care must be taken in determining the location in the application for creating dynamic threads, since the overhead may be more than the branch instruction.

Converting current SIMT rendering algorithms to use μ -kernels requires identifying critical branch statements that decrease processor efficiency more than the overhead for creating a dynamic thread. Branching statements can be either looping conditions or conditional branching statements that are typically data dependent and affect the runtime of the thread. In our ray tracing example, the three looping operations have a dramatic effect on the runtime for each thread and allows one long running thread to have a significant negative effect on the processor utilization.

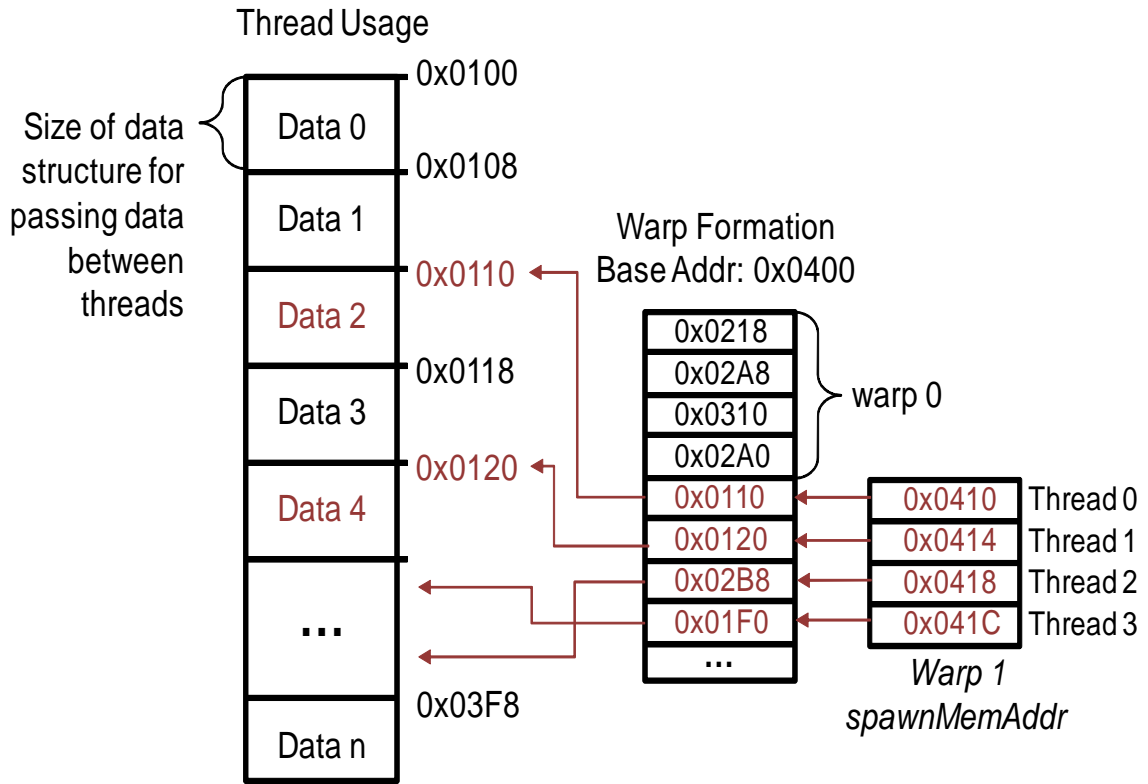


Figure 5.3: Spawn memory layout for threads accessing parent thread data using 4 threads per-warp and 8 bytes of storage between threads. Child threads use their special register to access the warp formation data. The warp formation data is a memory pointer to the parent thread data.

CHAPTER 6. CUDA Supported μ -Kernel

The initial μ -Kernel architecture outlined in Chapter 5 was targeted to accelerate ray-tracing algorithms that do not fully utilize all features of the CUDA programming model (such as shared memory and thread synchronization). To allow dynamic μ -Kernels to be a possible hardware/software branching solution, additional capabilities are required to support common features of current SIMT architectures.

6.1 CUDA Thread Hierarchy

To support CUDA applications, the μ -Kernel architecture must be extended to support thread hierarchy of block and grid properties. Since newly spawned threads can be combined with threads from different blocks, additional thread properties are required to be saved in spawn memory to allow threads access to block specific resources.

Thread block synchronization allows for threads in the same block to be synchronized to the same PC. PDOM thread synchronization is done by counting the number of warps that execute the synchronization instruction, where each block has its own counter. Warps that execute this instruction are not allowed to be scheduled again until the count of all warps at a synchronization point equals the number of warps in the block. Since μ -Kernels can re-group threads from different blocks into the same warp, counting warps is no longer possible. The PDOM thread synchronization method is extended to count threads instead of warps. The thread synchronization instruction argument list is then extended to include the block ID to which the thread corresponds too and can increment the appropriate counter. Once the total number of threads at a synchronization point for a block is reached, the block is flagged as being released from the synchronization point. For a warp to be released, all blocks having a

thread in the warp must be flagged as released. Bit masking operations are used to perform this check.

6.2 Improved Thread Spawning

Initial μ -Kernel spawning was naïve in the fact that a thread was spawned at specific branching locations without any information as to if the warp was diverging or if the performance would be better not spawning. Since the performance of μ -Kernels is a function of how many other warps are spawning to the same location, spawning new threads will only improve performance if there are enough threads to be recombined into new warps. To account for this, a performance model is added that evaluates the performance of each branching method at runtime to select the optimal solution.

6.2.1 Run-time Performance Model Execution

For Dynamic Micro-Kernels to improve performance, multiple warps are required to execute the same branching instruction. The more warps at a branching instruction, the more threads there are to form new warps. However waiting for multiple warps to reach the same branching instruction can also reduce performance. To keep track of how many warps are available at a particular branching point we introduce a new instruction called *diverge*. This instruction delays the warp from being executed until overall performance will decrease by not executing this warp. Delaying the warp from being executed is done by changing the warp schedule priority to a lower value. The *diverge* instruction can be inserted by the compiler as the first point of divergence.

Unlike the ray-tracing application targeted in Chapter 5, general CUDA applications may also require spawning new threads at the PDOM location of *diverge* instructions. In the ray-tracing application, spawning locations were selected to improve the performance for the nested loops. The three locations that were selected also ended up being PDOM reconvergence locations allowing for spawned threads to be reconverged into new warps at the PDOM location. Other CUDA applications are not guaranteed to have the same control flow structure and will require additional processing to ensure that when a warp reaches a PDOM location, it has at

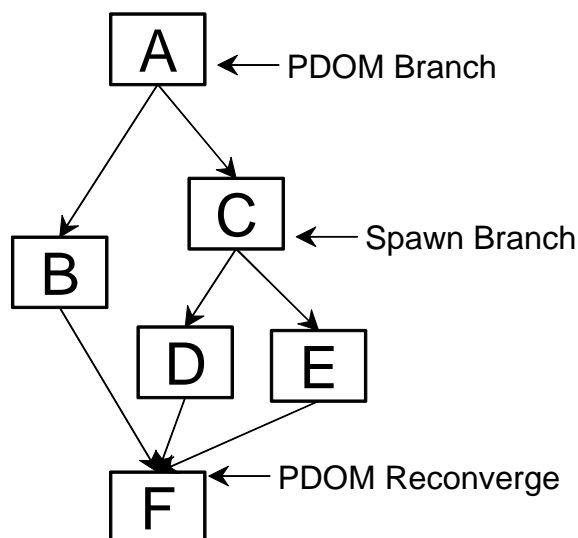


Figure 6.1: Potential processor efficiency lose from using Micro-Kernels. The nested branching operation at C uses the spawn branching method. Thread reconverging at block F see less efficiency then using PDOM branching at C since their are no threads to reconverge with.

least the same occupancy as warps using PDOM branching methods. Once such example is when the *diverge* instruction is performed in a nested branch with the non-nested branching method being PDOM (see Figure 6.1). In this case, only a fraction of the threads will spawn new threads and exit. The remainder of the threads in this warp from the alternative control flow sequence will continue executing. Once this warp passes the PDOM location at block F, the processor efficiency will be less than PDOM for the remainder of the application since threads executing the alternative control flow path at block C were spawned into new warps and are unable to be reconverged. To prevent the processor utilization from decreasing, *reconverge* instructions are inserted just before executing the PDOM location (at the end of blocks B, D and E). The *reconverge* instruction behaves similarly to *diverge* except the branching criteria is the number of exited threads in the warp. If a warp is executing the *reconverge* instruction and all threads in a warp have not exited (no spawning instructions have been performed for this warp), then the warp jumps to the PDOM location. If any of the threads in the warp have exited, then all threads are spawned to the PDOM point so they can be recombined with other threads for improved processor efficiency.

The *diverge* instruction is also used to start evaluating the run-time performance model for

selecting a branching method. The instruction operands are: the thread predicate register for all threads in the warp to determine the branching direction and two labels representing the PCs for the two different thread execution paths. If all threads have the same predicate value, all threads branch to the corresponding PC and no other action is performed.

If the predicate registers for the *diverge* instruction are different for different threads, indicating branch divergence, the warp is marked as being at a branching location. Once the warp is marked at a diverging point, the warp can only be issued until there are no other warps to be issued (decreasing the scheduler priority). At the same time the performance model for this diverging location is updated with the number of threads for the two control paths. When there are no eligible warps to be issued, the oldest diverging warp is allowed to be scheduled. Once released the warp checks the performance model to determine if Dynamic μ -kernels should be used. If Dynamic μ -kernels is predicted to improve performance, the warp PC value is set to the next instruction after the *diverge* instruction plus two. This is where the instructions for doing Dynamic μ -kernels are located. If PDOM is chosen then the next instruction after *diverge* is the PDOM branching instruction. Algorithm 3 shows the assembly code layout for a single branching location. Before the *diverge* instruction is called, the predicate register must be set. If all threads follow the same path, the threads branch to the appropriate label (Label_True or Label_False). If PDOM is used as the branching method then the threads also branch to the same labels using the instruction after the *diverge* instruction. If Dynamic μ -kernels are used, the *diverge* instruction jumps ahead by two instructions (over the PDOM branch instructions). If new threads are created using Dynamic μ -kernels, the starting locations are the labels marked with *spawn*. The instructions following the *spawn* label allow the threads to restore the active state and then continue along the correct control flow path.

By delaying the scheduling of a warp at a diverging location, other warps may reach the same diverging location and increase the likelihood that Dynamic μ -kernels will improve performance over PDOM. However if the SM is about to stall due to a lack of available warps, a warp is released to prevent processor inefficiency.

Algorithm 3 Assembly code for branch selection

```

1:  # Set predicate reg p0 before calling diverge
2:  diverge p0, Label.True, Label.False;
3:  @p0 bra Label.True;
4:  bra Label.False;
5:  # Start of spawning a new thread
6:  @p0 spawn Label.True_spawn;
7:  spawn Label.False_spawn;
8:  exit;
9: Label.True_spawn: # Start location for spawned thread
10: # Restore Active Registers
11: Label.True:
12: ...
13: ...
14:
15: # Label for second branch location
16: bra Label.False;
17: Label.False_spawn: # Start location for spawned thread
18: # Restore Active Registers
19: Label.False:
20: ...

```

6.2.2 Divergence Instruction Hardware

The data used for the performance model is stored in a Look Up Table (LUT) implemented as memory on chip for each SM. Each entry in the LUT corresponds to one of the branching instructions. The LUT is implemented as a Set-Associated Cache where the set size is equal to the number of supported diverging locations. The PC for each branching instruction is used as the index for each line. If there are more branching instructions than lines in the LUT, the compiler should limit the number of diverging instructions and use PDOM branching. The LUT stores the static information for each branching instruction along with a total running count for all threads at the branch location and their branching control flow path. The static information is populated at the start of the application and the diverge instruction is used to update the running thread count (See Figure 6.2). When the information in the LUT is changed, the performance model is re-run and the result is stored back in the LUT. When a warp at a branching point is scheduled, it checks the performance model result bit first to see if it needs to update the PC being issued. If a change is indicated the PC is read from the LUT. The thread count is also decremented as the warp is no longer at this branching location.

The total size for each line in the LUT is 45 bytes, assuming 32-bit PC values and a max

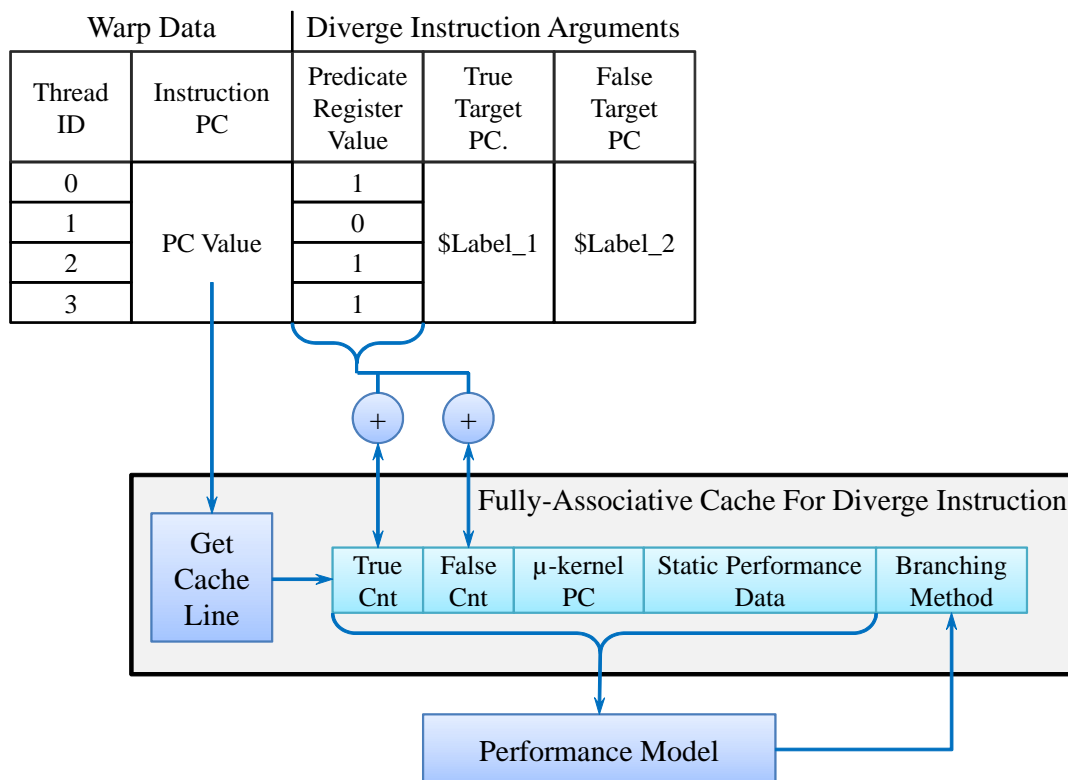


Figure 6.2: Implementation for the diverge instruction and updating parameters used by the performance model.

of 1024 threads per SM. Two 10-bit counters are used for counting the number of threads per path and cover the worst case of all threads branching to one path. A single 32-bit PC value is stored for the location of the *diverge* instruction. 128-bits are used for storing the static control flow results (storing four 32-bit values) which are used to estimate the number of instructions that will be executed.

6.3 Performance Model

A performance model is used both at compile time and runtime to evaluate potential branching performance. At compile time, each branching location is evaluated to determine if their is an optimal branching solution. If one is found, the code for that branching method is inserted. More commonly a clear method is not determined and the use of the *diverge* instructions is inserted that uses the run-time performance model to select the branching method.

The same algorithm is used as the performance model at both compile time and run-

time. Since the performance model uses the number of threads at a branching instruction, the compiler uses two performance models for each instruction, assuming different thread counts. The values used for the thread counters at compile time are the limits on the number of total threads. If all performance models result in the same branching method, the compiler inserts the code for that particular method and does not use the runtime model.

The performance model used during runtime determines which branching method should be used given the current state of all warps on an SM. To select the branching method, the runtime for both methods is predicted. Predicting runtime requires knowing the number of threads for each control flow path and having an estimate of the number of instructions that will be executed. Since Dynamic Micro-Kernels also spawn threads at PDOM reconvergence points, the count of the number of instructions also includes the overhead of executing the *reconverge* instructions. Since the number of instructions that can be executed for one of the control flow paths can vary due to additional nested branching, the minimal number of instructions is computed to the PDOM location. If a dynamic loop is found, an estimate on loop iterations is needed to count the maximum number of cycles. Using the minimal number of instructions before the PDOM is a conservative method for estimating the worst case performance for Dynamic Micro-Kernels.

The PDOM performance is computed by summing the performance to PDOM location:

$$PDOM_{performance} = (\text{Before PDOM}) + (\text{After PDOM})$$

$$\text{Before PDOM} = \# \text{ warps diverging} * ((\# \text{ instructions path 1}) + (\# \text{ instructions path 2})) \quad (6.1)$$

Performance before PDOM is the summation of the number of instructions for both control flow paths multiplied by the number of warps at the branching location. Since all warps will execute both control flow paths, we sum all instruction in both branches and multiply it by the number of warps at the diverging location.

Dynamic Micro-kernels performance is broken up into three parts (performance for path 1,

performance for path 2 and Dynamic Micro-kernel overhead) that are summed together:

$$\begin{aligned}
 \mu - \text{Kernel perf} &= (\text{perf for path 1}) + (\text{perf for path 2}) + (\mu - \text{Kernel overhead}) \\
 \text{perf for path } x &= (\# \text{ warps created for path } x) \\
 &\quad * (\# \text{ instructions for path } x \text{ to PDOM} + \# \text{ instructions after PDOM}) \\
 \mu - \text{Kernel overhead} &= ((\# \text{ warps diverging}) * (\text{overhead for storing state})) \\
 &\quad + ((\# \text{ warps being created for both paths}) * (\text{overhead for restoring state})) \\
 &\quad + (\text{overhead for spawning reconverge threads}) \\
 &\quad + (\text{overhead for restoring reconverge threads})
 \end{aligned} \tag{6.2}$$

The performance for one of the paths is the number of instructions in the path plus, multiplied by the number of warps required for that path. The overhead for using Dynamic Micro-kernels comes from storing and loading the active register state for both the *diverge* and *reconverge* operations, which is known at compile time. Computing the overhead then is just taking the summation of number of warps at the diverging location, number of warps needed for each control flow path, and multiplying by the number of cycles needed to store/load the register state (see Equation 6.2). The computation required for the performance model is done using fixed hardware logic.

CHAPTER 7. Experimental Setup

Evaluation of all proposed architectures was done using the GPGPU-SIM [7] simulator. This simulator is designed to simulate native PTX assembly applications that are generated using the NVIDIA CUDA compiler [48]. GPGPU-SIM also includes pre-processing of the PTX code that was used for inserting custom instructions when needed.

7.1 SIMT Hyper-threading

Table 7.1 shows the GPGPU-SIM simulator configuration for all SIMT hyper-threading tests except where stated elsewhere. All three SIMT hyper-threading stack modification algorithms are also implemented along with SIMT PDOM reconvergence for baseline performance comparison. Several tests were also simulated using SIMT PDOM using half the warp size and doubling the SIMT processor cores to maintain the same number of thread processing lanes as the SIMT Hyper-threading implementation.

Table 7.2 contains the benchmarks used for evaluating our hyper-threading architecture. These benchmarks are composed of both real applications with varying control flow complexity and synthetic benchmarks designed to test specific branching conditions. Benchmarks AES and STO contain no branch divergence where MUM and RAY both have complex control flow structure.

7.2 μ -kernels Setup

The spawn instruction and spawn memory space were added to the GPGPU-SIM simulator and configured to use on-chip SM memory. The simulator was configured to resemble an NVIDIA Quadro FX5800 GPU [45] with additional on-chip memory for the spawn memory

Number of SIMT Cores	15
Warp Size	32
# of Thread Lanes per SIMT Core	32
Threads per SIMT Core	1024
# of Registers per SIMT Core	16384
Shared Memory per SIMT Core	16KB
Number of Memory Modules	8
Memory channel Bandwidth	8 (bytes/cycle)
L1 and L2 Memory Caching	Perfect

Table 7.1: Hardware Configuration of the Simulator

Benchmark Name	Type	Discription
AES	Real	128-bit AES encryption and decryption [38].
BFS	Real	Breadth first search and single source shortest path algorithm [27].
BS	Real	Bitonic sort of 512 integers [15].
LPS	Real	Jacobe iteration for a Laplace discretisation on a 3D grid [4]
MUM	Real	DNA exact sequence alignment application [58].
NN	Real	A Neural Network solver [11].
NQU	Real	A N-Queen solver [52].
RAY	Real	Radius-CUDA ray tracing application.
STO	Real	Application to accelerate distributed storage systems [2].
EVEN	Synthetic	Synthetic benchmark for testing if-else branching divergence.
LOOP	Synthetic	Synthetic benchmark representing Fig. 4.3.

Table 7.2: Simulated Benchmarks.

space. Table 7.3 shows the specific configurations setup of our simulator for testing μ -kernels.

Simulations were done using two different thread scheduling models. The *block scheduling* configuration represents the FX5800 thread scheduling hardware. Warps are only scheduled via block scheduling if there are enough SM resources for the entire thread block, where thread blocks are composed of multiple programmer-defined warps. The block scheduling method allows for synchronization between all threads inside of a block. The *thread scheduling* configuration ignores block resources and schedules as many warps to an SM as other resources allow. Dynamic thread creation is designed for *thread scheduling* since thread synchronization is not required and allows for improved hardware usage.

Processor Cores	30
Warp Size	32
Stream Processors per-warp	8
Threads / Processor Core	1024
Thread Blocks / Processor Core	8
Registers / Processor Core	16384
On-chip Memory / Processor Core	64 KB
Spawn LUT Size / Processor Core	1024 Bytes
Memory Modules	8
Bandwidth per Memory Module	8 Bytes/Cycle
L1 and L2 Memory Caching	None

Table 7.3: Configuration used for simulation.

7.2.1 Benchmark Kernels

We used two benchmark CUDA kernels for our experimentation. The control algorithm is a ray tracing CUDA application called Radius-CUDA [8], which we used for performance measurements for the PDOM and MIMD configurations. Radius-CUDA uses a kd -tree [10] acceleration structure and Wald’s ray-triangle intersection algorithm [68]. The second benchmark kernel implements dynamic μ -kernels. The Radius-CUDA program was modified by removing the three looping operations and adding in state load/store operations and thread spawning. Modifying the Radius-CUDA application for dynamic μ -kernels is currently done at the Parallel Thread Execution (PTX) assembly language [47] level. To generate initial PTX assembly code and to determine the resources required for each μ -kernel, the original kernel written in CUDA C is broken up into multiple global function calls that are compiled separately. Individual global PTX functions are then manually combined into one large application containing all μ -kernels. For compilation of C code using NVIDIA’s NVCC compiler [48], function arguments were used to emulate the `spawnMemAddr` special register. During manual compilation of the PTX assembly code, this memory operation is modified to instead use `spawnMemAddr` as a register. The implemented algorithm used by the μ -kernels is the same for both benchmarks.

The per-thread kernel parameters are shown in Table 7.4. By using dynamic μ -kernels, the resources required per thread are less than the original kernel. This is a result of the μ -kernels using fewer instructions and the use of spawn memory space for additional register storage. Using μ -kernels results in 800 threads per SM. To achieve optimal performance for the

Resource	Traditional	μ -kernel	μ -kernel Minimum
Registers	22	20	16
Shared Memory	60 bytes	56 bytes	32 Bytes
Global Memory	388 bytes	384 bytes	0 Bytes
Constant Memory	128 bytes	24 bytes	8 Bytes
Spawn Memory	0	48 bytes	48 bytes

Table 7.4: Kernel processor resource requirements per thread.

traditional algorithm implementation using block scheduling, the block size is set to two warps, resulting in 64 threads per block and 512 threads per SM. The number of thread blocks that traditional hardware can run is then the limiting factor for the total number of threads that can run on a SM. While increasing the number of warps per block allows for a higher number of threads per SM, the additional un-coalesced memory operations resulting from high thread divergence degrades the performance below the performance of 64 threads per block.

384 bytes of the global memory per thread is for maintaining a stack data structure used for traversing the k d-tree. The value is arbitrarily chosen, since different tree structures represent different scenes which result in different stack memory sizes. The value is large enough for the largest scene in our benchmark dataset, however, the entire memory space is not used and usage varies between scenes.

The memory required for storing the state and for creating each dynamic thread uses the same data structure. The data structure requires 48 bytes of data to be stored and three 4-wide vector memory instructions are required for storing or restoring the state. The total memory then required for storing all possible thread states is less than the shared memory size. We implemented a naïve thread spawning method, where the entire store and restore operations for spawning a thread is performed for every loop iteration. Performance could be improved by first checking if a branch would cause divergence. If not, the branch could be taken by the warp instead of spawning new threads.

7.2.2 Benchmark Scenes

Three benchmark scenes were used for testing the performance and efficiency of our dynamic μ -kernels concept. Table 7.5 shows the rendered image and scene properties for each benchmark.


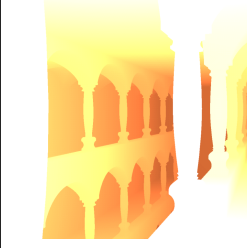
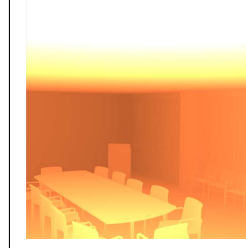
			
Benchmark	fairyforest	atrium	conference
Triangles	172,561	559,992	987,552
<i>kd</i> -tree Depth	36	37	35

Table 7.5: Benchmark scenes with object count and tree data structure parameters.

Number of SM Cores	1
Warp Size	32
Number of SPs per Core	32
Threads per Processor Core	1024
Thread Blocks per Processor Core	8
Number of Registers per Processor Core	16384
Shared Memory per Core	16KB
Number of Memory Modules	8
Memory channel Bandwidth	8 (bytes/cycle)
L1 and L2 Memory Caching	None

Table 7.6: Hardware Configuration of the Simulator

Fairyforest tests ray traversal efficiency for large open spaces with areas of highly dense object count. **Atrium** contains a uniform distribution of highly dense objects through the entire scene. The **conference** benchmark has a high number of objects that are not evenly distributed throughout the scene.

7.3 CUDA Supported μ -Kernels

Table 7.6 shows the configurations used for simulating μ -Kernels architecture that supports CUDA applications. Simulations are done with only a single SM using a represented workload from the benchmark. To fully evaluate our method we use a synthetic benchmark allowing us to easily change the algorithm to test different parameters. The algorithm for our synthetic benchmark is shown in Algorithm 4 and designed to represent workloads from real examples.

By changing the values of LOOP1, LOOP2, LOOPEND and the values stored in dataArray, we can test our performance model for a wide range of conditions.

Algorithm 4 Synthetic benchmark

```
1:  $index \leftarrow blockID * blockDim + threadID$ 
2:  $data \leftarrow dataArray[index]$ 
3: if  $data \geq 16$  then
4:   for  $i = 0 \rightarrow LOOP1$  do
5:      $data \leftarrow data + 1$ 
6:   end for
7: else
8:   for  $i = 0 \rightarrow LOOP2$  do
9:      $data \leftarrow data + data$ 
10:  end for
11: end if
12: for  $i = 0 \rightarrow LOOPEND$  do
13:    $data \leftarrow data * 2$ 
14: end for
```

CHAPTER 8. Experimental Results

8.1 SIMT Hyper-threading

Threads grouped into warps still present a performance bottleneck since all threads in the warp must finish before freeing up resources. To determine the maximum potential performance gain using warps, we simulate all benchmarks using hyper-threading where each thread can execute their own instruction (Full Hyper-Threading (HT)). Figure 8.1 shows the results compared to a MIMD architecture using the same size processor count. The simulation is configured to use a perfect memory system in order to eliminate the differences in MIMD and SIMT memory systems and focus on processor efficiency from control flow. Using hyper-threading can achieve similar results to MIMD for several of the applications. Applications such as BFS, BS, MUM, NN and RAY do not get the same performance using full hyper-threading as MIMD. These applications have threads with different run-times that prevent full hyper-threading from additional performance improvements.

Figure 8.2 shows the performance using the different hyper-threading stack modification algorithms and 2 instruction fetches per-warp. Different stack modification algorithms perform differently on the benchmarks. The EVEN benchmark uses if-else statements where the Parallel modification is designed to accelerate this type of branching. The LOOP benchmark contains loops where loop lapping is possible, due to this the work-ahead algorithm is able to accelerate this benchmark. While several of the benchmarks are able to get close to their maximum potential performance gain using hyper-threading, benchmarks such as MUM, NQU, RAY and LOOP are not able to get their maximum potential performance from 2-way hyper-threading. Figure 8.3 shows how the performance changes with increasing hyper-threading virtual processor count.

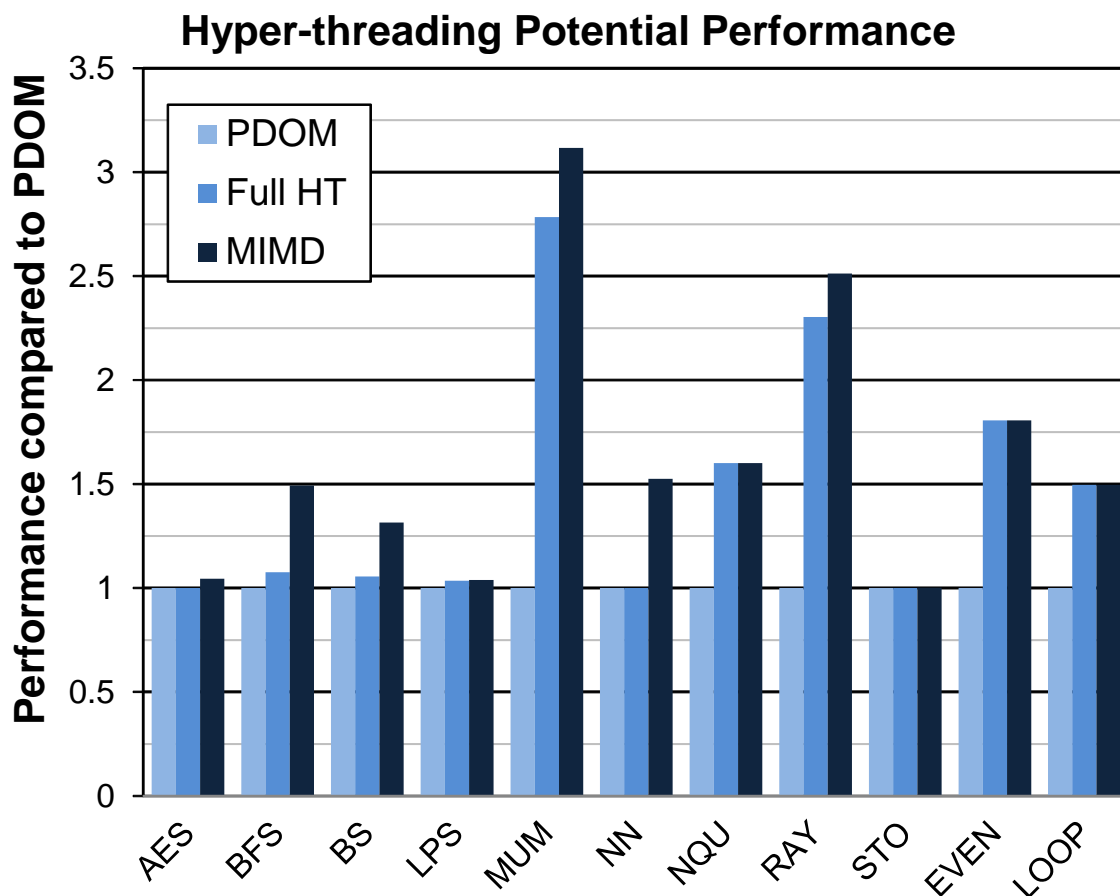


Figure 8.1: Limitations on the performance gain using hyper-threading on warps.

An alternative method to hyper-threading is to reduce the size of warps. While implementing these different architectures have different implementation requirements, we can compare performance results. While hyper-threading can be implemented with smaller warp sizes, we compare hyper-threading of 32 threads per-warp with a conventional SIMT architecture of 16 threads per-warp and double the number of SIMT processor cores to maintain the same number of thread lane processors. Figure 8.4 shows the simulation results. Benchmark BS is not able to utilize the additional SIMT processors due to smaller block sizes and results in a performance drop due to additional underutilized processing lanes. Benchmarks such as EVEN, LPS, NQU and LOOP are able to see a high performance improvement with hyper-threading since the utilization of the thread processing lanes are able to be customized to the needs of the application. Other benchmarks benefit from the performance gain that smaller warp sizes

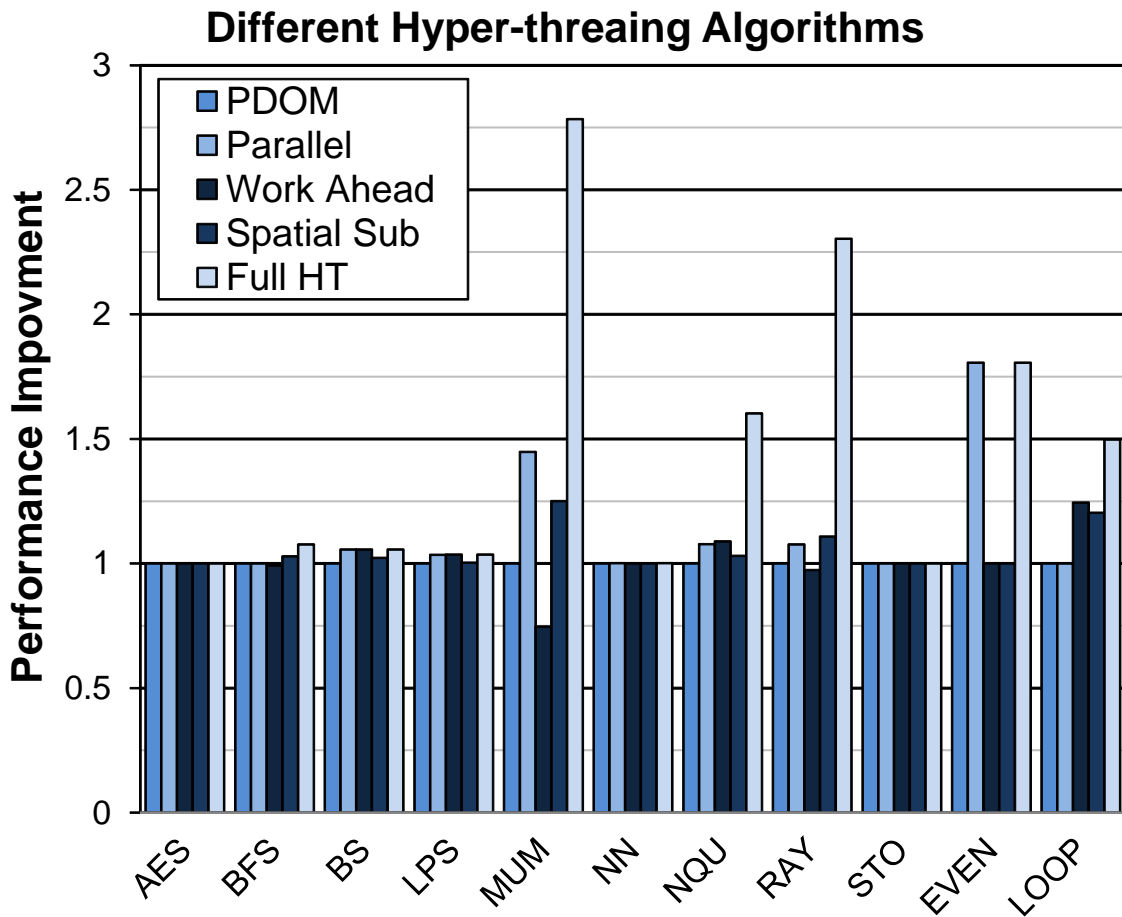


Figure 8.2: 2-way hyper-threading results.

offer due to the different run-times of threads allowing for warps to finish sooner and allow other threads to run.

Increasing the number of different instructions being executed places additional burdens on the memory system. Figure 8.5 shows the performance using the current SIMT memory system without any additional support for an increase in memory system requests. Either creating smaller warps or using hyper-threading to improve processor utilization results in additional memory system requests that must be addressed for future improvements.

8.2 Dynamic Micro-Kernel Results

To reduce simulation time, only the first 300k cycles were simulated at a resolution of 256x256. Simulation past 300k clock cycles results in similar behavior to the 150k to 300k range.

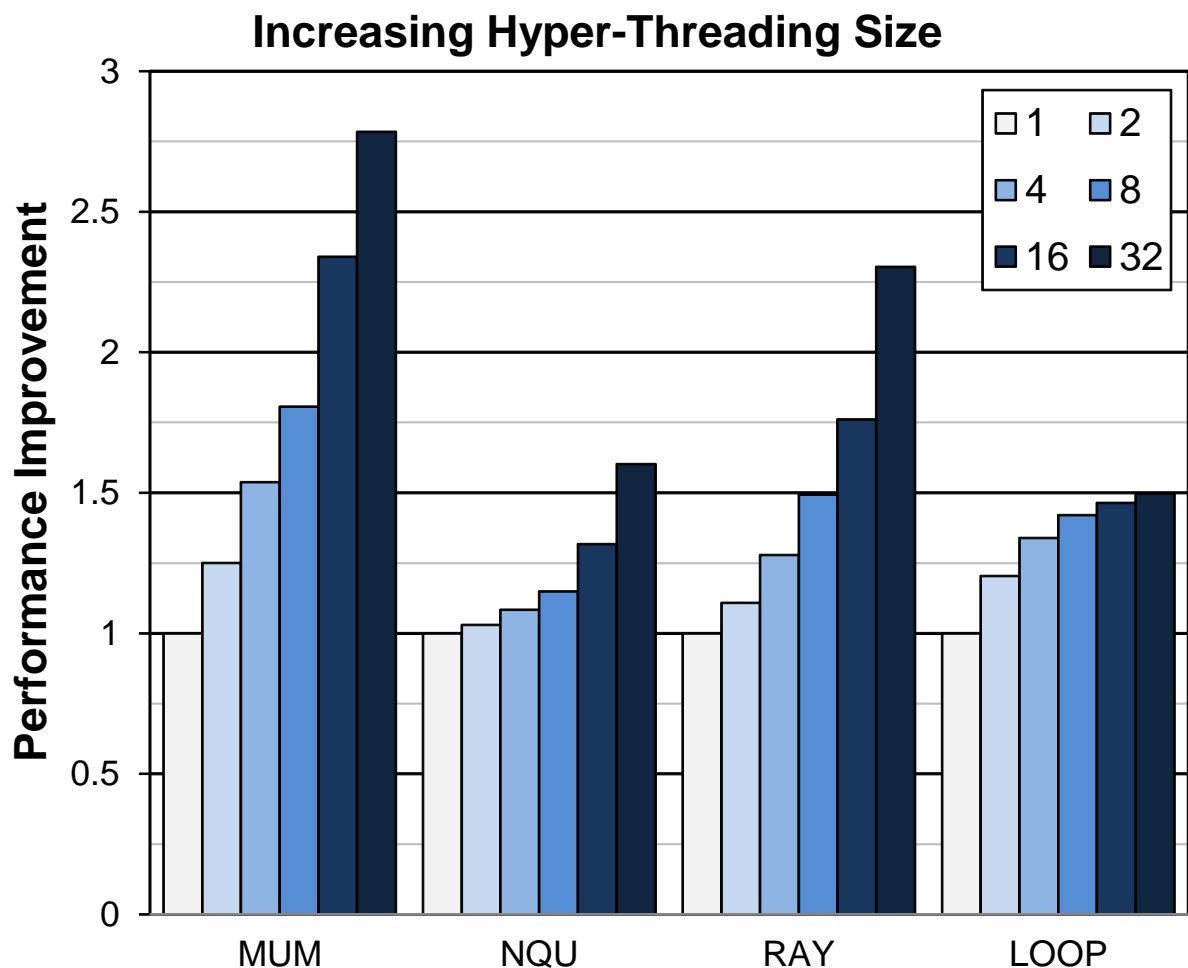


Figure 8.3: Effects on performance with increasing hyper-threading virtual processor count.

The first 300k cycles were simulated to demonstrate the sharp drop in processor efficiency at the start of the application. Performance numbers for MIMD and PDOM were generated by running the original Radius-CUDA application on our simulator.

We start our analysis with the assumption of no bank conflicts for the spawn memory space. This assumption allows for simulation of future programming models or compiler optimization designed to eliminate a majority of the bank conflicts. Figure 8.6 shows the number of threads participating in all warps over time for the `conference` benchmark scene. The other two benchmarks have similar plots. Similar to Figure 2.3, this plot categorizes a warp into 10 different categories based on the number of threads in a warps that are not idle. The SIMT

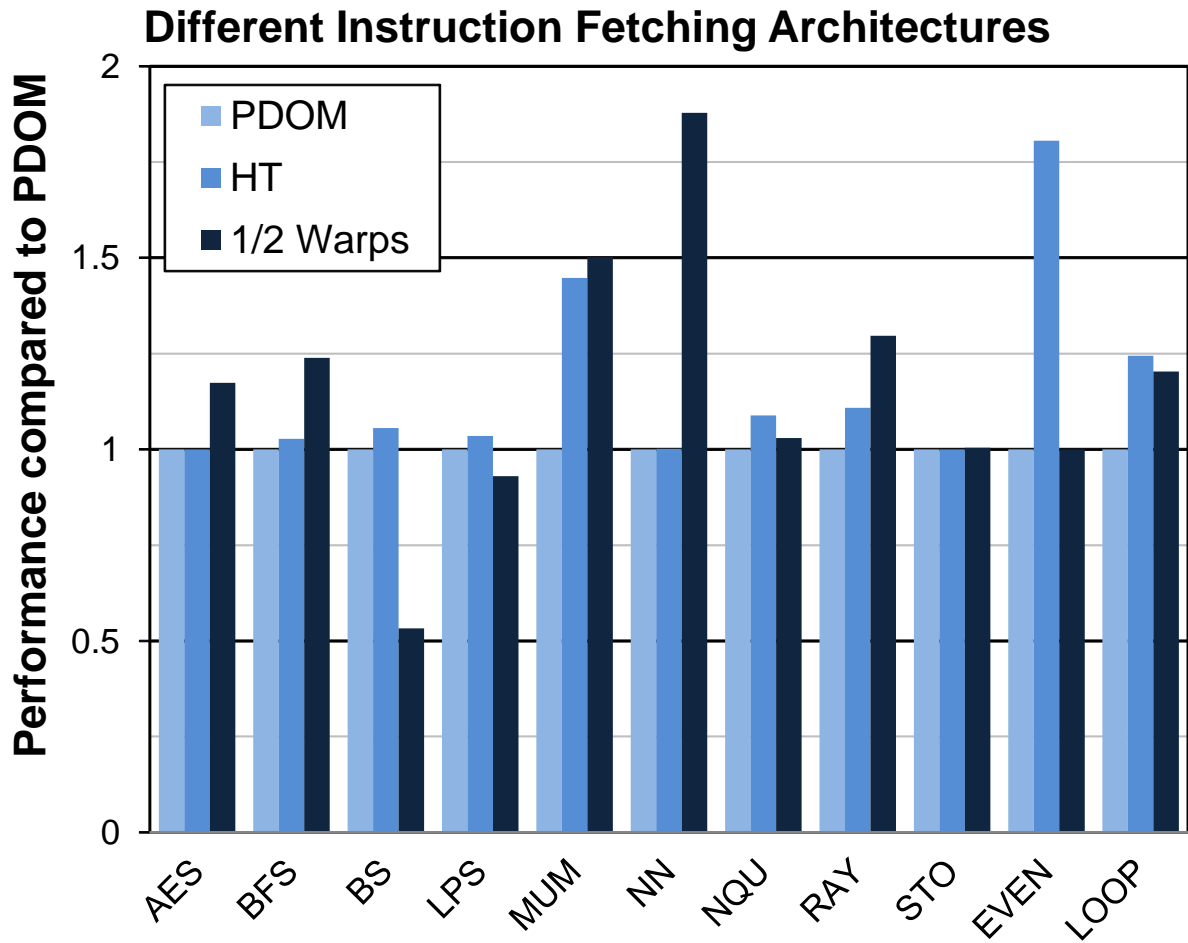


Figure 8.4: Comparing performance results of using hyper-threading and smaller warp sizes.

processor efficiency can be directly correlated to this plot. Using dynamic thread creation allows for a much higher utilization of all the processors on a chip. The average instructions per cycle for the **conference** benchmark is 615, $1.9\times$ higher than current hardware's 326 instructions per cycle. Using dynamic thread creation still results in some processors running idle due to branching within spawned functions and pipeline stalls from memory operations.

Since dynamic thread creation introduces additional instructions, comparing the instructions per cycle performance does not directly correlate to the rendering performance. Figure 8.7 shows the rate at which rays are processed for all benchmark scenes and both scheduling models. Block scheduling requires enough resources for the entire thread block before all threads in the block are executed. Warp scheduling will execute the maximum number of warps the

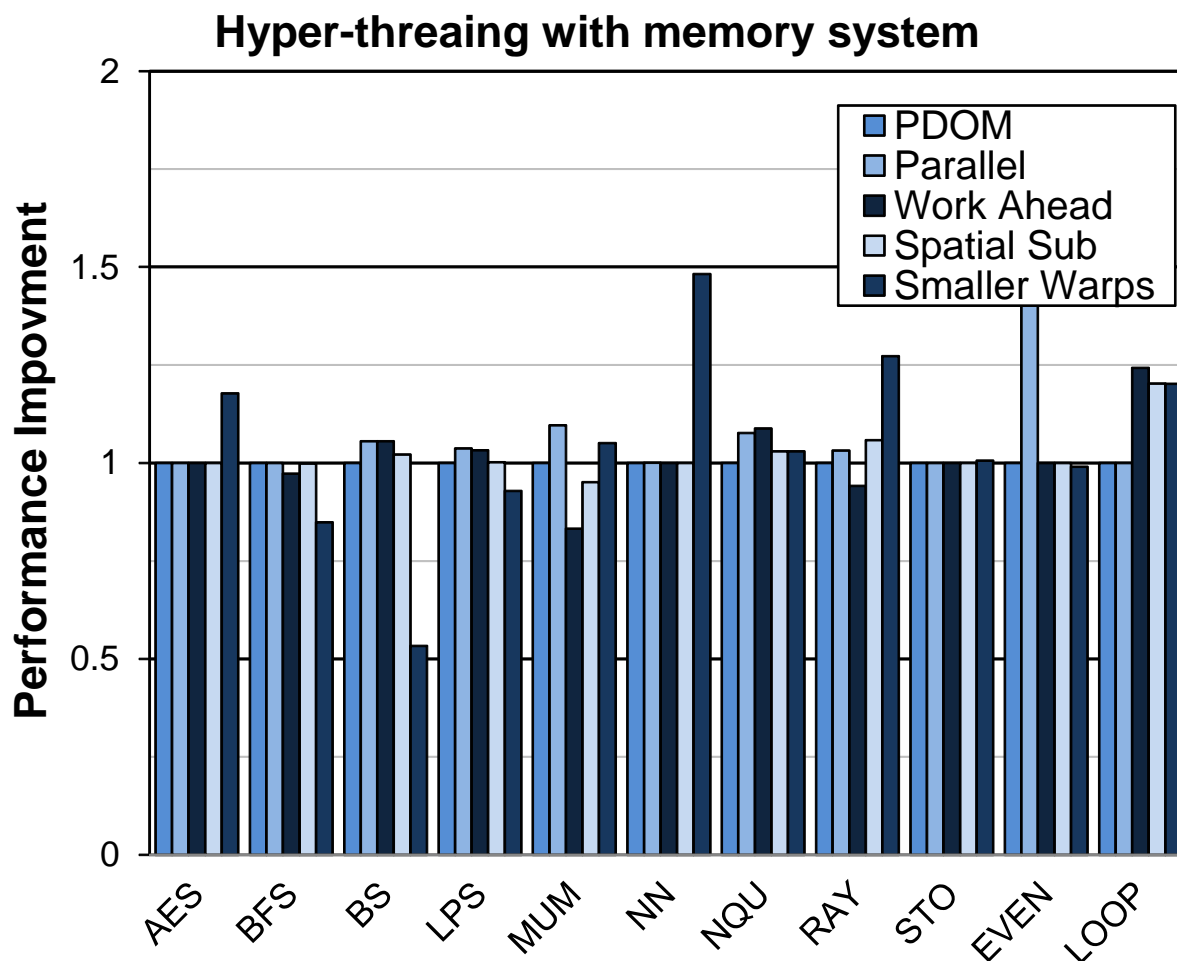


Figure 8.5: Hyper-threading results using existing SIMT memory systems

hardware can support, breaking up blocks if an entire block does not fit. PDOM Warp achieves a higher performance than the traditional hardware (PDOM Block) by allowing more threads to be scheduled to an SM and allows for more memory latency to be hidden. Our dynamic threads are able to achieve higher performances, due to the reduction of critical branching statements and improved memory coalescing.

The creation of dynamic threads requires additional memory operations. Table 8.1 shows the required bandwidth per image required to render a scene. Bandwidth values were computed from the number of down traversals and intersection tests required to render a single frame. The values are computed without any caching or separation between off-chip and on-chip

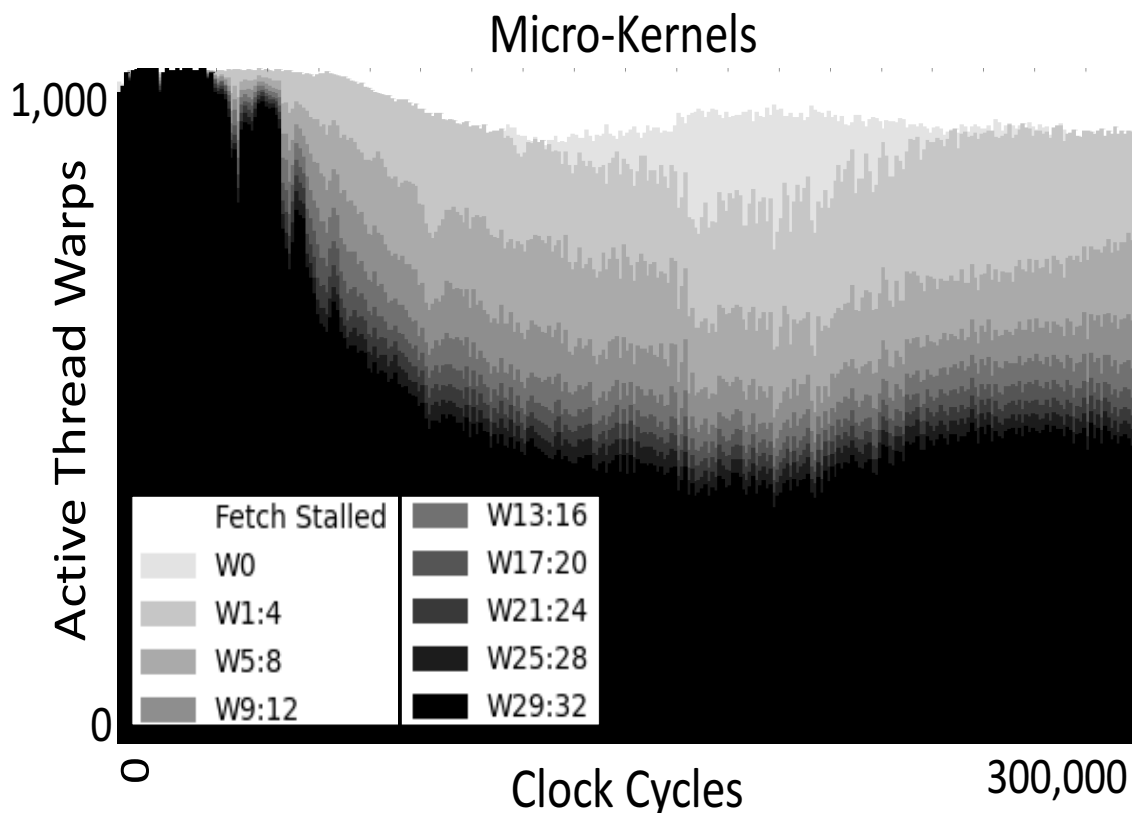


Figure 8.6: Divergence breakdown for warps using μ -kernels for the `conference` benchmark. Warps are able to keep more threads active by creating new warps at critical branching points.

memory spaces. The memory bandwidth required for dynamic thread creation is the difference between the traditional and dynamic values. The resulting overhead from creating dynamic threads results in an average bandwidth increase of $4.4\times$ for reading data. The total increase in bandwidth for reading and writing is an average of $7.3\times$.

Bank conflicts do affect our performance results by introducing additional memory latency. Figure 8.8 shows the active threads within a warp simulated with bank conflicts for the spawned memory space. An increase in pipeline stalls is introduced with bank conflicts due to serialization of all conflicting bank memory operations to the spawn memory space. While the number of pipeline stalls has increased, processor efficiency is still superior to traditional branching methods and maintains an average instructions per cycle of 429, $1.3\times$ higher than current hardware.

Theoretical branching performances is shown in Figure 8.9 for the `conference` benchmark

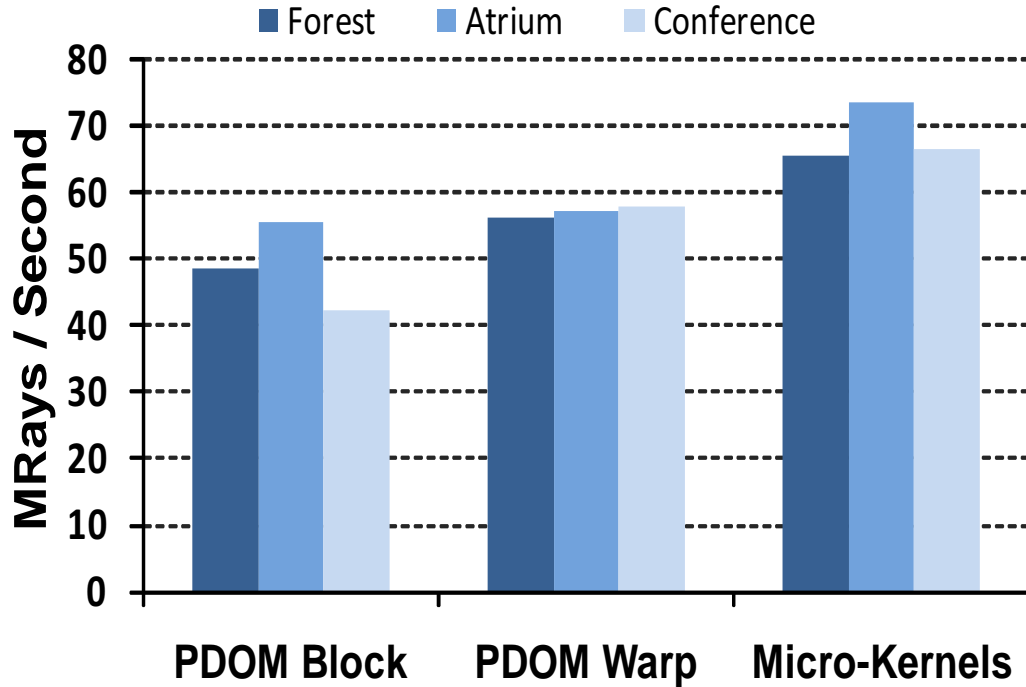


Figure 8.7: Performance results for all benchmarks using different branching and scheduling methods.

scene. Ideal memory systems were simulated (no memory latency) to determine the algorithm's theoretical performance. PDOM has no performance increase when simulated with an ideal memory system indicating its performance is limited by the branching hardware. Dynamic μ -kernel execution improves performance up to 45% of the MIMD Theoretical with the potential to achieve 60% of the MIMD Theoretical.

8.3 CUDA Supported Micro-kernels

We first evaluate the performance of our approach for different percentage of threads diverging in a warp. In this experiment 32 warps are evaluated and all warps result in diverging threads. The speed up results are shown in the Figure 8.10. On the horizontal axis we vary branch ratio, defined as the ratio of code size on the true path to that of false path. We vary ratio from 1 to 100. The speed up, defined as the number of cycles taken to execute the benchmark using our selection based approach to stack based reconvergence approach is always greater than 1.0. The best speed up ($6.1\times$) is achieved when more threads in a warp are taking

Benchmark	Reading	Writing	Total
fairyforest Traditional	62.1 MB	0.25 MB	62.35 MB
fairyforest Dynamic	296.5 MB	203.7 MB	500.2 MB
atrium Traditional	88.5 MB	0.25 MB	88.75 MB
atrium Dynamic	372.9 MB	258.1 MB	631.0 MB
conference Traditional	64.2 MB	0.25 MB	64.45 MB
conference Dynamic	263.3 MB	179.6 MB	442.9 MB

Table 8.1: Memory bandwidth requirements for drawing a single image without caching. Values are calculated from the number of tree traversal operations and intersection tests.

the shorter path, and speed up decreases when more threads are taking the longer path but always remains better than stack based control flow. As the fewer number of threads take the branch with longer path, our approach executes much fewer instructions by regrouping those threads into complete warps, but stack based control flow approach the longer path for each warp individually. But, with more percentage of threads branching towards longer path makes our approach equal or perform better than reconvergence based method.

We also evaluate the performance benefits for varying percentage of warps diverging in a block as a function of branch ratio. The speed up results are shown in the Figure 8.11 with 50% divergent threads in a warp. The best theoretical speed up of $2\times$ is achieved when all warps in a block are divergent. With fewer warps diverging our approach cannot form any newer efficient warps and speed up decreases. With a very few number of warps diverging the PDOM approach starts to perform better due to the small overhead introduced in the hardware to evaluate the performance model runtime.

To emphasize the importance of reconverging threads, simulations were performed without using the *reconverge* instructions. Figure 8.12 shows the performance as the number of instructions after the PDOM point increases. For the same branch ratio, performance decreases as the number of instructions after PDOM increase. Since Dynamic Micro-Kernels are not able to reconverge without the *reconverge* instruction, kernels created for improving the branch divergent code start to decrease performance gain after the reconvergene point. As the number of instructions increase, the higher the probability that PDOM will be selected as can be seen in Figure 8.12 at 3.5 Ratio with a low branch ratio.

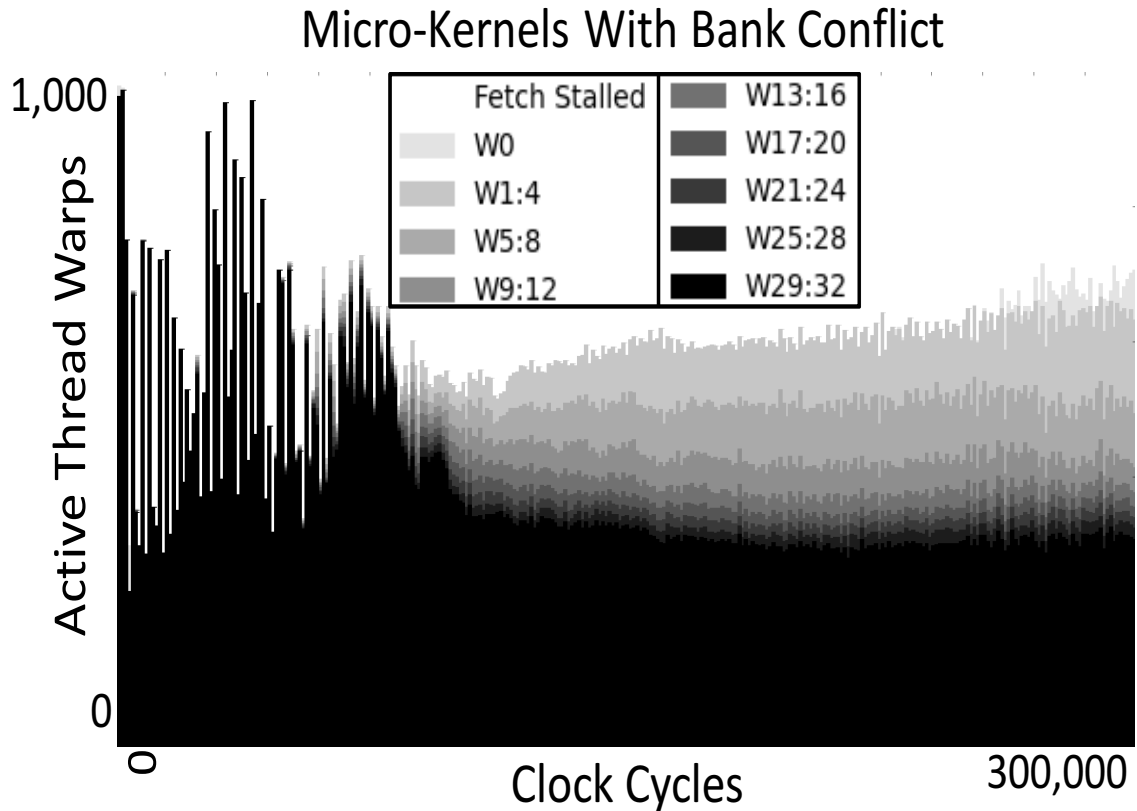


Figure 8.8: Divergence breakdown for warps using dynamic thread creation with bank conflicts for the `conference` benchmark. Warps still maintain more active threads over traditional branching methods, however, additional pipeline stalls are introduced by bank conflicts.

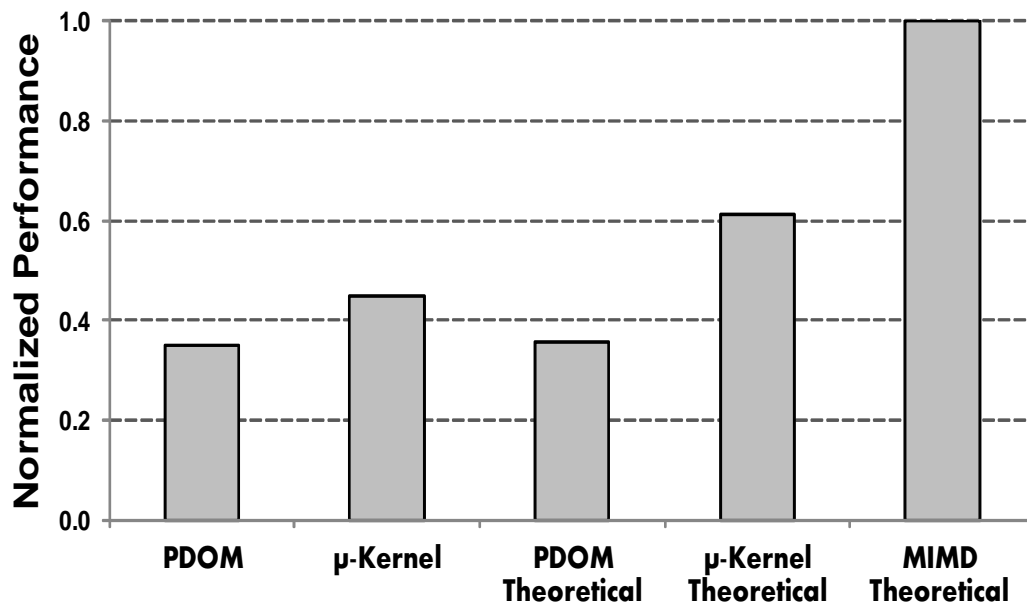


Figure 8.9: Branching performance for the `conference` benchmark. Theoretical results were simulated with an ideal memory system.

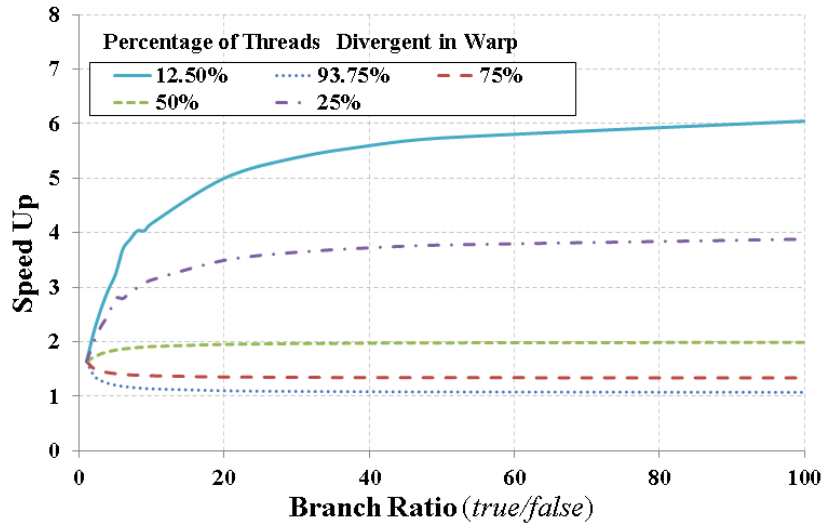


Figure 8.10: Performance for all warps diverging with different percentage of threads going to the two different branches. As we have more threads executing the shorter of the path, we expect higher performance since our method will perform fewer instructions.

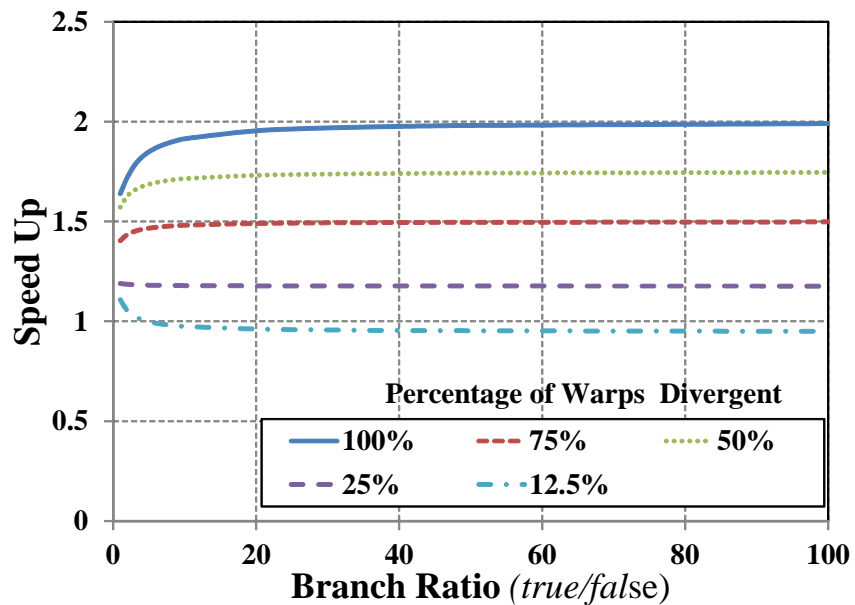


Figure 8.11: Performance for 50% of threads diverging in a warp with different percentage of warps diverging in a block. As we go to lower number of warps diverging we have fewer threads to form new warps, PDOM performance starts to be the better branching method.

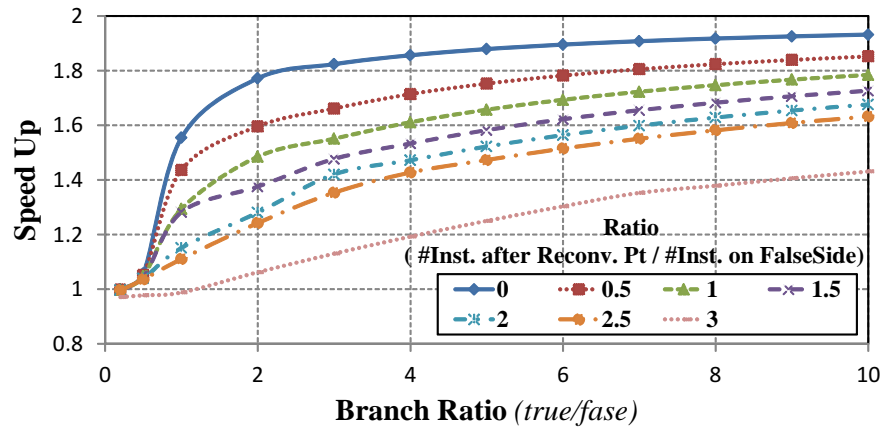


Figure 8.12: Performance results for 50% of threads diverging in a warp with different number of instructions after reconvergence point. As the number of instructions after reconvergence increases above three times the branching instructions, PDOM becomes more efficient when *reconverge* instructions are not used.

CHAPTER 9. Future Work

9.0.1 Spawn Memory Caching

Using off-chip memory for spawn memory frees up all of the on-chip memory for the CUDA application. To reduce the performance loss from using off-chip memory for the spawn memory space, a memory cache specifically designed for the spawn memory could be researched. The spawn memory is used for creating a thread in two ways. One area is used for storing the threads' register state and the other is used during warp formation processes. The cache will have to be designed to account for the two different usages while still maintaining a high hit rate. In addition pre-fetch logic, can also be implemented to improve cache hit rates for restoring thread memory state.

9.0.2 Register and Shared Memory Utilization

There are three limiting factors that are used to compute the maximum number of threads that can be executed on a SM: the maximum number of hardware supported threads, the number of registers and the amount of shared memory. Typically, once one of these resources is used up (not enough resources available for an entire block), no more threads can be scheduled for that SM. There can be exceptions where an application could be written such that all three resources are depleted at the same time, but this is very unlikely to happen. As a result, at least one of the memory spaces (registers or shared memory) will not be fully utilized by the application. Future research could seek to see if dynamic thread spawning use the remaining free memory resources to reduce the required off-chip memory storage and bandwidth.

To take advantage of un-used registers, the process of spawning instructions could leave the required data in the register file instead of saving the register states to spawn memory. During

the processes of warp formation, care could be taken to guarantee that threads are scheduled on the same SP such that it can re-use its original registers. Additionally, new instructions could be added to allow SPs to quickly pass register data between other SPs allowing for threads to be assigned to different SPs and then pass the required data in the register file between all SPs.

New warps can continue to be scheduled in this method until there are no more free registers. When all the registers' space has been consumed, registers used for spawn threads can be moved to spawn memory to make room for the new warp. Saving registers to spawn memory could then be done by spawning a micro-kernel that is designed to save register states to spawn memory. The hardware scheduler will add this micro-kernel to the scheduled warps. Once this warp has finished executing (saving all registers to spawn memory), the new warp can be scheduled.

The use of shared memory can be done in a similar method. When registers are being saved to spawn memory, they can first be saved into shared memory that is not used by the applications. When shared memory is filled, the register states will then be saved to off-chip memory (through the cache). When additional shared memory is required by new warps, another micro-kernel designed to flush shared memory can be executed in a similar fashion as the one for clearing registers. The micro-kernel will then save the register states that were saved in shared memory to off-chip memory.

This method requires a more complex scheduling hardware to keep track of all the different memory resources required by each thread and the location where each thread data is stored. To help alleviate the additional hardware functionality, additional software micro-kernels are introduced. Hardware assisted micro-kernels are used for moving thread register states in memory and can be generated by a compiler.

9.1 Programmable Thread Grouping

Diverging control flow is not the only characteristic that affects processor performance. Un-coalesced memory accesses can also decrease efficiency. Un-coalesced memory accesses are where two or more threads in a warp perform a memory operation with addresses that are outside of a single memory hardware block range. Memory accesses are then sequentialized

resulting in multiple memory accesses for a single warp. When performing warp formation, processor performance could be improved if threads could also be grouped based on future memory operations. The exact variable values and number of variables that could be used for helping form warps will be application-specific and the logic to organize threads can vary. To allow for application developers to optimize how warps are formed, a sorting algorithm can be executed that will order all possible threads awaiting to be placed into warps. The sorting algorithm will be another micro-kernel, that developers can write, that is executed when the processors are about to run idle due to a lack of work. Waiting until the processor runs idle allows for a larger number of threads waiting to be placed into warps. Unlike previous micro-kernels used to alleviate the hardware complexity, this micro-kernel will have to be developed by the application developer.

With this approach, the overhead of running a sorting algorithm could negate the long term benefits of more efficient warp formation. Alternative implementations could use fixed hardware to sort a specific number of variable options and logic sorting operations.

9.2 Programming Model for Spawning Threads

Previous sections also introduce new functionality that is not currently supported in modern day SIMT programming models. Features such as thread creation and programmable warp formation can be performed at the PTX level; this however does not integrate nicely with industry recommendations of using a high level programming model. Development of a programming model, that is similar to CUDA but includes additional functionality, would also increase the feasibility of this work.

CHAPTER 10. Conclusions

SIMT branching performance has been shown to have significant performance impacts for certain parallel applications. Initial targeted applications for SIMT architectures contained large amounts of parallel threads requiring the same execution kernel with basic control flow structures (such as rasterization shading applications). Since these types of applications don't require large amounts of branching instructions, sequential execution of the control paths did not result in significant performance loss. Today as SIMT processors have continued to advance in functional capabilities and performance, more complex applications consisting of more intricate control flow structures are being implemented on SIMT processors. While not all applications developed for SIMT processors are composed of complex control flow, both cases use the same branching behavior originally developed for simple graphics shading algorithms.

This thesis proposed using a hardware-software integrated solution for branching that allows developers the flexibility to choose the branching method used. Two new branching methods were introduced for SIMT architectures that are controlled through software instructions. By allowing the branching method to be controlled through software, developers can pick between the two new branching methods or the existing one. Each branching method has different performance overheads and potentially different effects on the processor efficiency. Developers creating simple shading applications can use the existing branching method that has the smallest amount of overhead and since the application are typically short, little performance loss will occur with this branching behavior. Alternatively, developers creating large complex control flow applications may choose one of the new proposed branching methods, that have more overhead setting up the branching instruction, but will see improved efficiency over PDOM branching when continuing past the branching location.

The first presented branching method is SIMT hyper-threading that allows for multiple

control paths from a single warp to execute in parallel. To implement this architecture, the warp meta-data was extended to include additional fields and the instruction fetch unit requires additional ports for fetching multiple instructions per cycle. To determine which instructions to fetch for a warp and also manage the case where thread divergence is larger than the supported hyper-threading architecture, the per-warp stack is expanded to include additional fields for supporting hyper-threading. Three generic algorithms are also presented that use hyper-threading to improve performance for different types of branching conditions. Performance increase is determined by the applications control flow structure. Applications composed of diverging branches see a performance improvement. Example benchmark applications observed performance improvements using 2-way hyper-threading close to $1.5x$ and diverging code sections can have close to $2.0x$ performance improvement.

The second proposed branching method is done by creating threads during runtime to execute μ -kernels. In our scheme, new threads are created to replace branching statements that cause low processor efficiency. Also presented is a hardware architecture for creating new threads and grouping similar control flow threads into new warps. The scheduler hardware allows for scheduling of new warps when enough processor resources are available. By replacing critical branch statements with dynamic thread creation, we are able to increase the performance of a ray-tracing application an average of $1.4x$.

APPENDIX A. Hardware Accelerated Ray Tracing Data Structure

The increase in graphics card performance and processor core count has allowed significant performance acceleration for ray tracing applications. Future graphics architectures are expected to continue increasing the number of processor cores, further improving performance by exploiting data parallelism. However, current ray tracing implementations are based on recursive searches which involve multiple memory reads. Consequently, software implementations are used without any dedicated hardware acceleration. In this work, we introduce a ray tracing method designed around hierarchical space subdivision schemes that reduces memory operations. In addition, parts of this traversal method can be performed in fixed hardware running in parallel with programmable graphics processors.

We used a custom performance simulator that uses our traversal method, based on a *kd*-tree, to compare against a conventional *kd*-tree. The system memory requirements and system memory reads are analyzed in detail for both acceleration structures. We simulated eight benchmark scenes and show a reduction in the number of memory reads of up to 70 percent compared to current recursive methods for scenes with over 100,000 polygons.

Introduction

Ray tracing algorithm performance has improved due to the reduced time spent processing individual rays as well as the abundant parallelism that modern hardware has enabled. One variable that has led to the increased efficiency of ray computations has been the use of custom data structures. Data structures define how geometric data are stored in memory and determine what elements should be tested for a given ray. A data structure that reduces the number of computations and memory reads while providing relative geometry elements for intersection

testing will likely result in an overall performance improvement. Since all ray types (visible, shadow, reflection, etc) require intersection tests, a common data structure can be used for all rays. Current data structures determine geometry elements by traversal, which is a process of stepping through hierarchical layers. This results in a recursive search processes and multiple memory reads. In this work, we propose a method for reducing the number of recursive steps by starting traversal of the tree data structure farther down the tree rather than starting at the top.

Today's graphics hardware supports large numbers of multiple cores for data parallelism in graphics rendering. To further increase the amount of computational parallelism, our approach calls for parts of the traversal to run on additional fixed hardware, freeing up processor time for other computations. User programmability is offered by performing intersection tests on the graphics processor, enabling user-defined code to interact with system memory and setting up the format of the data structure in memory.

We use a custom performance simulator for comparing our data structure implementing a *kd*-tree and a conventional *kd*-tree with a stack [29]. The performance simulator reveals the total system memory required for all implementations and the resulting memory reads. Our experimental results using this simulator show a significant reduction in memory reads for scenes with 100,000 or more polygons.

Previous Work

To accelerate rendering time, a variety of rendering methods [71] [19] [24] use accelerated data structures and parallelism offered by hardware. In the past, data structures and the hardware have been developed separately; however in the near future it is expected that graphics hardware is expected to move away from z-buffer rendering and directly incorporate scene data structures for ray tracing [39].

Accelerated Data Structures

Hierarchical Space Subdivision Schemes (HS3) like *kd*-tree [29] and Bounding Volume Hierarchies (BVHs) [64] have been the common data structures used for ray tracing implementa-

tions. Their popularity comes from their ability to adapt to the geometry of a scene allowing for efficient ray triangle intersection tests. Current-day implementations of HS3 have focused on reducing the number of memory operations and maintaining ray coherency for specific hardware platforms.

Today's *kd*-tree data structures for GPUs have improved the runtime performance for traversing the data structure and are achieving faster rendering time than CPUs [54] [30]. The use of HS3 does result in a recursive traversal method requiring frequent memory reads. Ray coherency [69] has been used to group similar rays together to improve the locality of memory reads for cache optimization. Still, a significant time is spent in the recursive search of these data structures. Quad-BVH [17] utilizes processor vector-width to convert binary trees into quad based trees, reducing the size of the tree and the number of traversal steps.

Uniform grid data structures [70] are commonly used because of their ability to quickly step to neighboring grid points. Grid data structures do not require any recursive search method, however, they cannot adapt to the scene geometry and can result in multiple iterations before performing relative ray triangle intersection tests. Because of its non recursive traversal, uniform grid data structures outperform HS3 data structures for scenes that are uniformly distributed. Because most scenes are not uniformly distributed, uniform grid data structures are not commonly used.

Ray Tracing Hardware

Research in graphic architectures has resulted in fixed function [26] [20] and programmable multi-core designs [66] for accelerating ray tracing rendering. Fixed function hardware such as SaarCore [59] [60] and RPU[73] [72] implement a fully defined rendering pipeline for ray tracing. Both designs have been fully implemented on FPGA and an ASIC model was developed for RPU. Dedicated hardware for traversing tree data structures is included in both designs, but recursion and multiple memory reads are not eliminated. Additionally, legacy rendering methods such as rasterization and others using the current programmability of modern pipelines are not supported.

Other architectures for ray tracing introduce a high number of general multi-core processors

and memory systems designed for graphic computations. Intel's Larrabee [63] is composed of several in-order x86 SIMD cores that can be programmed to support any graphics pipeline in software. Acceleration comes from running large amounts of graphic computations in parallel and running multiple threads on each processor to reduce the latency for memory operations. CUDA [48] and Copernicus [25] also offer large numbers of cores and can hide memory latency by having large numbers of threads with no switching penalty. Ray tracing implementations on these architectures is accomplished through software kernels that then run on the processors. Direct hardware acceleration is supported for several graphic computations, but none for data structure traversal.

Group Uniform Grid

Overview

To reduce the number of recursive steps required by HS3 to processes any ray type (visible, shadow, reflection, etc), we propose implementing an additional data structure called Group Uniform Grid (GrUG) over the HS3 data structure. GrUG makes rays bypass parts of the tree structure allowing for traversal to begin closer to its final leaf node. GrUG is an axis-aligned subdivision of space consisting of only two hierarchical layers. The top layer is a uniform grid data structure that divides the scene into grid cells. The lower layer consists of groups of top layer cells and corresponds to nodes of the HS3 tree structure. Figure A.1 shows a 2-D example of the two layers and a *kd*-tree.

To traverse this data structure, only the mapping between the two layers and mapping between each ray and its grid need to be addressed. Once this mapping is complete, the HS3 structure can be traversed starting at the node identified by GrUG. Mappings between layers and ray is performed using a hash lookup table. The value of the hash function of a ray origin equals the top layer cell ID corresponding to the ray. By this method, a collision produced by a hash function indicates that the rays are in the same cell. The hash value is then used in a table lookup to determine the memory address of the HS3 node data structure. Figure A.2 shows the entire hashing process starting with integer X,Y and Z coordinates.

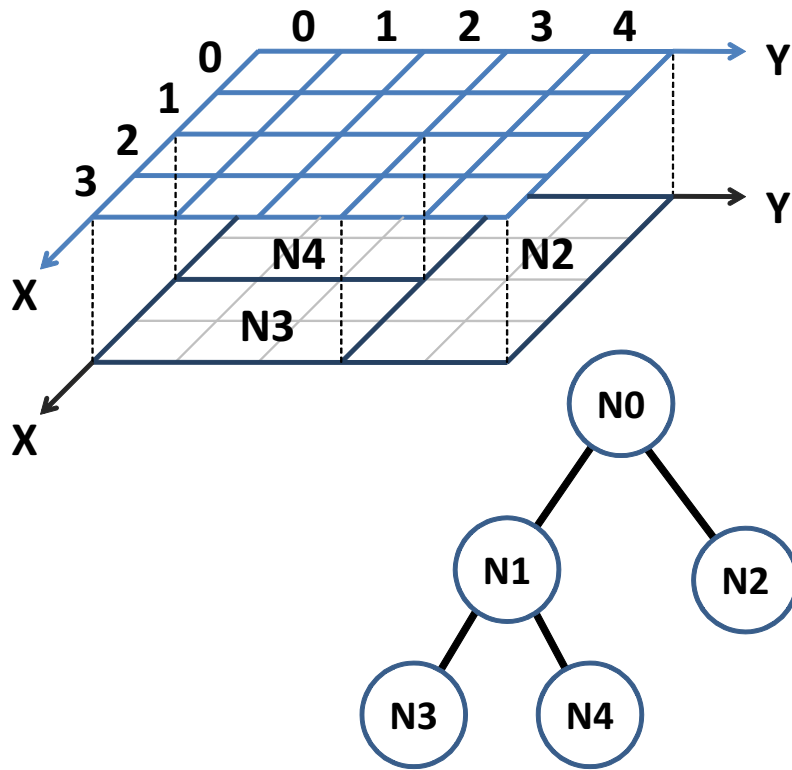


Figure A.1: GrUG data structure composed of a uniform grid for the top layer and a lower layer that maps to a HS3 tree structure.

Using GrUG to reduce recursion and memory reads results in the top data structure resembling a uniform grid. Rays that require additional traversal outside of its original group bounding tree cannot continue using HS3 methods, but require traversing the uniform grid. Stepping to the next grouping can be done by a 3-D Digital Differential Analyzer (DDA) [3]. The DDA allows quick stepping between cells because they are on a uniform grid pattern. Stepping to neighboring cells does not guarantee that the lower layer tree node has changed, so this process is repeated until a new tree node is found. This process can only be performed after the initial traversal and is not needed for intersection testing, and therefore can be run in parallel with triangle intersection tests.

To create a hash function with collisions resulting in rays being in the same top layer cell requires an integer format representation for ray origin. A uniform grid data structure maps nicely to integer values as each cell is assigned to an index value. With the DDA implemented on top of the uniform grid, integer arithmetic can be applied to the DDA. With proper setup,

the use of integer operations inside the traversal of GrUG still maintains floating point accuracy.

The use of dedicated hardware for GrUG traversal cannot stand on its own, but must be able to interface to the processing cores for intersection testing and traversal of the HS3 tree. While a complete interface is not presented for this implementation, a high level architecture model is shown for performing data structure traversal. This model relies on the software processor to run HS3 traversal, triangle intersection tests and shading while the GrUG hardware performs data structure traversal.

Hash Table Function

There are two parts to a hash table function: the hash function itself and the table memory. For our approach, the table memory is used to take a grid cell and map it to the tree node of the HS3 structure. HS3 nodes are stored in system memory, so a pointer to system memory is stored inside the hash table. Our table size is then equal to the total number of cells created by the uniform grid. For simplification, only powers of 2 are allowed for grid spacing. The number of bits needed to address the table memory is then $\text{Log}_2(\text{number of cells})$.

The hash function result is used as the address in the table memory and must produce a value between 0 and the $(\text{total number of cells} - 1)$. Furthermore, a hash function collision indicates that the two ray origins that produced the collision are in the same uniform grid cell. If the ray origins are represented in integer format, this is a simple operation of concatenating the most significant bits of each axis. The number of significant bits is determined by the grid spacing.

Creation

Creating a GrUG data structure requires setting up two memory spaces, the hash table and the HS3 tree. Since the hash table memory contains pointers to the HS3 tree, we setup the HS3 tree structure first and then populate the hash table memory with pointers to the appropriate HS3 tree node. GrUG requires only one constraint on building the HS3 creation algorithm. GrUG operates on a uniform grid structure, so splitting locations of the HS3 structure must align with these bounds until a mapping between GrUG and the node is defined. By forcing the

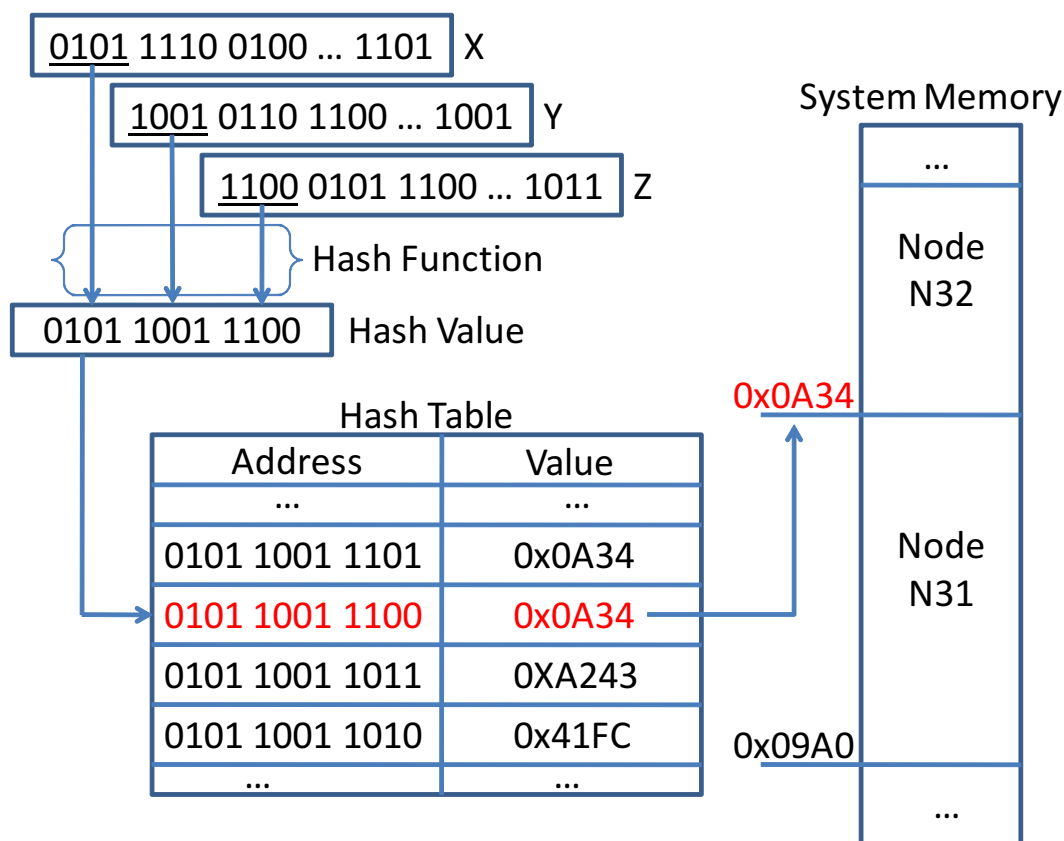


Figure A.2: Hash function starting with X,Y,Z coordinates in integer format and producing the data structure that contains geometry information for intersection testing.

HS3 tree structure to align with GrUG, leaf nodes generated with aligned splitting locations are identified as thresholds. Thresholds have spatial bounds corresponding to cells in the uniform grid and are the nodes that get mapped to GrUG. Threshold nodes can then continue being subdivided without any restraints on the splitting locations. The mapping of threshold nodes to GrUG requires that every uniform cell defined in the threshold bounding area gets populated with a pointer to that node. Once this is complete, every node above the threshold node can be removed from memory, resulting in several smaller trees. Each tree is mapped to the GrUG structure.

In addition to storing the resulting trees, bounding values must also be stored with the threshold node. The specific format for storing bounding values and HS3 forests are user-defined and requires the ray intersection code to interpret pointers to system memory generated by GrUG. This allows for grouping of geometric data with rendering parameters such as color

and texture coordinates in the same memory location. While the specific implementation is left to the user, two requirements must be met:

- Tree nodes and geometry data inside the scene must be present and accessible by only providing the memory address to the data structure. This allows traversal of GrUG to produce a single memory address and have the tree traversal and ray polygon intersection code be able to test all relative polygons in the cell area.
- Six bounding values of the grouped grid must be contained in the data structure. These values are used for computing the next grouping of ray intersects if no intersection is found in the current grouping. These values must also be accessible by providing the same pointer to system memory.

Stepping Between Neighbors

To step between uniform grid cells, a DDA method is used. While this method allows stepping between cells, this process repeats until reaching the boundary of the starting group. DDA steps one cell at a time and is a function of both the current cell and the direction of the ray. The absolute position of the ray inside the cell is only needed once for the entire traversal of the ray. Three parameters are needed per axis for stepping: *tmax*, *delta* and *step* (Figure A.3). The *tmax* value is the independent variable in the 3-D parametric line equation and is incremented when traversed along that axis. The *delta* value defines how much *tmax* gets incremented by and is the value needed to cross an entire cell. The final value is the *step* value that specifies what direction to step and is discussed in more detail in the next section. Stepping is then done along the axis that has the lowest *tmax* value since it is the closest to the edge of the cell. The *tmax* value is then incremented by the *delta* parameter to represent the new distance from the edge of the cell. This process is repeated until the cell index value is outside of the group. The boundaries of groups must be provided to the hardware before this process can begin and is stored in the data structure for the current group.

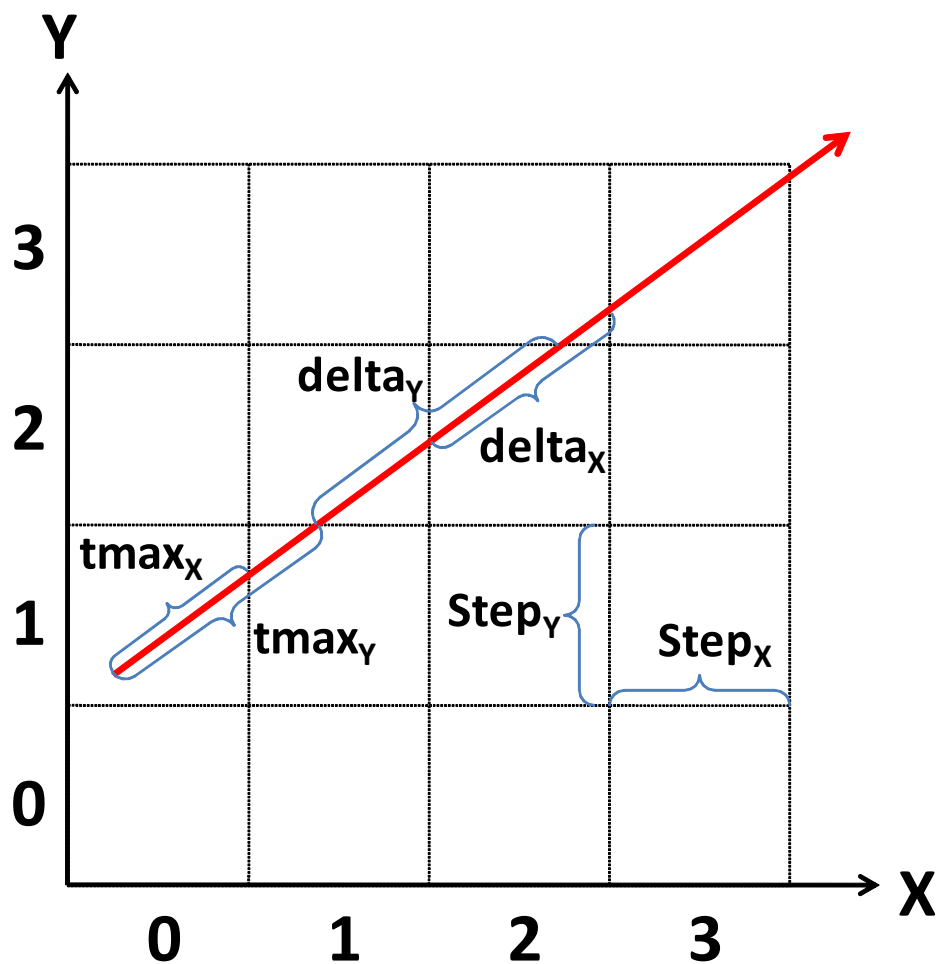


Figure A.3: Variables needed for DDA traversal.

Integer Operations

The GrUG hash function uses integer values to represent the entire grid spacing. Conversion of ray origin to integer values should utilize the entire integer range. A fixed size of N bits is used for representing the integer value and the grid resolution is specified during initialization, the number of integers that are inside of a cell is then equal to $\frac{2^N}{GridResolution}$. The index of each cell is then equal to the hash function for any points inside that cell. Pre-computed integer scaling values are computed from the maximum and minimum limits of the scene. Linear interpolation can determine the location inside the uniform grid. Once the integer value has been found, it is used throughout the rest of the traversal.

Since the location is now stored as an integer format, integer arithmetic is used for the DDA

traversal. Both $tmax$ and $delta$ are calculated in a similar fashion but the step parameter is then assigned a constant value of $\pm \frac{2^N}{GridResolution}$. This value is the number of integers in a cell. When traversing to a neighbor cell the integer location value must change by that amount to leave the current cell. The \pm determines the direction of the step and has the same sign as the ray direction.

Hardware Implementation

An architecture for the GrUG traversal process is presented in Figure A.4. For a multi-core implementation, each processor would require its own hardware implementation. The only exception is the table memory that would be shared among all the processors. This proposed architecture breaks down into two main areas, hash table lookup and traversal. Hash table lookup takes two inputs, one for new rays and the second for performing another traversal of the data structure. Since new rays only need to be initialized, returning rays can be sent directly to the hash function. The output of the hash table can then be passed directly to the processor for triangle intersections. The traversal hardware also requires two inputs for initializing new rays and performing further traversal operations. The output of traversal is then sent to a buffer that waits for the processor to indicate if the ray has finished or needs to continue traversing.

The architecture presented is a pipelined implementation and operation is performed in the following order for new rays corresponding to the numbers presented in Figure A.4:

1. New rays are passed into the architecture from the processor and are initialized in parallel for both the hash table function and the traversal using DDA.
2. The output of the hash initialization is then inputted into the hash function where the hash value of the ray is computed. The computation results in the ID for the cell that the ray belongs to. The traversal initialization results are also passed into the DDA traversal where it waits for further inputs.
3. The resulting hash value is then used as the address in the table memory. The resulting memory value is then the pointer address for a node/leaf in the HS3 tree.

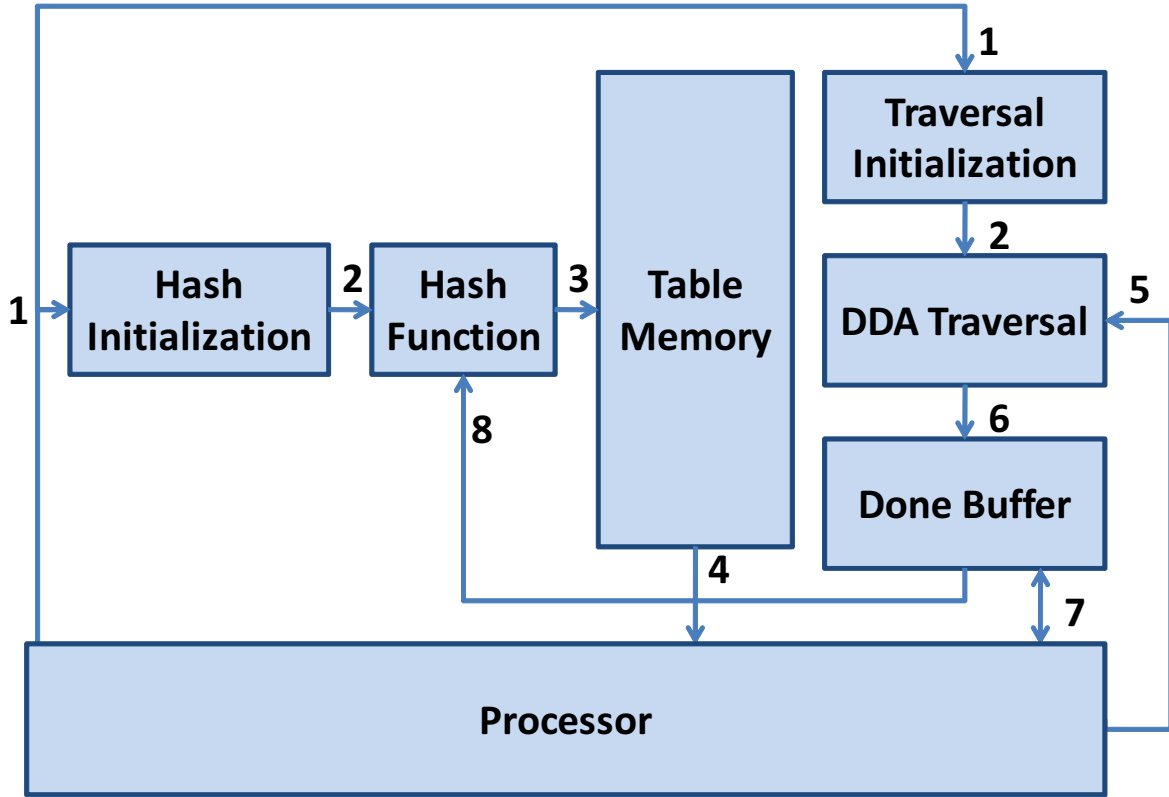


Figure A.4: Architecture of Grouped Uniform Grid

4. The processor then begins executing the HS3 traversal and intersection kernel and performs a memory read to get the bounding values of the tree node/leaf.
5. The processor outputs the bounding values to the DDA traversal to determine the next grouping the ray intersects. The processor then begins traversal of the HS3 and computing the intersection results in parallel with the DDA.
6. The DDA traversal finishes and outputs the results to the done buffer.
7. The processor finishes intersection tests and tells the done buffer of any intersection results.
8. The done buffer checks the intersection results and the DDA result to determine if the ray requires additional traversing. If the ray must continue traversal the new ray location is passed into the hash function and begins step 3 again. If the ray intersects a valid

polygone, it is removed from the pipeline.

While a software implementation of GrUG is feasible, A hardware implementation of our GrUG approach will have greater performance improvement by allowing additional operations to perform in parallel that would not be possible with a software implementation. In the hardware method the DDA traversal is able to run in parallel with triangle intersection, allowing for its computational cost to be hidden. In addition to operating in parallel, specific computations can be accelerated including the hash function and the operations performed for each axis. The hash function and axis operations require multiple instructions in software that can be performed by a single functional unit in hardware. This acceleration allows for a hardware implementation of GrUG to have a greater performance over an equivalent software implementation.

Pipeline Architecture

Fixed Hardware

The hardware pipeline stages for GrUG traversal are shown in Figure A.5. Memory-addressed registers are used for configuring initial parameters of the pipeline for the size of the grid resolution and are set at the start of the application. The first stage of the pipeline is ray projection. Rays requiring additional traversal operations (rays that did not find an intersection in an earlier GrUG traversal) need to be projected out of the original GrUG grouping. Rays are projected by the t_{min} value using 3 floating point multiply accumulators, one for each axis. This requires that when rays are finished with the user-defined traversal algorithm, t_{min} must be set to the t_{max} value, resulting in the ray being projected out of the current scene grouping. Next, the ray undergoes GrUG traversal using the hardware hash function. The result of the hash function is used to read the entry in the hash table that is stored in an on-chip cache. Cache misses stall the pipeline and fetch the value from global memory. The programmable processors are then used to read the bounding box of the GrUG groups that are stored in the tree data structure located in device memory and the t_{max} value is computed (see Algorithm 5).

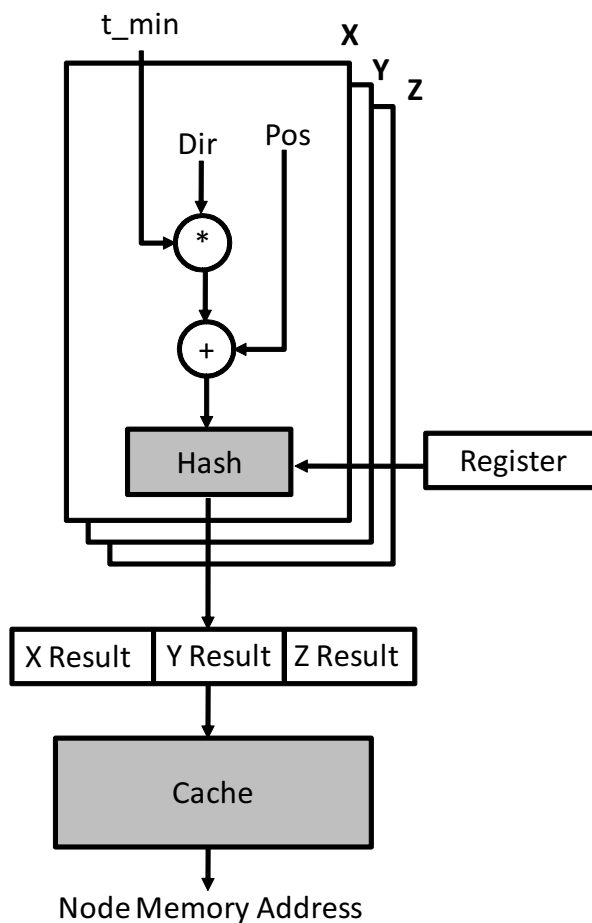


Figure A.5: Integration of the GrUG pipeline into a multi-core graphics processor and the fixed hardware stages for the GrUG pipeline.

The pipeline is designed for a large throughput of rays and allows for a ray to be outputted every clock cycle. The number of cycles corresponding to the individual stages are shown in Table A.1. To allow for an even higher throughput of rays, each pipeline stage can be vectorized, allowing multiple rays to be processed in parallel for each pipeline stage. This configuration is ideal for wide streaming processors requiring execution of different threads every clock cycle resulting in repeating instruction calls to the pipeline.

Hash Function

The hash function's responsibility is to determine in which GrUG uniform grid cell a ray belongs to. The resulting uniform grid cell ID is converted to a memory address that the GrUG

Algorithm 5 Ray Tracing with GrUG Pipeline

Require: *Thread ID*
Ensure: *pixel color*
 $ray \leftarrow \text{createRay}(\text{Thread ID})$
if ! $\text{intersectSceneBox}(ray)$ **then**
 return
end if
 $t_{min} \leftarrow \max(\text{intersectBoxEnter}(ray), 0)$
 $t_{absMax} \leftarrow \text{intersectBoxExit}(ray)$
while $t_{min} < t_{absMax}$ **do**
 $node \leftarrow \text{GrUGHardwareTrav}(ray, t_{min})$
 $t_{max} \leftarrow \text{computeMaxT}(node.BBox, ray)$
 $hit \leftarrow \text{userDefTrav}(ray, node, t_{min}, t_{max})$
 if ! hit **then**
 $t_{min} \leftarrow t_{max}$
 else
 break
 end if
end while
 $pixel\ color \leftarrow \text{shade}(ray)$

Pipeline Stage	Pipeline Depth	Throughput
Ray Projection	2	1
Hash Function	5	1

Table A.1: GrUG performance parameters for each fixed hardware pipeline stage.

software algorithm uses to determine the location of the root node for the user-defined traversal algorithm. The hash function hardware shown in Figure A.6 takes as input one single precision floating point value representing the ray location for one scene axis. To support all three axes, three hash function pipelines are used in parallel and the results are concatenated to form one cell ID. The output for each hash function pipeline is a 9 bit value, resulting in a maximum grid size of 512 x 512 x 512. Smaller grid sizes are supported by truncating the least significant bits for each pipeline output. The hash function hardware, composed of integer subtraction and shift registers, is compatible with all floating point values from 1.0 to -1.0 . Values outside of this range are not supported, requiring all scenes to be scaled to fit into a 1.0 to -1.0 sized box. Our hardware pipeline operates correctly for all floating point values between these limits except for two exceptions, values ranging from 0.5 to 0.503906 and -0.5 to -0.503906 . In these cases the fraction bits are zero and the exponential value results in hash value being equal to zero. At the same time, values close to ± 0.25 also result in a hash function of zero for similar

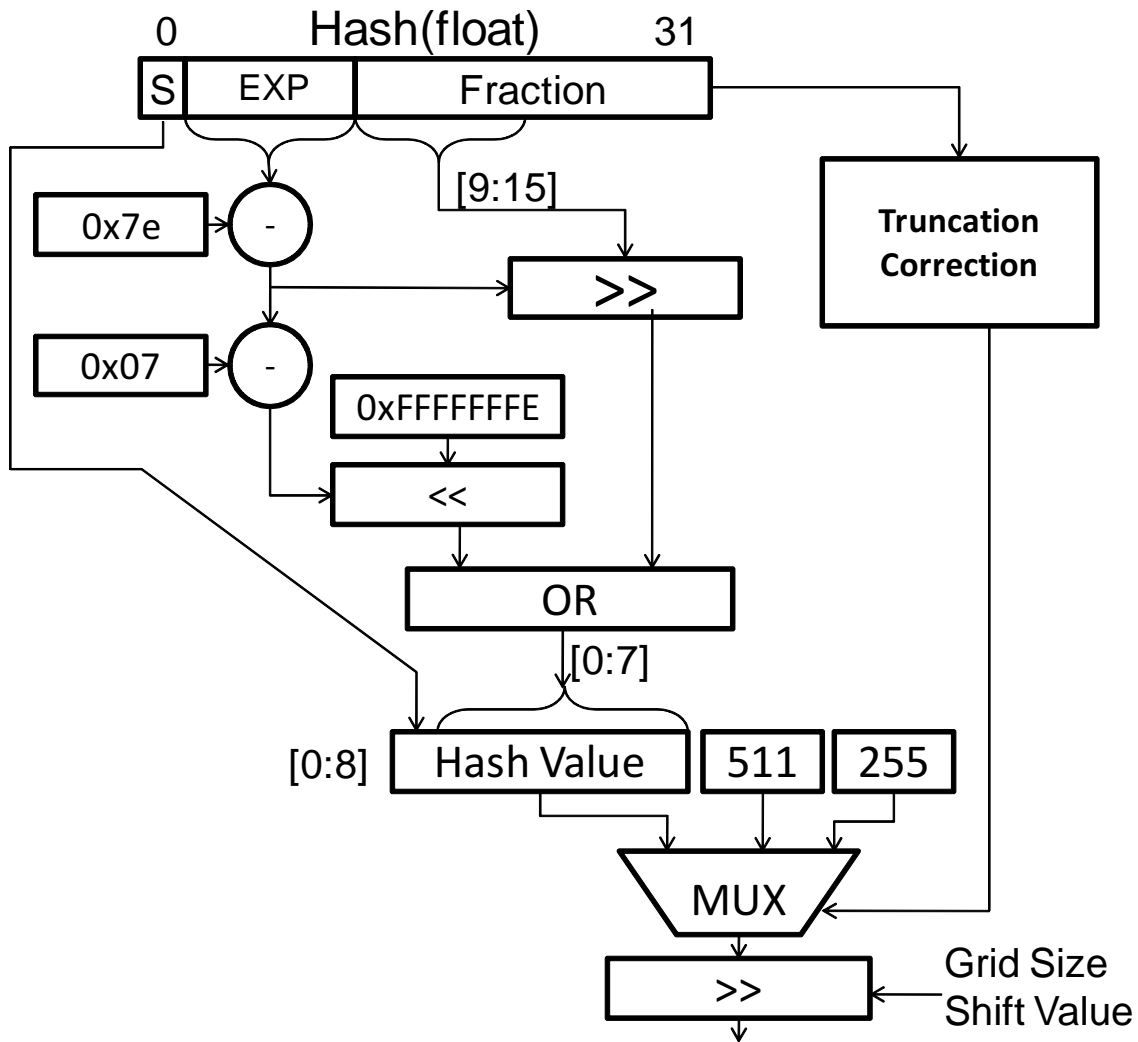


Figure A.6: Architecture of GrUG hash function for one axis using a 512 grid.

reasons. To correctly identify these ranges, truncation correction is used to output corrected values for the ± 0.5 range allowing for each uniform grid cell to have a unique hash value.

Performance Analysis

A total of eight different scenes, each with different polygon counts, were tested using our custom CUDA-based performance simulator. Table A.2 list the benchmarks with scene data and overall results.

Our performance simulator implements only the traversal of primary visible rays and does

Benchmarks	polygons	kd-tree Mem. (MB)	GrUG Memory (MB)			
			Tree	Boundary	Hash-table	Total
Ulm Box	492	0.01	0.01	0.01	512.00	512.02
Stanford Bunny	69451	1.34	0.87	1.41	512.00	514.28
Fairy Forest	172561	2.66	2.32	1.02	512.00	515.34
Cabin	217903	3.12	2.88	0.70	512.00	515.58
Atrium	559992	9.30	8.77	1.61	512.00	522.38
Conference	987522	8.54	8.23	0.93	512.00	521.17

Figure A.7: Benchmark data for each scene and the total memory required by the data structure.

not perform any rendering. The HS3 data structure implemented is a *kd*-tree used in RADIUS-CUDA [8]. To measure performance of GrUG, only system memory requirements and the number of memory reads are reported. All benchmark scenes were simulated at a resolution of 1920x1080, resulting in a total of 2,073,600 rays, and a GrUG grid size of 512x512x512.

Memory Utilization

Figure A.7 shows the total system memory needed for storing the different data structures used by GrUG. The use of a hash table in GrUG results in a significant overhead in memory requirements to store the entire hash table in memory. The required memory is based on the GrUG grid size, 4 bytes per grid cell. A fixed grid size of 512x512x512 will use 512MB for storing the hash table.

In addition to the hash-table, the *kd*-tree structure and bounding dimensions of all threshold nodes must be stored in system memory. Figure A.8 shows the memory requirements for both *kd*-tree and GrUG. GrUG uses less memory for storing the tree structure and bounding dimensions of threshold nodes since it does not need to store the entire tree structure, but only the tree nodes that are at and below the threshold node. The additional 24 bytes needed for storing the bounding dimensions, equivalent of 3 nodes, is smaller than the tree structure above threshold nodes.

Figure A.8 also shows that the memory space required for storing GrUG tree data sets

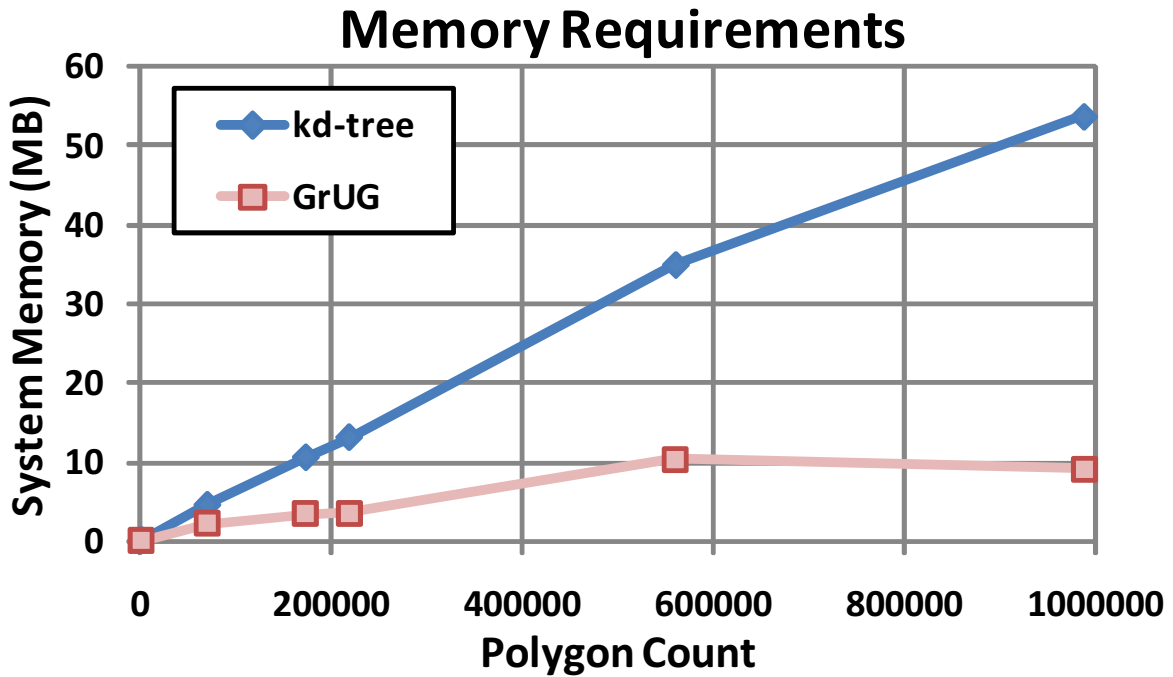


Figure A.8: The memory requirements of GrUG compared to *kd-tree*.

are not linear with respect to the number of polygons in the scene. *kd-tree* memory usage scales linearly with polygon count because the number of nodes created is a function of the tree depth, which is determined by the number of polygons in the model. Memory requirements for GrUG are influenced by the density of polygons in a scene. Polygon density is a function of the number of polygons in a scene and how evenly distributed they are in the entire scene. Scenes with higher polygon density will result in more threshold nodes, resulting in deeper trees and additional bounding dimensions being stored.

Memory Operations

The resulting memory reads for both GrUG and *kd-tree* are shown in Figure A.9. The number of memory reads are for traversal of GrUG and *kd-tree* data structure and do not include the memory operations needed for triangle intersection testing. Memory reads from triangle intersection testing are not reported because both implementations resulted in nearly identical intersection tests.

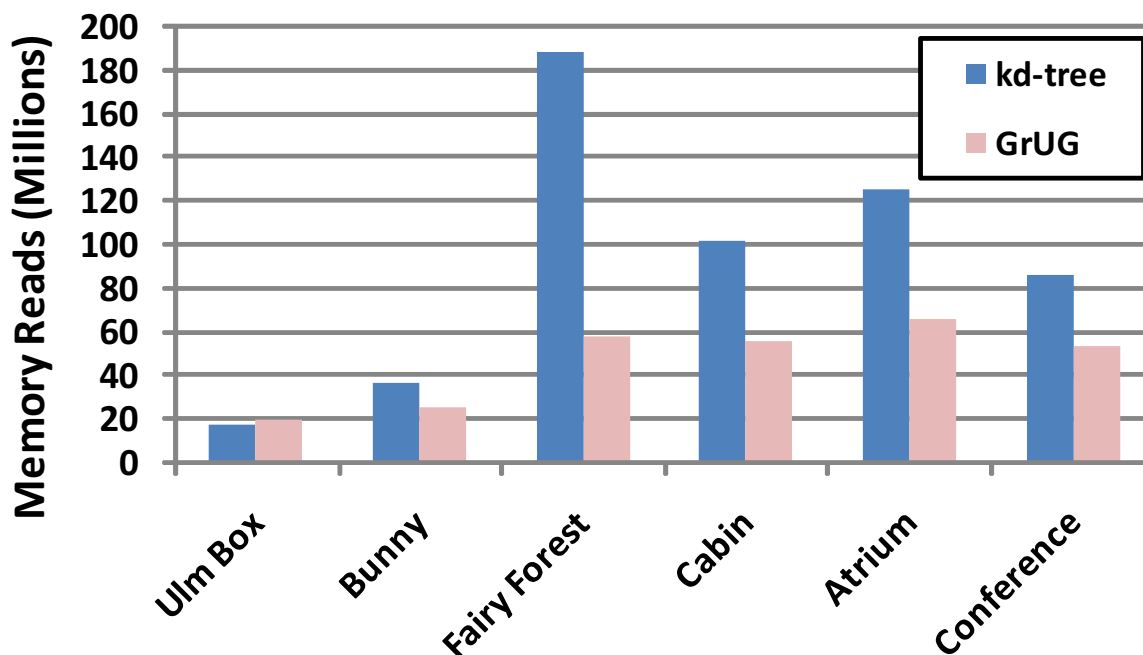


Figure A.9: Number of memory reads required for traversing GrUG and *kd*-tree.

All but one benchmark scene required fewer memory reads using GrUG. UlmBox resulted in higher memory reads due to having a small *kd*-tree size. While GrUG allowed for traversal to begin farther down the *kd*-tree, the resulting memory read operations for getting boundary data was larger than the savings of starting lower on the *kd*-tree. The remaining five benchmarks resulted in varying reduced memory reads. The number of reduced memory reads needed by GrUG is greatly dependent on the polygon density.

The overall performance results are shown in Figure A.10. By using our hardware pipeline we reduce the number of tree traversal steps by an average of 32.5X for visible rays. The overall speedup for traversal using GrUG is an average of 1.6X for visible rays, with a maximum of 2.74X for the Robots benchmark. This reduction in executed instructions is due to shallower tree data structures as shown in Table A.2. While the tree traversal executed instructions has drastically decreased, the GrUG software traversal kernel becomes the most time consuming part of the traversal algorithms. The GrUG software traversal kernel requires two back-to-back device memory reads. The first operation retrieves the root node tree data structure and

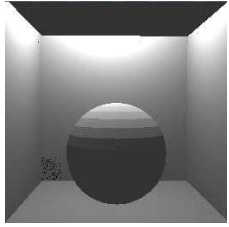
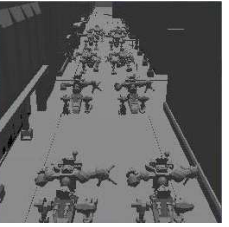
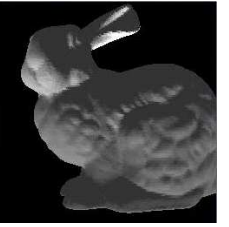





				
Benchmark	UlmiBox	Robots	Stanford Bunny	Kitchen
Triangles	492	69,296	69,451	110,540
<i>kd</i> -tree Depth	17	33	27	31
GrUG Tree Depth	4	17	8	13
				
Benchmark	Fairyforest	Atrium	Conference	Happy Buddha
Triangle Count	172,561	559,992	987,522	1,087,716
<i>kd</i> -tree Depth	36	37	35	34
GrUG Tree Depth	21	19	20	14

Table A.2: Benchmark scenes with triangle count and tree data structure parameters.

the second operation reads the root node boundary values for computing t_{max} . In addition, the GrUG software kernel is called more times than there are down traversal operations, due to smaller tree data structures and that many of the GrUG groups do not require a tree data structure. Using smaller grid sizes for GrUG results in a higher number of software tree traversal steps and an increase in the number of executed instructions. While adding software tree traversal steps increases the run time, the performance for a grid size of 128 is still improved over software implementations by 1.9X compared to 2.15X for a grid size of 512.

In addition to comparing the number of executed instructions, we simulated the first 300,000 cycles for the Conference benchmark scene using a resolution of 128. Figure A.11 shows the number of finished rays during this time period. The GrUG hardware pipeline was able to compute 2,500 more rays than the Radius-CUDA implementations.

Executed Instructions For Visible Rays

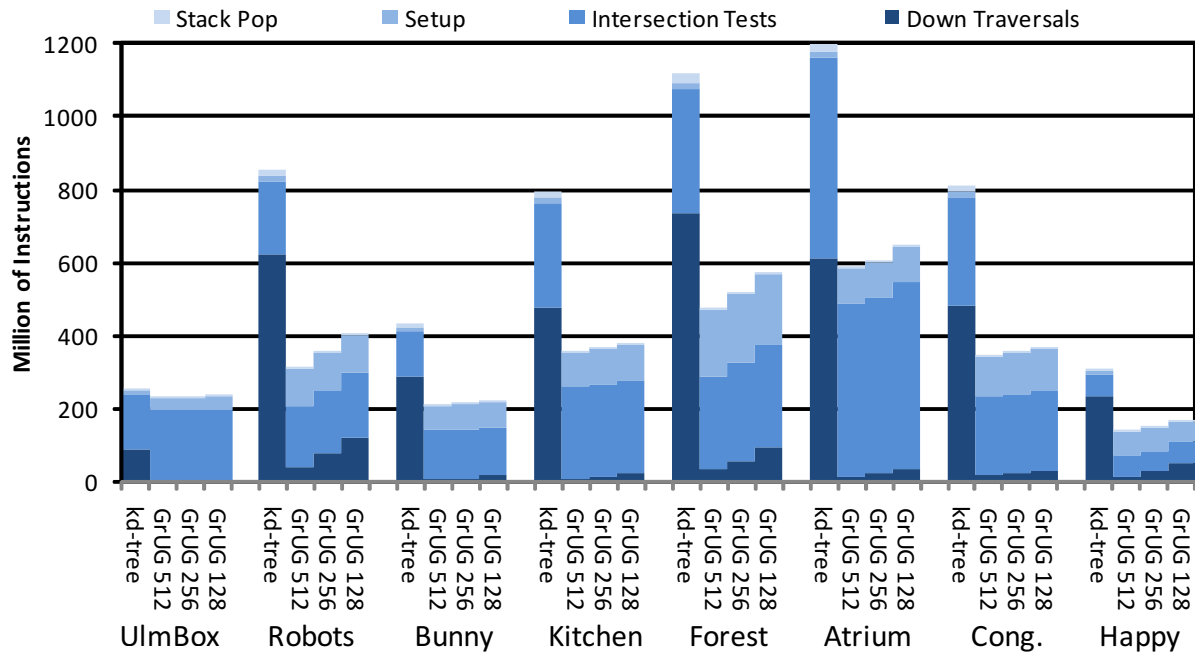


Figure A.10: Total number of instructions executed for traversal of visible rays. *kd-tree* results are compared against GrUG using grid sizes of 512, 256 and 128.

Memory

Figure A.12 shows the total system memory needed to store the different data structures used for combining GrUG and *kd-tree*. The use of a hash table in GrUG results in a significant overhead in memory requirements to store the entire hash table in memory. The required memory is based on the GrUG grid size, 4 bytes per grid cell. A fixed grid size of 512 will use 512MB to store the hash table. Four bytes per grid cell is a conservative usage resulting in a maximum of 4,294,967,296 supported GrUG groups. Scenes requiring 65,536 or fewer GrUG groups can use 2 bytes per grid cell, resulting in only a 256MB hash table. The use of smaller grid sizes reduces the memory requirements for the hash table down to 4MB for the 128 size grid. On average, using a grid size of 128 requires only 1.5 times the memory of a *kd-tree*, compared to a grid size of 512 that requires 27.6 times the memory of a *kd-tree*. Figure A.13 shows the ratio of the performance increase divided by the memory overhead for different grid sizes. Having larger grid sizes offer better overall performance, but the memory overhead grows

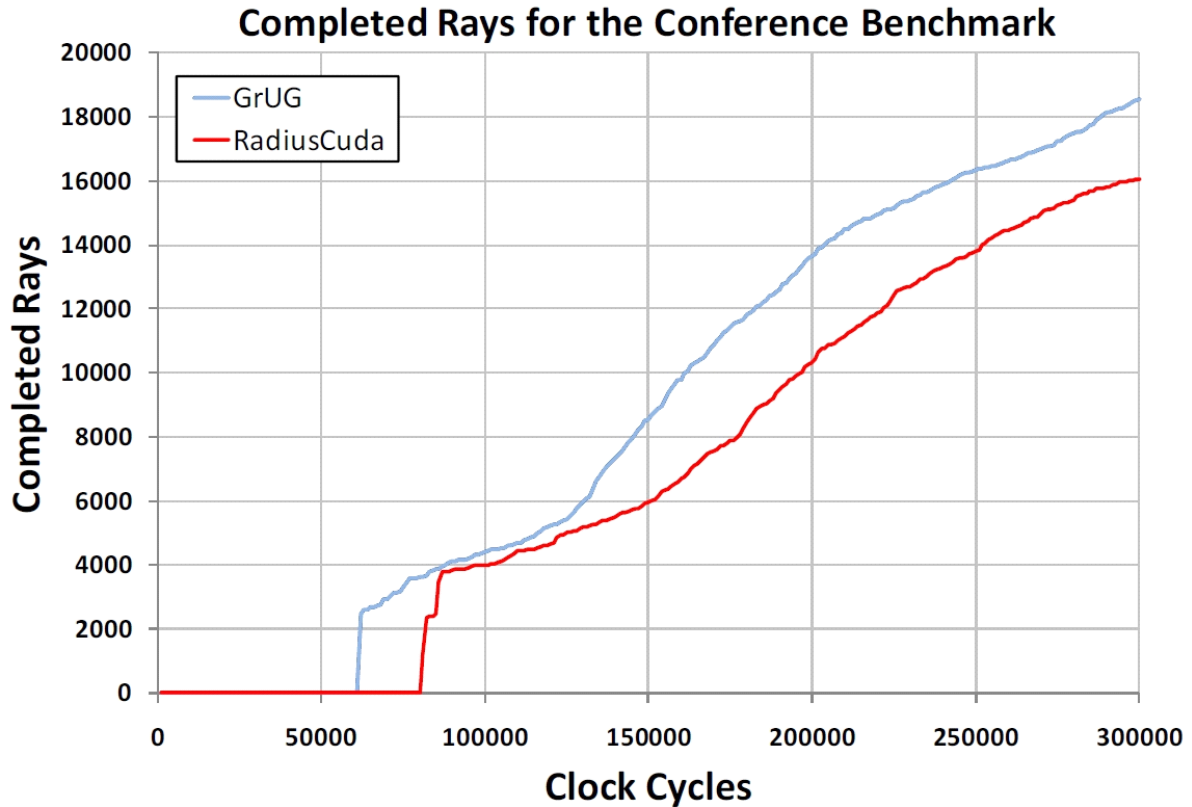


Figure A.11: Number of completed rays for the first 300,000 clock cycles of the Conference benchmark scene.

faster than the performance benefit. Using smaller grid sizes offer a nice balance of increasing performance to the increased memory requirements.

In addition to the hash-table, the kd -tree structure and bounding dimensions of all threshold nodes must be stored in system memory. While GrUG requires a smaller tree data structure, the memory requirements are similar to the full kd -tree data structure. An additional 24 bytes are needed for storing the bounding dimensions per GrUG grouping, equivalent to 3 nodes. This results in similar memory requirements for storing the tree data structures as a full kd -tree.

Bandwidth

While GrUG requires more memory storage than standalone tree data structures, the average memory bandwidth per frame is smaller. Figure A.14 show the memory bandwidth for each benchmark without caching. Since we are reducing the number of down tree traversal steps,

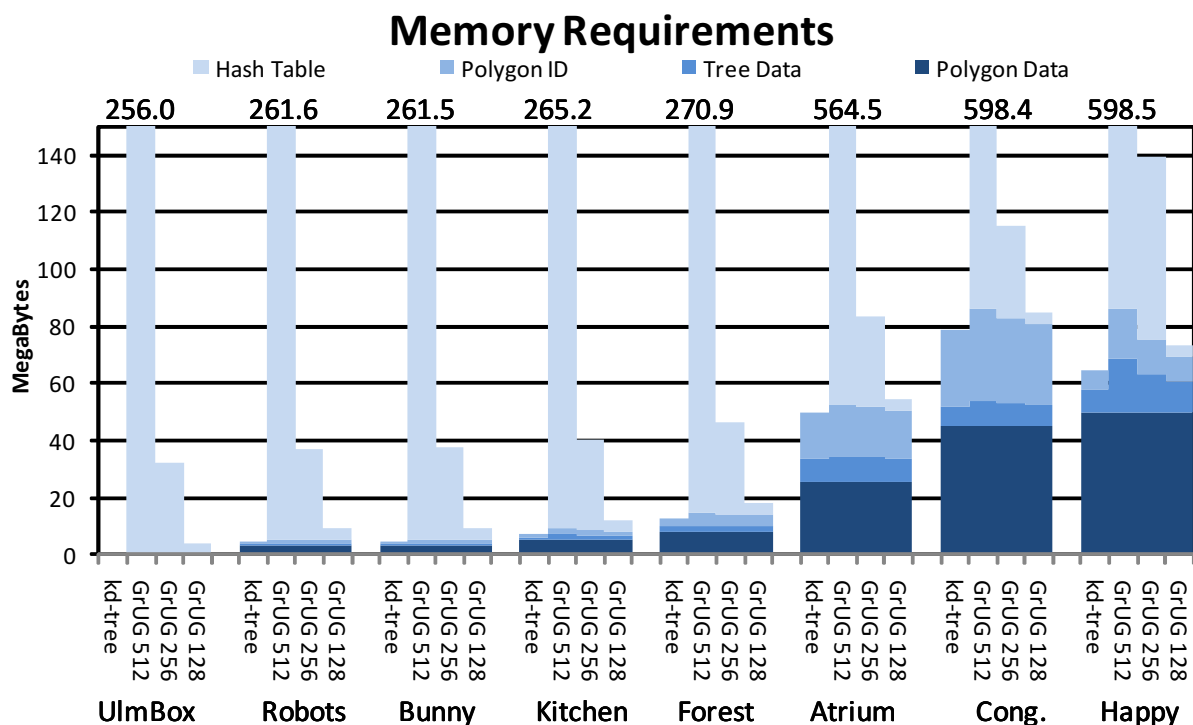


Figure A.12: Memory requirements for kd -tree and GrUG.

the amount of device memory transactions required is significantly less. The use of GrUG results in a majority of the bandwidth required for rendering a frame being used for post GrUG software traversal instead of user-defined traversal due to the number of memory reads from the boundary box and root node for every ray outputted from the hardware pipeline. However the sum of the memory bandwidth of GrUG and down tree traversal is smaller than the down traversals required by a full software implementation.

Conclusions

Data structures have had a dramatic impact on the performance for ray-based rendering methods. To further increase performance of data structures, this paper proposed a method for reducing the number of recursive steps while still implementing common HS3 data structures. Our method of reducing the number of recursions in the data structure allows for an accelerated hardware implementation, further reducing the workload of the processor. We used an existing performance simulator for comparing our GrUG implemented kd -tree against a modern day kd -

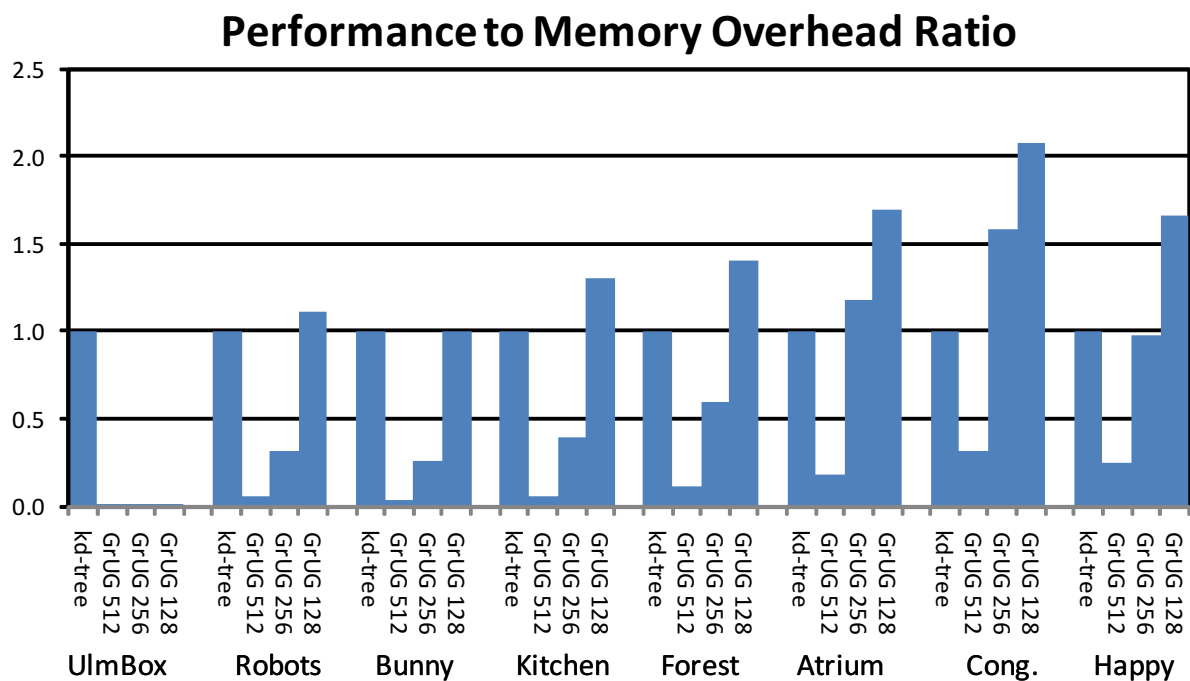


Figure A.13: Ratio of the performance increase divided by the memory overhead for different benchmarks and grid sizes. Performance increase and memory overhead are based on the *kd-tree* algorithm resulting in a value of 1.0 for the *kd-tree*.

tree implementation. Our experimental results show a significant reduction in system memory reads for large polygon count scenes.

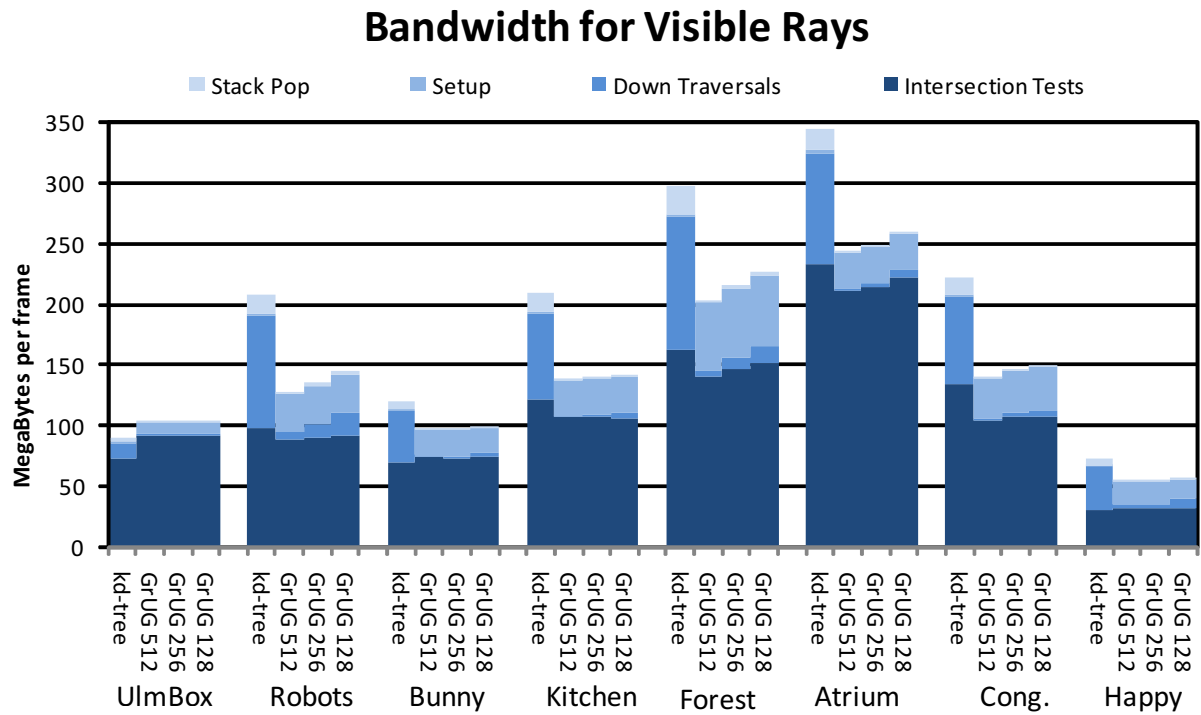


Figure A.14: Memory bandwidth per frame required to render visible rays without caching.

APPENDIX B. Teaching Graphics Processing and Architecture

Introduction

Computer graphics is becoming increasingly important to modern-day computing. A wide range of computation is being pushed to GPU processors (displays, 3D graphics, image processing, parallel computing) [16]. Originally implemented as a fixed-function processor for 3D graphics, GPUs have evolved into a multi-purpose programmable parallel processor. Mastery of modern computer graphics requires more than the knowledge of a 3D graphics API. Today, graphics developers need to understand the underlying GPU architecture, both its strengths and limitations, as well as the interaction between the CPU and GPU, in order to write efficient high-level code. Seniors in computer engineering at Iowa State University (ISU) are exposed to concepts in device interfacing and hardware/software optimization through multiple classes in software development, computer architecture, digital logic and signal processing. A course that focuses on graphics processing and architecture has the potential to nicely tie together several instances of these concepts in an integrated environment.

At ISU we have created a senior elective class for teaching graphics processing. While this class is offered as an elective in the computer architecture focus area, course topics are introduced from the systems perspective, requiring students to use knowledge across multiple computer engineering subfields. For students to acquire a deep understanding of the complexity required to support modern computer graphics, weekly laboratories require students to implement specific graphics processing components, spanning the entire system implementation from software API to the specific GPU architectural components. These architectural components implemented in lab are loosely derived from the conventional OpenGL 3D rendering pipeline. Each lab assignment expands the processing pipeline, refining a stage from the software do-

main to the hardware domain. By the end of the semester, students are able to run OpenGL applications that are rendered using their graphics processing system. Students are able to prototype their OpenGL pipeline by synthesizing their architecture to an FPGA. By using an existing standard (OpenGL), the implemented system allows for existing graphical applications that would conventionally run on a workstation to also be rendered using the students' designs running on the FPGA.

To attract a wider range of students, no computer graphics prerequisite is required. Due to this, part of the class time is spent on teaching basic computer graphics principles. To differentiate this class from a conventional computer graphics course, we make the trade-off of breadth versus depth of knowledge, spending more time on understanding the entire system perspective for the concepts versus presenting a wider range of more advanced algorithmic concepts in the computer graphics domain. Since the entire system perspective is covered from software to hardware design, prerequisites for the class requires strong software and Hardware Description Language (HDL) programming skills, limiting the students to upper-level electrical and computer engineers.

This class was first introduced in Spring 2011 and is being taught again in Spring 2012. The class is offered as a 4 credit (3 hour lecture + 3 hour lab per week) senior elective course. The students' completed laboratory assignments surpassed our expectations in terms of creativity from both the software and the hardware design perspective.

Laboratories

A total of 7 laboratory assignments were given during the semester. Students worked in the same group of 3 to 4 students for all laboratories and had two weeks to complete each lab. During the two week time period, students have two 3-hour lab sections, four 2-hour teaching assistant hours in the lab, four 2-hour instructor office hours and an on-line forum related to the course that was monitored frequently by the teaching assistant and instructor. Groups were formed on the first day of class and allowed students to form their own groups with the exception that all students in a group must be in the same lab. To aid in creating groups, students evaluated their experience in three categories: software development, hardware

design and computer graphics. Students were advised to form well-balanced groups, since all laboratory assignments require some aspect of all three categories. We did allow teams to re-organize during the semester, based on student feedback.

The first laboratory served as an introduction to the software tools and hardware infrastructure that was common across all labs. The remaining 6 assignments implemented individual pipeline stages for a fixed-function OpenGL-compliant 3D rendering pipeline. We refer to the laboratory assignments as Machine Problems (MP), as they each included significant software and hardware components. Throughout all the MPs, students were responsible for implementing parts of the software driver for OpenGL API functions, hardware architecture for the OpenGL pipeline, HDL implementation of the architecture, culminating in running the complete system using a workstation connected to an FPGA-based platform. In addition, students were also required to implement their own OpenGL applications for additional debugging and verification. Each MP also had a bonus for either the fastest design or most creative, depending on the requirements for the MP. Each group was only allowed to receive the bonus two times, allowing for each group a chance to receive the bonus.

The course was organized such that by the start of each MP, students had been introduced to the functional requirements for each pipeline stage, as well as a mathematical foundation. Students were allowed to utilize the provided algorithmic solution or create their own, but were required to meet the functional requirements for the component. Since all the MPs build on the OpenGL pipeline, students were provided with the previous labs solution and encouraged to use it for the next MP. This prevented students from falling behind and also simplified debugging of subsequent labs, since the entire class was working from the same codebase. To prevent students from falling behind with regards to the course material, each group was required to demonstrate each MP, no matter their degree of completeness. Additional in-person mentoring was provided for groups that had difficulties in understanding key concepts; however, implementation errors were much more common than conceptual errors.

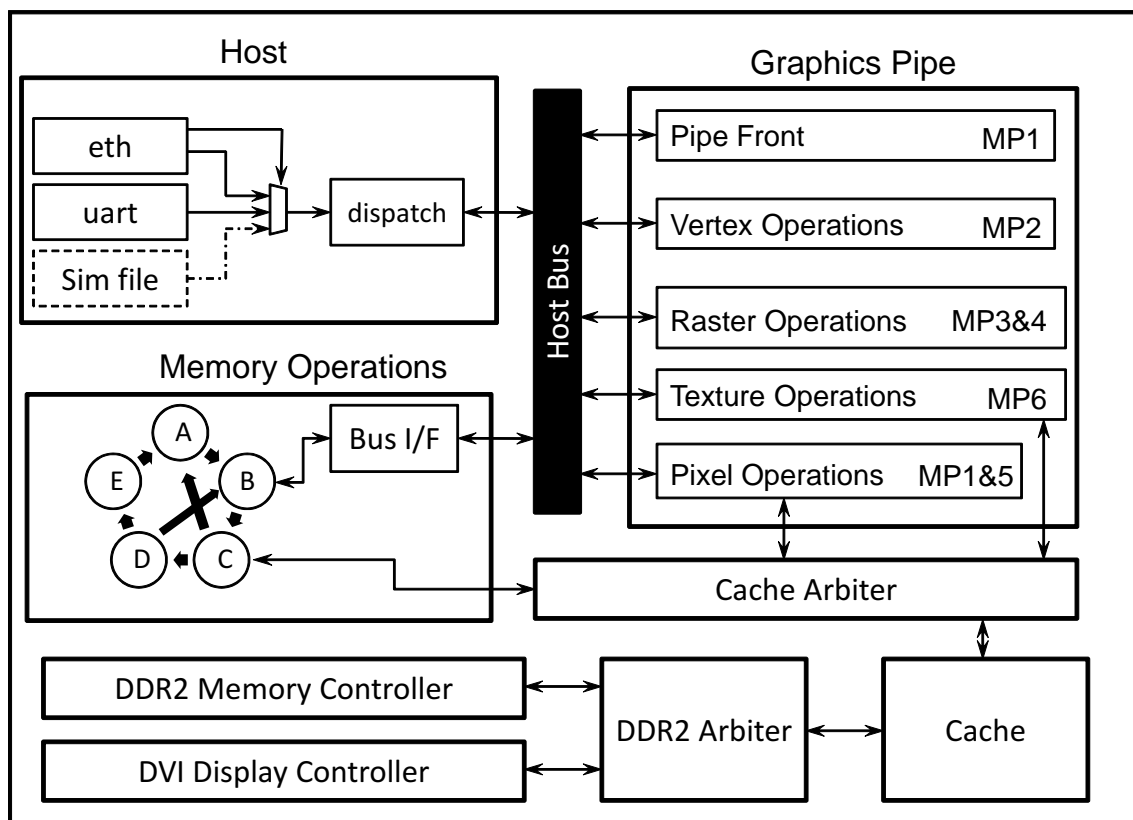


Figure B.1: FPGA architecture framework provided to students. Students were responsible for implementing code in the graphics pipe sub-unit.

Laboratory Infrastructure

To allow students to focus more on the specific components being taught in each MP, an initial hardware/software framework was provided. Students then expanded the capability of the framework in each of the MPs. An overview of the provided framework is shown in Figure B.1, which supports the following capabilities: source code for an OpenGL API (absent components the students will implement), communication protocol between the PC and FPGA board (software library and hardware interface), FPGA-based hardware Network-on-Chip, FPGA-based DDR memory system, FPGA-based hardware DMA controller and FPGA-based DVI display interface hardware. Providing these capabilities allowed the students to spend their time on implementing the pipeline stages, without worrying about all of the details of building a complete hardware/software system from scratch (which is the focus of other courses at ISU).

The open-source project *GLIntercept* [34] was used as a baseline for our OpenGL driver.

GLIntercept is an OpenGL API function call logger that replaces the workstation's native OpenGL driver. For every OpenGL API function call, an entry in a log file is created. *GLIntercept* also loads the native driver and forwards all API calls to the workstation's GPU (in our case, an NVIDIA GeForce GTX 480 GPU [44], so that the application will be correctly rendered to the screen. *GLIntercept* was modified to send instructions to the FPGA GPU reference design. Each OpenGL API function call has a function in *GLIntercept* that allows for custom messages to be sent from the workstation to the FPGA framework. By rendering a scene both natively on the workstation as well as using the FPGA-based framework, students are able to visually inspect their solutions to detect discrepancies from the "correct" workstation GPU implementation.

To aid students in implementing driver code and sending instructions between the workstation and the FPGA-based GPU implementation, a protocol was provided along with source code implementation and a corresponding HDL code for the on-chip packet forwarding. The protocol sends messages of 32 bits and must be formatted to meet the protocol outlined in Figure B.2. This protocol allows for data to be sent from the workstation OpenGL driver to multiple sub-units memory mapped as registers. The **Length** field specifies the number of 32-bit packets that follow the starting packet and allows for custom data formats to be sent to the sub-units. The **OpCode** allows the sub-unit to support multiple instructions and distinguish between them. The **Address** field specifies which of the sub-units to send the messages too. The provided network-on-chip/forwarding hardware interprets the incoming messages and forwards them on to the appropriate sub-unit registers. The physical connection between the workstation and FPGA uses either Gigabit Ethernet or RS232 serial UART. Selecting between the two methods is possible using Linux environment variables, and is fully supported by the framework. Having multiple communication speeds between the workstation and FPGA-based GPU implementation provided additional performance debugging methods.

For prototyping the GPU architecture a Xilinx Virtex-5 FPGA XUPV5-LX110T board was selected [32]. The XUPV5-LX110T board has 256MB of DDR2 memory available for a frame buffer and additional data storage. The Virtex-5 FPGA uses two clocks for system components and another one for the graphics pipeline. The graphics pipeline component (where student

code was implemented) uses a 100Mhz clock; consequently, students were required to meet this timing requirement for all of their hardware modules. The rest of the system runs at 200Mhz. To simplify the display hardware and clocking, a fixed resolution of 1280x1024 was implemented using the DVI display interface. The memory address for the frame buffer is set through a memory-mapped register, allowing for students to switch between different frame buffers. All memory addresses provided to students are in a virtual address space. The FPGA DDR2 memory uses a word size of 128-bits, equivalent to 4 pixels of 32-bits each.

To simplify the access to the DDR2 memory module, a cache and address translator was provided. The virtual address word size was 32-bits wide, allowing for each unique address to represent a single pixel. The cache provides the address translation by providing two different port sizes. A total of 8 megabytes is allocated for each frame buffer. While the total required memory is 5 megabytes for a tight compaction of data, the equation for computing a pixel address for a tight compaction is: $address = (linewidth) * Y + X = 1280 * Y + X$. Since 1280 is not a power of two, a hardware multiplier would be required. Assuming the width of the display to be the next power of two, allows the base design to use only a shifter to compute the address, rather than a multiplier (the display hardware knows to stop reading at 1280 and not continue to 2048). By padding the framebuffer, we reduce the hardware complexity and simplify the math, at the expense of using more memory. In our framework, reconfigurable FPGA resources are more limited than DDR2 memory.

All but the first MP assignment expands the functionality of this framework. Specifically, MP-1 through MP-6 adds to the graphics pipeline by adding sub-units. Each sub-unit uses the same interface protocol allowing for new sub-units to be inserted between other sub-units. While most of the MPs focus on a specific sub-unit, some assignments span across multiple sub-units. Figure B.1 shows the basic OpenGL pipeline sub-units implemented and the corre-

X	Length												OpCode								Addr.										
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSB																															LSB

Figure B.2: The protocol for sending data from the workstation to the FPGA. This format represents the first 32-bit message in a packet.

sponding MP assignments.

Machine Problem 0 (MP-0)

The goal of the first MP was to allow students to become familiar with their new team members and the lab's hardware/software infrastructure. We also wanted to give students a refresher in HDL design methodologies. To ease the transition to future labs, MP-0 required students to implement matrix-vector multiply-accumulation on the FPGA. A constant 4x4 matrix along with 10000 4x1 vectors are used in computing the 4x1 sum. Students were provided with sample code that provided all the initial data stored in FPGA memory and an interface for sending results to the workstation. The MP-0 lab handout walked students through the process of understanding the provided framework code and how to compile, synthesize, program, and run the code on the FPGA. After running the code, students realize that the output from the provided framework code is not the correct solution (since the vector sum hardware is not implemented). The teams were given two weeks to implement the hardware to perform the matrix-vector multiply-accumulation, requiring students to read the vector data from on-chip memory and perform all math operations for the matrix vector multiplication and accumulation. As the bonus criteria for this MP is performance, a cycle counter is included in the HDL code to count the number of cycles required for all operations to be performed. Students indicate when all operations are finished by setting a register that stops the counter and triggers the output results to be sent over UART to the workstation for verification. Students were not allowed to modify the provided code with the exception of changing the on-chip (blockram) memory width.

MP-0 Student Results

Common solutions for this problem used 4 multipliers in parallel and then summed the results for computing one of the values in the vector. A Finite State Machine (FSM) is implemented for loading data out of memory and feeding the multipliers and directing the outputs to the right register locations to compute all 4 values of the vector. The initial state of the FSM reads the 4x4 matrix values and stores them in appropriate registers that the multipliers

will access. The final two states of the FSM oscillates between loading the vector data from memory and performing the multiplications.

Teams competing for the bonus increased the number of multipliers to perform more operations in parallel. One such implementation added as many multipliers as permitted by the resources on the FPGA. These implementations also utilized an FSM for reading data from the FPGA memory and a separate one for performing the multiplications.

Machine Problem 1 (MP-1)

The second MP is the first assignment in developing the 3D OpenGL rendering pipeline. Students are also exposed to the framework that they will be using for the remainder of the class. Since the provided framework is very large and complex, a majority of the assignments is understand the capabilities of framework and how it is all implemented. The MP-1 handout guides students through all the software libraries and the provided HDL code. In addition, students are also introduced to all the protocols that they will have to use. The lab handout asks students to evaluate different parts of the framework and describe how it works in their report.

In addition to understanding the provided framework, students are required to implement a portion of the driver and HDL code. The driver implementation requires students to implement the API function that clears the entire screen. Implementing the clearing instruction requires students to send messages from the workstation driver to the FPGA framework memory controller to write zeros to a range of memory addresses that correspond to the frame buffer address. Correct implementation requires 4 32-bit messages to be sent to the FPGA, with the challenge being understanding how to use the software driver, workstation to FPGA protocol, functionality of provided HDL sub-units and the FPGA memory space.

The second portion is to draw pixels to the screen. The driver code for sending OpenGL vertices from the workstation to FPGA was provided to students. The format for sending vertices to the FPGA is done using multiple lists queues outlined in Figure B.3. The advantage for of using list queues reduces the amount of data that is required to be sent between the workstation and FPGA for large complex scenes. Students had to implement the HDL code

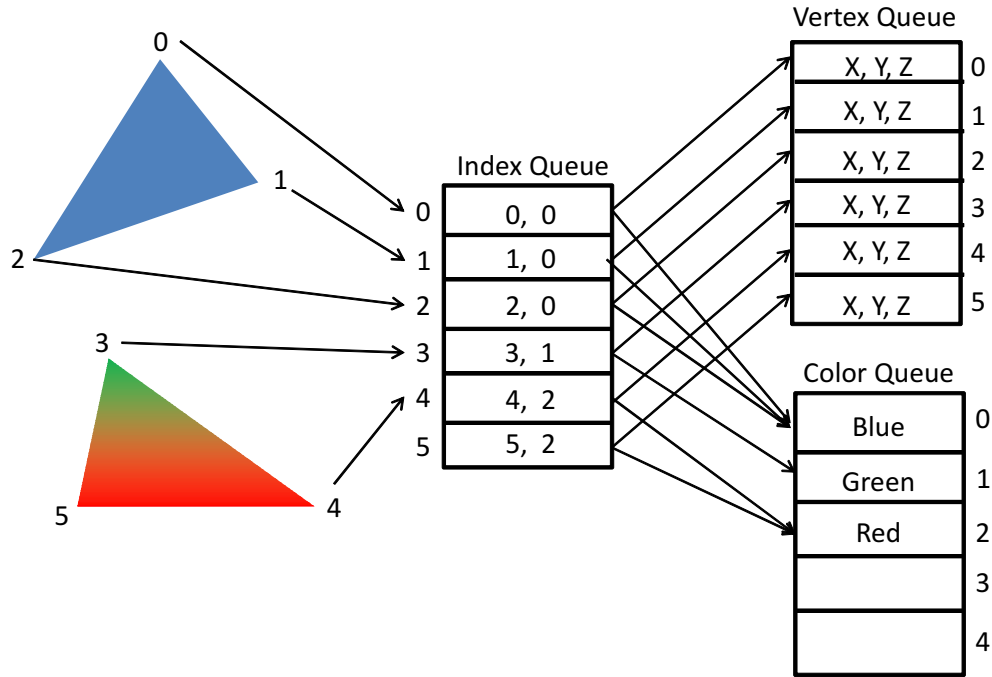


Figure B.3: Format of how vertex attributes are sent to the FPGA and then stored in memory. The index queue stores address for the vertex and color queue for each vertex.

to read from the different list queues to get the X, Y locations for the vertices and the correct color for each vertex. Once the X,Y vertex location and color are known (no transformations were required for this lab) the framebuffer memory address is computed. Once the address is known, students interface with the provided memory controller to write the color value to the address in memory. To verify their code, students also had to implement their own OpenGL application that drew pixels. The bonus for this MP is how creative students can be in creating an application using only pixels. Students then had to demo their OpenGL application being rendered with the FPGA.

MP-1 Student Results

Two similar solutions for reading vertex data were implemented. One method implements the process of reading the data from the queues as a pipeline. A counter is used to increment through the index queue and feed the pipeline with the data results from the index queue. The second stage uses the data in the first pipeline stage as addresses to read from the remaining index queues. The output from these queues is then used for computing the memory address.

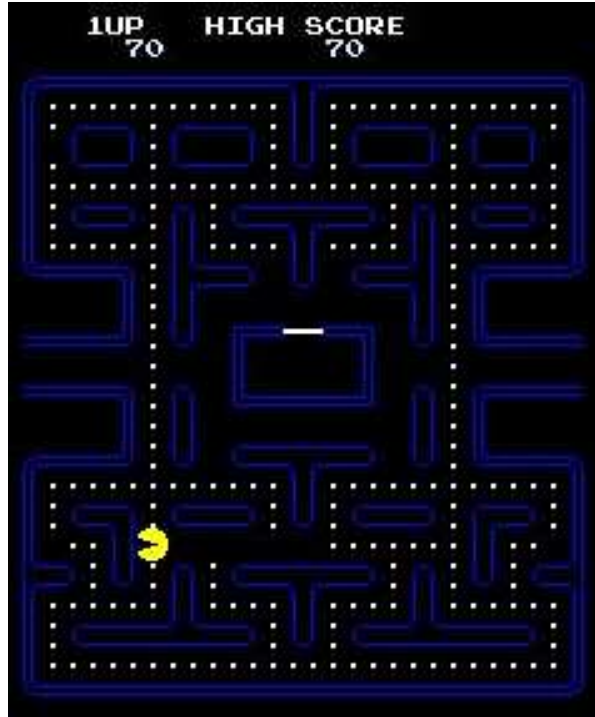


Figure B.4: Sample student OpenGL application that draws a 2D video game using pixels.

The second method implements a FSM. The different stages of the FSM reads values from the different queues. While the FSM transitions between states in order decreasing performance, students find implementing state machines easier than implementing multiple pipeline stages.

A sample OpenGL application developed by students drawing individual pixels is shown in Figure B.4. Students implemented a fully functional 2D video game with multiple moving characters. Other student applications drew their name or other basic shapes.

Machine Problem 2 (MP-2)

The mapping of 3D points defined in the OpenGL coordinate system to the 2D screen is accomplished through the application of multiple transformation matrices. OpenGL uses 2 4x4 matrix transformations, one vector division and a 2x2 matrix transformation for converting 3D coordinates to 2D screen coordinates [61]. The transformation process is shown in Figure B.5. MP-2 requires students to implement the hardware for all of these matrix multiplications and division. No software driver implementation is required, due to the complexity of the hardware implementation. Students are also required to describe how the different matrices are sent to

the FPGA in their lab writeup. Due to limited resources on the FPGA only 4 64-bit multipliers, a single 64-bit divider and half of the available DSP slices could be used. Since floating-point operations are not efficient on FPGAs, fixed-point notation is used. Conversion to fixed-point notation is done inside of the workstation driver code.

The fixed-point notation for input vertices is Q32.32. The two 4x4 matrix transformations are also stored in Q32.32. The division operation is used for inverting vertex values requiring no fixed-point notation. The 2x2 matrix transformation is composed of 11-bit unsigned integers and the final vertex output corresponds to the pixel location requiring two 11-bit unsigned integers for a 1280x1024 resolution. When designing their architecture for MP-2, students were required to understand the precision requirements needed for each multiplication operation to implement the correct size multiplier. In this lab, the bonus is a function of performance divided by area. Performance is measured for the cycles required to render a specific benchmark application divided by the number of FPGA LUTs used.

MP-2 Student Results

Students implemented the first two matrix transformations in similar method as MP-0. The four 64-bit multipliers are used for the transformations and the FSM is slightly modified to perform the 4x4 matrix vector multiplication twice. The results after the two 4x4 matrix transformations are written into a FIFO. The FIFO is drained by a second FSM that reads one value every 3 cycles. The 3 cycle delay allows for 3 of the vector results to go through a pipelined divider implemented using the FPGA DSP slices. Results from the divider are

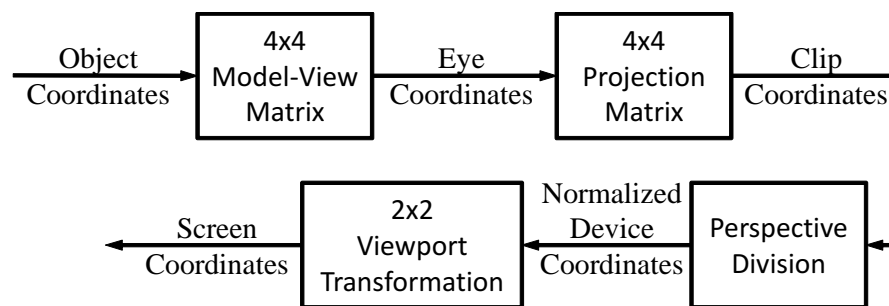


Figure B.5: Transformation process for converting 3D OpenGL coordinates to screen coordinates.

regrouped back into a vector and stored into a FIFO. The final transformation is implemented using 4 11-bit multipliers implemented in a pipeline method, since only 4 multipliers are needed for this transformation.

Machine Problem 3 (MP-3)

The rasterization pipeline stage is split into two labs, MP-3 and MP-4. In the rasterization stage, predefined OpenGL primitive types are pixelized such that every pixel enclosed in the primitive type is identified. All OpenGL primitive types are defined such that they all can be composed of 3D triangles. For example, a quad can be composed of 2 triangles. By having the basic primitive building block be a triangle, a single GPU rasterization hardware primitive can be implemented. Since the OpenGL API supports multiple primitive types, the process of composing primitives using individual triangles must be implemented in the driver or in hardware. Also, due to the fact that different primitives can reduce the number of vertices required to be passed from the workstation to FPGA, we implement primitive composition in hardware. For example a quad requires only 4 vertices whereas implementing a quad using 2 triangles requires 6 points (2 points being replicated).

MP-3 requires students to form individual triangles for rasterization from the different primitive types supported by OpenGL. Since OpenGL uses primitive types that are easily composed of triangles, creating the individual triangles can be performed by pipelining vertices and using some logic to select vertices from different stages of this pipeline. The second requirement for MP-3 is to generate the pixels that fill the triangle. Calculating the color for each pixel requiring interpolation using the vertex data is done in MP-4.

Students were provided with an algorithm and HDL code that checks if a pixel defined in X,Y screen coordinates is inside of the triangle being rasterized. To simplify the hardware requirements the algorithm only worked for testing pixels sequentially and sequential pixels must be adjacent. The starting pixel is one of the vertices of the triangle that is by definition inside the triangle. For students to check if a pixel is inside of a triangle, students must send commands to the provided HDL rasterization code. The commands are POP_MOVE_LEFT_CMD, POP_MOVE_RIGHT_CMD, MOVE_RIGHT_CMD, MOVE_LEFT_CMD and PUSH_MOVE_DOWN_CMD.

Using the PUSH commands allows for students to save their current location and return to it using the POP command. The push buffer is only a single entry deep.

MP-3 Student Results

To rasterize a triangle, all students implemented a FSM that used a zig-zag rasterization pattern. A sample zig-zag pattern is shown in Figure B.6. The logic for the FSM is as follows. First the current state is pushed. Then the Left direction is checked. The state stays in the left direction until the pixel is no longer in the triangle. The state is then popped, re-pushed and the right direction is checked. Once the pixel is out of the triangle the state is popped again and moved down. One exception to this is when the pixel is moved down and the pixel is not in the triangle. In this case both directions must be searched until the pixel is found to be inside of the triangle or the triangle bounding box is reached.

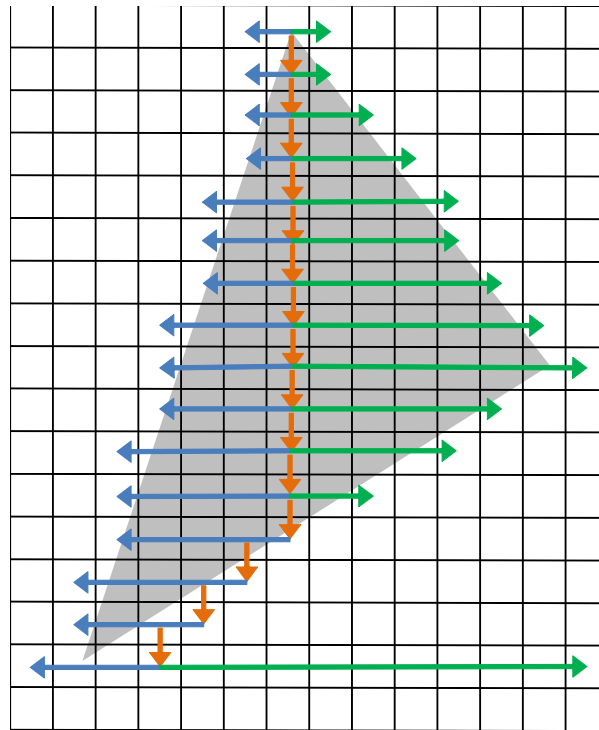


Figure B.6: A rasterization zig-zag pattern implemented by students to identify all pixels inside of a triangle.

Machine Problem 4 (MP-4)

The second stage of rasterization is to interpolate each pixel color from the three triangles vertex colors. Pixel color can be calculated in parallel with the rasterization check from MP-3. Students were required to implement the hardware unit that takes the same commands from MP-3 zig-zag rasterization commands and generate the pixel colors. The provided HDL code provides students with each color channel DX and DY values for the current triangle along with the vertex color for the starting vertex in the rasterization processes. Since the rasterization commands are sequential pixel offsets, students were required to subtract or add to the vertex colors based on the commands being inputted to the unit. Students were also required to implement their own OpenGL application that uses multiple primitive types to draw 3D objects that will also be used to test their color interpolation.

Machine Problem 5 (MP-5)

An important operation for drawing 3D objects is depth testing, which determines if a pixel for a triangle should be written to the framebuffer. While there are multiple depth testing functions, the most commonly used one is to test if a pixel for a triangle is in front of the current pixel in the framebuffer. If the recently generated pixel is in front of the one currently in the framebuffer the pixel should be written to the framebuffer. If the current pixel in the framebuffer is in front of the new pixel, the new pixel is discarded. To keep track of the depth for each pixel in the framebuffer a depth buffer is allocated in DDR2 memory, each pixel in the framebuffer has a corresponding address in the depth buffer. When depth testing is enabled, each pixel generated from rasterization must read the depth value from the depth buffer. Then, test the depth buffer value against the depth of the generated triangle using a specified function. The function used is set through the OpenGL API. If the new pixel passes the depth test, both buffers are written to with the new pixel values (color and depth).

MP-5 required students to implement the HDL code for reading depth values from the depth buffer and testing them against the depth function. All eight depth functions were required to be implemented and students were also required to write OpenGL applications to stress test

their design as well as prove correct functionality. It is important to note that the XUPV5-LX110T FPGA board we use only allows for a single DDR2 memory controller. As depth checking requires the hardware to read from the depth buffer, perform the test, write to the depth buffer, and write to the framebuffer before subsequent pixels can be processed, students were not able to fully pipeline this stage. Because of this limitation students are also required to implement methods for measuring the performance overhead of using the depth test.

MP-5 Student Results

Due to the long latencies of reading from the DDR2 memory with only a single memory controller, most students implemented another FSM for this MP. The first step is to read from the DDR2 memory to get the depth value. Once the depth buffer is read, they evaluate the depth function. The eight depth functions are implemented in a similar method as a conventional ALU. All eight functions evaluate the results and a mux is used to select the results for the depth functions specified by the OpenGL API. If the depth function passes, the depth value and color value are written to memory.

The second part of this MP was for students to implement OpenGL code to test their depth function. One innovative application is shown in Figure B.7. For each horizontal bar, two primitives are drawn using different colors. The first primitive is drawn at a constant depth value of zero using the color black. The second primitive uses a sin wave for the depth and sets the color based on the depth (a depth of 1 is blue and -1 is red). Each horizontal bar uses a different depth function. This team then examined the pattern of the two colors to determine if the depth function is working correctly. For example, the second horizontal bar uses the depth function of greater than. In a correct implementation, only pixels from the sine wave that are greater than zero (colored in blue) should be drawn.

Machine Problem 6 (MP-6)

The final MP completes the basic OpenGL pipeline functionality by allowing images to be mapped onto triangles. This process is commonly referred to as texturing, or texture mapping. Each pixel generated from rasterization has both a color value (interpolated from the triangle's

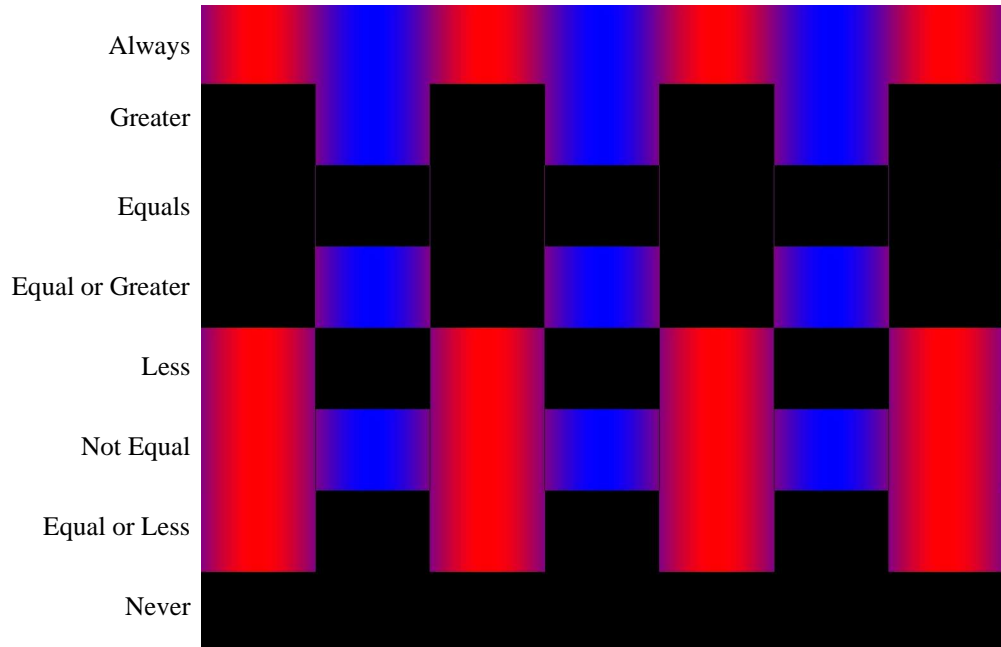


Figure B.7: OpenGL application used to test all 8 depth functions. Each horizontal bar uses a different depth function test. The black color is a plane at depth 0 and the red and blue colors are a sin wave with depths from -1 to 1.

vertex color values) and also a texture coordinate (also interpolated). Texture coordinates typically range from 0 to 1 as shown in Figure B.8. Each pixel has two texture coordinates that map the image's X and Y axis. The texture coordinates are then used to look up the pixel color from the texture image. While there are multiple filtering algorithms used to determine the pixel color from a texture image, only the nearest pixel value is implemented requiring no image processing algorithms. In addition, texture coordinate interpolation from rasterization currently does not perform projection transformation resulting in a slight errors when rendering images using a perspective viewing transformation.

MP-6 required students to implement the hardware architecture to compute the memory address of the texture image stored in memory from the pixel texture coordinates. When a texture is to be applied to a triangle, the image is copied into a texture cache allowing the starting memory address of the image to be zero. To complicate the computation, OpenGL only restricts the texture image to be a power of two (they can otherwise be any size). The size of the texture image are stored in two registers that students use to compute the memory address.

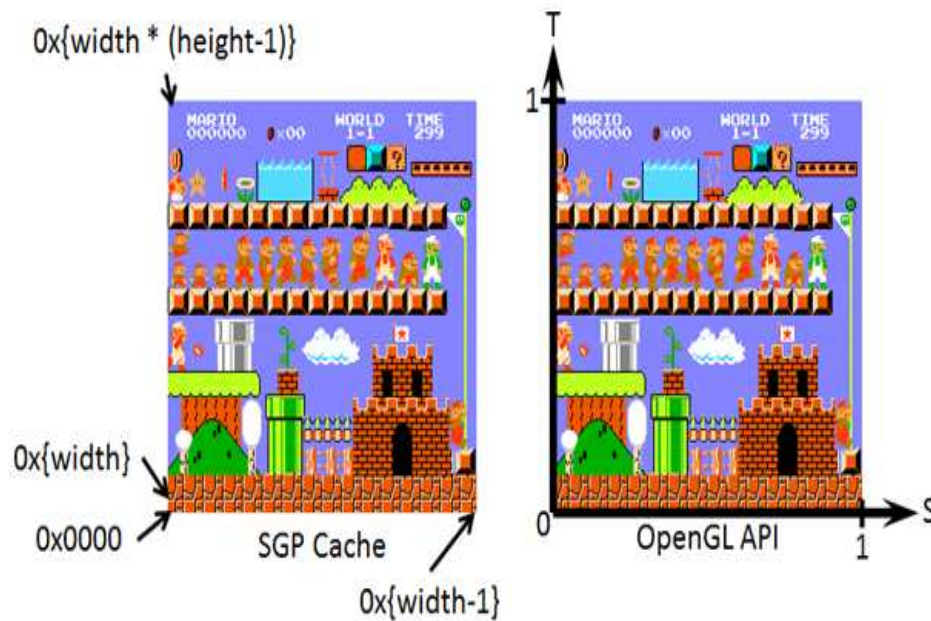


Figure B.8: Texture coordinates and how they map to the texture image memory address.

In addition, to the texture mapping students were required to analyze the entire graphics pipeline implementation to identify performance bottlenecks using some non-trivial applications, such as ID Software's Quake [65] video game.

MP-6 Student Results

In MP-6 students had difficulty implementing the calculation to compute the texture image memory address from texture coordinates and only a few groups were able to get textures working. In addition, while many groups were able to perform a detailed analysis of the pipeline performance, none were able to significantly alleviate any of the bottlenecks. Students struggled with understanding the fixed-point notation requirements needed since texture coordinates are implemented as fixed-point values. To add to the difficulty, both simulation and FPGA synthesis increased in time due to the increasing complexity from all the previous MPs.

Conclusions

We have created a senior elective computer architecture class that focuses on graphics processing and architecture. By the end of the semester, students gain an understanding of key concepts in computer graphics along with a complete system-wide perspective for rendering 3D images. Students also obtain valuable hands-on experience in implementing the 3D graphics pipeline on an FPGA. Laboratory assignments allowed students to implement a 3D OpenGL pipeline from the workstation API all the way down to the GPU architecture running on a physical FPGA board.

Students filled out conventional evaluations for this course at the end of Spring 2011 semester. The feedback from students was generally positive - they enjoyed the opportunity of being able to work on the entire system rather than being limited to specific problems. Out of the 22 respondents, on average the students rated the course as a 4.91 out of a 5 point scale for overall effectiveness. In terms of whether or not the course inspired students to learn more about computer graphics and architecture, on average the rating was a 4.50 out of a 5 point scale. Specific feedback comments pointed to both positive and negative aspects of the MP structure. While the framework prevented students from falling more than 2 weeks behind the rest of the class, some concerns were raised that this also limited the potential for creative solutions to those same 2 week intervals.

The Spring 2012 offering was modified to replace MP6 with a multi-week project. Students can pick from a provided list of projects or implement their own idea. The project is designed to allow students the flexibility of generating their own hardware and software implementation strategies (either new functionality or improving current functionality) without any limitations on using the provided framework. Future modifications will look into improving the graphics pipe unit interfaces to reduce pipeline stall logic the students were required to design. While not all of the pipeline stall logic can be removed entirely since we want to support a wide design space, improvements can be made to reduce the complexity to allow students to focus more on the main MP learning objectives.

All provided student material (hardware designs, software infrastructure, and laboratory

manuals) as well as solutions for the individual Machine Problems are available to instructors by request through our research group's website [56].

BIBLIOGRAPHY

- [1] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics*, pages 145–149, 2009.
- [2] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. Storegpu: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 165–174, 2008.
- [3] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, pages 3–10, 1987.
- [4] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos. Comparing cuda and opengl implementations for a jacobi iteration. In *Technical Report SFB-Report No. 2008-025*, Universitt Graz, Graz, Austria, 2008.
- [5] Aaron Ariel, Wilson W. L. Fung, Andrew E. Turner, and Tor M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 164–174, 2010.
- [6] ATI. ATI Stream Computing, Compute Abstraction Layer Programming Guide 2.0, 2010.
- [7] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of Performance Analysis of Systems and Software*, pages 163–174, 2009.
- [8] Benjamin Segovia. Radius-CUDA. <http://www710.univ-lyon1.fr/~bsegovia/demos/radius-cuda.zip>, 2008.

- [9] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray tracing on the cell processor. In *IEEE Symposium on Interactive Ray Tracing*, pages 15–23, sept. 2006.
- [10] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [11] Billconan and Kavinguy. A neural network on gpu. In <http://www.codeproject.com/KB/graphics/GPUNN.aspx>.
- [12] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The illiac iv system. *Proceedings of the IEEE*, 60(4):369–388, april 1972.
- [13] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.
- [14] Ian Buck and Pat Hanrahan. Data parallel computation on graphics hardware, 2003.
- [15] D.W. Chang, C.D. Jenkins, P.C. Garcia, S.Z. Gilani, P. Aguilera, A. Nagarajan, M.J. Anderson, M.A. Kenny, S.M. Bauer, M.J. Schulte, and K. Compton. Ercbench: An open-source benchmark suite for embedded and reconfigurable computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 408–413, 31 2010-sept. 2 2010.
- [16] NVIDIA Corporation. Directcompute programming guide v3.2. <http://developer.download.nvidia.com/>.
- [17] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum*, 27(4):1225–1233, 2008.
- [18] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. In *Proceedings of the International Symposium on Microarchitecture, MICRO '44*, 2011.
- [19] Philip Dutre, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*. AK Peters, Ltd., Natick, MA, USA, 2002.

- [20] Joshua Fender and Jonathan Rose. A high-speed ray tracing engine built on a field-programmable system. In *Proceedings of Field-Programmable Technology*, pages 188–195, 2003.
- [21] Randima Fernando and Mark Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, Indianapolis, IN, USA, 2003.
- [22] Wilson Fung and Tor Aamodt. Thread block compaction for efficient SIMT control flow. In *Proceedings of High Performance Computer Architecture (HPCA)*, pages 25–36, 2011.
- [23] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of Symposium on Microarchitecture*, pages 407–420, 2007.
- [24] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Computer Graphics*, 18(3):213–222, 1984.
- [25] Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William Mark. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of IEEE/ACM Symposium on Microarchitecture*, pages 176–187, 2008.
- [26] Johannes Hanika. *Fixed Point Hardware Ray Tracing*. PhD thesis, Ulm, Germany, 2007. University-Universität Ulm.
- [27] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing, HiPC'07*, pages 197–208, 2007.
- [28] Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.
- [29] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Prague, Czech Republic, 2000. Czech Technical University in Prague.

- [30] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of Interactive 3D graphics and games*, pages 167–174, 2007.
- [31] RapidMind Inc. Libsh. In <http://libsh.org/>.
- [32] Xilinx Inc. Xupv5-lx110t development board. Available: <http://xilinx.com/univ/xupv5-lx110t.html>.
- [33] Dave Baldwin John Kessenich and Randi Rost. The OpenGL Shading Language (Version 1.10), 2004.
- [34] Phil Frisbie Jr. Gltrace. Available: <http://hawksoft.com/gltrace>.
- [35] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.
- [36] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31:7–17, 2011.
- [37] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 149–158, New York, NY, USA, 2001. ACM.
- [38] Svetlin A Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. *IEEE International Conference on Signal Processing and Communications*, 9 Suppl 2(November):65–68, 2007.
- [39] William R. Mark and Donald Fussell. Real-time rendering systems in 2010. In *ACM SIGGRAPH Courses*, page 19, 2005.
- [40] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

- [41] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 235–246, 2010.
- [42] Aaftab Munshi. The OpenCL Specification (Version 1.2), 2011.
- [43] 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 International Symposium on Microarchitecture, MICRO '44*, pages 308–317, 2011.
- [44] NVIDIA. NVIDIA GeForce GTX 480. http://www.nvidia.com/object/product_geforce_gtx_480_us.html.
- [45] NVIDIA. NVIDIA Quadro FX 5800. http://www.nvidia.com/object/product_quadro_fx_5800_us.html.
- [46] NVIDIA. NVIDIA GeForce 8800 GPU Architecture Overview, 2006.
- [47] NVIDIA. NVIDIA Compute PTX: Parallel Thread Execution ISA 1.1, 2007.
- [48] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide 1.0, 2007.
- [49] NVIDIA. Cuda community showcase. http://www.nvidia.com/object/cuda_apps_flash_new.html, 2010.
- [50] NVIDIA. NVIDIA CUDA C Programming Guide, 2011.
- [51] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [52] Pcchen. N-queens solver. In <http://forums.nvidia.com/index.php?showtopic=76893>.
- [53] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

- [54] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.
- [55] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH Courses*, page 268, 2005.
- [56] Iowa State University Reconfigurable Computing Laboratory. <http://rcl.ece.iastate.edu/>.
- [57] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucec Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *In 31st International Symposium on Microarchitecture*, pages 3–13, 1998.
- [58] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. In *BMC Bioinformatics*, volume 8(1):474, 2007.
- [59] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: a hardware architecture for ray tracing. In *Proceedings of Graphics Hardware*, pages 27–36, 2002.
- [60] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware*, pages 95–106, 2004.
- [61] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 1.2.1). <http://www.opengl.org/documentation/specs/version1.2/opengl1.2.1.pdf>, April 1999.
- [62] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 4.0), 2010.
- [63] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, pages 1–15, 2008.

- [64] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. AK Peters, Ltd., Natick, MA, USA, 2003.
- [65] ID Software. Quake. <http://idsoftware.com/games/quake/quake>.
- [66] J. Spjut, D. Kopta, S. Kellis, S. Boulos, and E. Brunvand. Trax: A multi-threaded architecture for real-time ray tracing. In *Proceedings of Application Specific Processors*, pages 108–114, 2008.
- [67] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28(1), 2009.
- [68] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.
- [69] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In *Proceedings of EUROGRAPHICS*, pages 153–164, 2001.
- [70] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.
- [71] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.
- [72] S. Woop, E. Brunvand, and P. Slusallek. Estimating performance of a ray-tracing ASIC design. *Symposium on Interactive Ray Tracing*, pages 7–14, 2006.
- [73] Sven Woop, Jrg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*, 24(3):434–444, 2005.