IMPROVING GPU SIMD CONTROL FLOW EFFICIENCY VIA HYBRID WARP SIZE MECHANISM

A Thesis Submitted
to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Electrical and Computer Engineering
University of Saskatchewan

by

Xingxing Jin

Saskatoon, Saskatchewan, Canada

Permission to Use

Abstract

High single instruction multiple data (SIMD) efficiency and low power consumption have made graphic processing units (GPUs) an ideal platform for many complex computational applications. Thousands of threads can be created by programmers and grouped into fixed-size SIMD batches, known as warps. High throughput is then achieved by concurrently executing such warps with minimal control overhead. However, if a branch instruction occurs, which assigns different paths to different threads, this warp will be broken into multiple warps that have to be executed serially, consequently reducing the efficiency advantage of SIMD.

In this thesis, the contemporary fixed-size warp design is abandoned and a hybrid warp size (HWS) mechanism is proposed. Mixed-size warps are generated according to HWS and are scheduled and issued flexibly. Once a branch divergence occurs, split warps are squeezed according to the proposed algorithm, and warp sizes are down-scaled wherever applicable. Based on updated warp sizes, warp schedulers calculate the number of cycles the current warp needs and issue the next warp accordingly. As a result, hybrid warps are pushed into pipelines as soon as possible and more pipeline stages are overlapped. The simulation results show that this mechanism yields an average speedup of 1.20 over the baseline architecture for a wide variety of general purpose GPU applications.

This work also integrates HWS with dynamic warp formation (DWF), which is a well-known branch handling mechanism aimed at improving SIMD utilization by forming new warps out of split warps in real time. The warp forming policy is modified to better tolerate warp conflicts. Also, squeeze operations are added before a warp merges with other warps. The simulation shows that the combination of DWF and HWS generates an average speedup of 1.27 over the DWF-only platform for the same set of GPU benchmarks.

Acknowledgments

The work presented in this thesis would not have been possible without the help of many individuals. First, I would like to express my sincere appreciation to my supervisor, Dr. Seok-Bum Ko, for his valuable guidance and support during the last two years.

I would also like to thank my supervisor, Dr. Brian Daku, for his constant support of my research and graduate studies. Also, I would like to thank the members of my M.Sc. thesis committee for giving their valuable time to review my thesis.

I would like to thank Dr. Ron Bolton, Dr. Li Chen, Dr. Daniel Teng and Dr. Khan Wahid in the Department of Electrical and Computer Engineering for their valuable guidance and help during my graduate studies.

I would like to thank my colleagues in the VLSI lab, Liu Han, Cinnati Loi and Zhuo Wang, for their helpful advice and the fun we had. I would also like to thank my colleague Mike Rowe for his continuous support and constant encouragement during my research. In addition, I would like to thank other friends in the VLSI lab. It has been my honor to work with them.

Finally, I owe my family a big thanks for their moral support throughout the years.

This is the dedication to my loving family.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AES	Advanced Encryption Standard
AIB	Atomic Instruction Block
ALU	Arithmetic Logic Unit
BFS	Breadth-First Search
BS	Black-Scholes
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
DWF	Dynamic Warp Formation
DWS	Dynamic Warp Subdivision
en-DWF	Enhanced DWF
en-DWF&HWS	Integration of Enhanced DWF and HWS
GPGPU	General Purpose Graphic Processing Unit
GPGPU-Sim	General Purpose Graphic Processing Unit Simulator
GPU	Graphic Processing Unit
HWS	Hybrid Warp Size
ID	Identification
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
LIB	LIBOR Monte Carlo

LPS	3D Laplace Solver
LWM	Large Warps Microarchitecture
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MLP	Memory-Level Parallelism
MMX	MultiMedia eXtension
MSHR	Miss Status Holding Registers
MUM	Maximum Unique Match
NN	Neural Network
NQU	N-Queen Solver
NVCC	NVIDIA CUDA Compiler
PC	Programming Counter
PCI-E	Peripheral Component Interconnect Express
PDOM	Immediate Post-Dominator
PDOM&HWS	Integration of Immediate Post-Dominator and HWS
PTX	Parallel Thread Execution
RAY	Ray Tracing
SBI	Simultaneous Branch Interweaving
SDK	Software Development Kit
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data

SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Data
SM	Streaming Multiprocessor
SMP	Symmetric Multiprocessor
SP	Streaming Processor
SSE	Streaming SIMD Extensions
STO	Store GPU
SWI	Simultaneous Warp Interweaving
TBC	Thread Block Compaction
TID	Thread Identification
TLP	Thread-Level Parallelism
VP	Virtual Processor
WST	Warp Split Table

1. Introduction

Although Moore's law has continued to drive smaller semiconductor devices, the difficulty of clock rate scaling and the limitation of uniprocessor performance scaling have forced the computing industry to switch to parallel hardware and software [1]. This migration is further fueled by rapidly growing computing demand. To date, the parallel computing landscape has been greatly extended, including various architectures with a range of core counts and various optimized memory systems.

The transition to parallel computing has coincided with the evolution of graphics processing units (GPUs) from special purpose devices to general purpose programmable cores [2]. Furthermore, driven by graphics applications' enormous requirement for computation and bandwidth, GPUs have grown as the dominant parallel architecture available for many computational applications.

To leverage the huge development cost of parallelizing general purpose applications, several new programming models have been created such as CUDA [3], OpenCL [4] and a growing set of familiar programming tools. These languages implement the single instruction multiple thread (SIMT) model and specify enormous parallel threads running on SIMD cores, which allow programmers to perform fine grained code level design by explicitly specifying thread behaviors.

For applications which can be greatly parallelized, containing only simple control structures and few dependency hazards (like graphics applications, for example), the performance is maximized since most threads take the same path during the program, and the SIMD cores can be fully utilized. However, for many other applications which have a considerable number of branch instructions and irregular memory access patterns, threads in a warp will take different paths and induce *branch divergence* problem, which will significantly affect SIMD performance.

This thesis proposes and evaluates a novel branch handling mechanism, hybrid warp size (HWS), aiming to tackle the branch divergence problem and further improve SIMD control flow efficiency. Although the implementation is based on NVIDIA GPU devices, it is correspondingly applicable to other SIMD architectures.

The rest of this chapter describes the motivation behind this thesis, the contributions of this work and gives the outline of the remaining chapters.

1.1 Motivation

Thread level parallelism (TLP) when compared with instruction level parallelism (ILP) is becoming the dominant technique to satisfy increasing computation demand, as single thread performance improvement slows. Intel's Larrabee [5], IBM's Power7 [6], NVIDIA's Tesla GPU [7] and AMD's Fusion APU [8] all employ TLP in various ways. These devices typically implement TLP within graphic processing units (GPUs). GPUs have become the dominant parallel architecture in these devices because of GPUs' significant computational power, large bandwidth and high energy efficiency.

GPUs are characterized by numerous simple yet energy-efficient computational cores, thousands of simultaneously active fine-grained threads and large off-chip memory bandwidth [1]. Thousands of threads created by programmers are grouped into fixed-size single instruction multiple data (SIMD) batches, known as warps. Generally, warp size is equal to or a multiple of SIMD width. Correlated threads within a warp execute the same instruction in sequence on different registers in parallel. This organization amortizes the overhead of instruction fetch and decode, and therefore, more processing units can be integrated onto a single chip. Contemporary GPUs employ the fine-grained multithreading organization, to hide stalls that arise from long-latency operations. When any thread within a warp experiences a long-latency operation or a data hazard, the entire warp is stalled. However, other warps that are ready to be executed will be issued to the pipelines. Multiple warps will occupy pipelines concurrently and the throughput loss will be reduced. For example, for NVIDIA's GPUs, the latency of read-after-write dependencies is approximately 24 cycles. If there are more than 192 active threads (8 GPU cores per multiprocessor $\times$ 24 cycles of latency $=$ 192 active threads, or 6 interweaved active warps of size 32), the latency can be completely hidden through this multithreading technique [9].

GPUs have tremendously accelerated many applications. For example, up to February 2012, NVIDIA had listed 1287 GPU applications with 214 of these applications obtaining a speedup of 50 or more and 135 of the 214 obtaining a speedup of 100 or more [10]. However, there are many applications that can achieve only limited performance improvement or no improvement at all. One major barrier to performance improvement is branch divergence.

The SIMD organization saves control overhead and increases computation density. However, when a branch instruction is executed within a warp resulting in different paths for different threads, this warp will be broken into multiple warps which have to be executed serially. Meanwhile, warp occupancy will be decreased and throughput will be reduced significantly. Figure 1.1 shows warp occupancy for a set of general purpose applications. The weight of the branch instructions is also shown. The warp size here is set to 32. Each stacked bar represents an application. Within each bar, every block indicates the percentage of cycles corresponding to a certain number of active threads. The figure shows that benchmarks BFS [11], NN [12], MUM [13], LPS [14] and NQU [15] (see table 5.2 on page 48) have relatively higher numbers of under-filled warps. Meanwhile, they all have comparatively more control flow instructions. This indicates that control flow intensive applications will more likely suffer from branch divergence leading to more idle computation resources.

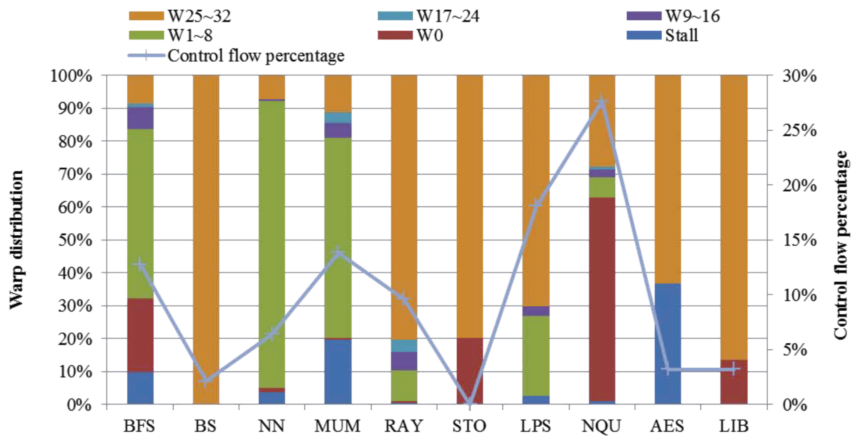


Figure 1.1 Warp occupancy and percentage of control flow.

1.2 Contributions

This thesis makes the following contributions:

1. It proposes a novel mechanism to overcome throughput loss due to branch divergence. It abandons current fixed-size warp design and introduces a hybrid warp size (HWS) mechanism. Warp size is set dynamically by hardware with the aim of achieving as high a throughput as possible.

2. It combines HWS and dynamic warp formation (DWF), a well-known technique that deals with the GPU control flow issues. Unlike DWF, it introduces a new squeeze algorithm to make individual warps denser before combining them with other warps. Meanwhile, it modifies the pattern of warp formation to better tolerate warp conflicts.

3. It gives a theoretical method to estimate throughput loss due to low warp occupancy and, furthermore, approximates the potential room for performance improvement.

4. It analyzes the relationship between the performance improvement brought by HWS and the branch instruction weight of the application.

1.3 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 discusses related work done by other researchers. Chapter 3 provides the essential concepts of parallel processor organizations and the baseline GPU microarchitecture. Chapter 4 describes the proposed hybrid warp size mechanism and outlines the integration with DWF. Chapter 5 describes the methodology of this work, including the simulator used, system configurations and benchmark properties. Chapter 6 describes the simulation results. Chapter 7 summarizes this thesis and suggests possible future work.

2. Related Work

This chapter discusses earlier work done by other researchers that is related to this thesis. Section 2.1 discusses the conventional hardware and software methods to deal with the SIMD control flow issue. Section 2.2 compares some recent proposals that involve warp optimizations.

2.1 SIMD Control Flow Handling

This section discusses the conventional methods that are used to overcome the SIMD control flow issue. Branch predication, the first method discussed below, is a basic and widely used mechanism found in almost every modern GPU device. Reconvergence mechanisms, the second topic of discussion, are used to improve performance by merging split branches whenever possible. Of the several variations of reconvergence mechanisms, two are examined. One of them inserts a $JOIN$ instruction into the original program based on the control flow analysis and merges the split branches at the point of the inserted instruction. Another dynamically creates new kernels according to the branch condition and manages the sub kernels by an interprocessor. Branch divergence elimination, the third topic discussed, is an alternative approach that prevents any occurrence of branch divergence.

Branch Predication

The most popular method used to overcome the control flow issue is a technique called branch predication. It derives from guarded instruction [16] and exists in most modern GPUs [3, 17]. When a branch instruction is applied to SIMD cores, a set of predications or masks are used to manage the multiple branches. When using branch predication, none of the instructions whose execution depends on the evaluation condition gets skipped [3]. Each thread is associated with a condition code or *predicate* that depends on the evaluation of the controlling condition. Every instruction is scheduled, but only those with true predicates are actually executed. Instructions with false predicates do not evaluate memory addresses or read operands, and no results are written back [3]. For subroutine calls, branch predication works in a similar way. It is worth noting that these predicates are organized into stacks so that nested branches can work. This mechanism dynamically controls the different branches, but it is not efficient. If there is a single path which no threads execute, SIMD cores will still schedule the instructions of that path and finish them, a procedure which yields nothing and wastes hardware resources, especially for those programs with long branch paths.

Reconvergence Mechanisms

Although branch predication solves the branch divergence problem, it is not efficient. It works well with short paths, but creates significant performance loss for long branches. Also, it cannot eliminate input dependent loops. Once branches are generated, they will never be merged even if they have a chance to converge. Lorie and Strong [18] invented a method to converge the split branches back based on the control flow analysis during the compile period. They introduced two instructions that could be inserted into the original program to facilitate convergence: $JOIN$, $ELSE$. The execution of $JOIN$ causes all processors waiting for the current block execution to be activated, while the execution of $ELSE$ results in the change to the current block order to be minimized [18]. By enforcing the priority ordering, the program will converge the split branches at the point where the $JOIN$ instruction is inserted.

Another similar technique called *conditional streams* is described in [19]. A conditional stream is a data stream that is accessed conditionally based on a case value. It can create multiple kernels from a single kernel according to the branch condition. The generated kernels communicate with each other through inter-kernels managed by an interprocessor. Once each generated kernel is executed by SIMD cores, they will be combined into a single data stream. This method requires an interprocessor to manage conditional switching and load-balancing, and a number of buffering registers are needed for the inter-partition communications. The overhead of creating extra kernels is also unavoidable.

Branch Divergence Elimination

In some cases, branch divergence can be eliminated in a relatively simple way to reduce performance loss. In their ray processing unit, Woop *et al.* [20] introduced a compound branch instruction which pairs a regular branch instruction with an arithmetic instruction. The compound branch instruction uses a mask to select a subset of the results of all the processing elements and performs an *and* or *or* operation to derive the final branch condition. Since each branch has the same outcome, no branch divergence exists anymore. This reduces the occurrence of jumps and fits for ray processing. However, this does not fit the most general purpose applications.

Another method was introduced by Krashinsky *et al.* [21]. They proposed a *vector-thread* architecture, which is characterized by two fetch units within each *virtual processor* (VP): *vector-fetch* and *thread-fetch*. A *vector-fetch* command can issue *atomic instruction blocks* (AIBs) to all VPs in a similar way used by conventional vector machines. On the other hand, a *thread-fetch* command allows a VP to request its own AIBs and thereby branch to its specific path. All VPs work concurrently. As a result, no branch divergence exists in this organization. However, the instruction bandwidth will be decreased because of the increased number of instructions. In addition, each VP requires an extra control logic and two fetch units, which will significantly raise the complexity of the entire system and require more hardware resources.

In general, branch predication and reconvergence mechanisms are fundamental techniques and employed widely, but the throughput loss between the divide point and the merge point is still a problem, especially for control-intensive applications. By integrating with HWS, they can eliminate the idle cycles during that period and improve the entire performance. Branch divergence elimination only works under limited circumstances or demands a considerable hardware overhead. Complete branch divergence elimination is challenging, but further reducing the undesirable impact of branch divergence is achievable.

2.2 Warp Involved Methods

Recent work on GPU branch divergence has focused on mapping threads to warps. Among the proposed methods, dynamic warp formation (DWF) is the most popular one. Thread block compaction is the evolved version of DWF, aiming to solve the increased memory access issue and the *starvation eddy* [22] problem. Dynamic warp subdivision is another method that is orthogonal to DWF and focuses on memory level parallelism. Simultaneous branch and warp interweaving is a recent issued mechanism that enables dual instruction issuing within each SM. Veynu *et al.* [23] proposed large warps microarchitecture (LWM) and suggest fewer larger warps with dynamic subwarp scheduling. All of them improve SIMD efficiency in some respect, but at the same time introduce related side effects.

DWF

DWF aims to address the under-utilization of the SIMD resources caused by branch divergence. It increases the throughput by regrouping threads with the same PC and filling the holes in the split warps. However, it also brings additional memory divergence since DWF regroups threads into new warps which no longer contain consecutive threads. In addition, SIMD resources are still under utilized due to idle quarter-warps. HWS addresses this by downscaling the warp size before issuing and pushes only the effective quarter-warps into the pipelines. Therefore, throughput loss due to idle quarter-warps is eliminated. Another disadvantage of DWF is the relatively high warp conflict rate. According to DWF, two warps are able to merge only if none of the 32 lanes conflicts. This condition is strict but can be loosened. HWS proposes that as long as the total number of the threads in each lane is less than 4, the two warps can be merged. In all, HWS is orthogonal to DWF and these two methods can be combined and yield better SIMD efficiency.

Thread Block Compaction

Thread Block Compaction (TBC) [22] evolves from DWF and aims to address two pathologies of DWF. The first is the increased memory divergence, as mentioned in the last paragraph. The second is called *starvation eddy* [22]. It occurs when two branches from a single node acquire uneven workloads and the faster one executes through the merge point without waiting for another branch. This is universal for most applications. TBC enables warps within a thread *block* (the measurement unit of threads issued to SMs) to share a block-wide reconvergence stack for branch divergence handling instead of separate *per-warp* stacks. With this more flexible thread compaction mechanism, control flow locality can be explored fully, and therefore, additional memory accesses due to DWF can be reduced. With the use of the block-wide reconvergence stack, all the warps that will eventually arrive at the reconvergence point will be recorded and synchronized using a warp barrier, and, as a result, *starvation eddy* will be eliminated. On the other hand, the extended overhead of regrouping threads due to TBC is a problem, especially for applications with a large number of divergent threads. This may counteract the benefit of TBC and increase the execution time for the entire block. Furthermore, idle quarter-warps still take a considerable weight of the entire executed warps according to the simulation results of TBC. HWS offers a solution by dynamically scaling warp size for TBC as it does for DWF. Also, the modification of the merge pattern to DWF could be applied here and produce fewer and denser warps.

Dynamic Warp Subdivision

Dynamic Warp Subdivision (DWS) [24] focuses on memory level parallelism (MLP). Compared to current GPU architecture, DWS allows a single warp to occupy more than one slot in the scheduler. Upon branch divergence, a warp will be divided into two warp-splits. Conventionally these two warp-splits have to be executed serially due to the per-warp reconvergence stack, while for DWS, these two warp-splits can be issued concurrently by two scheduler slots. Consequently, memory latency can be hidden. It works in a similar way for memory divergence. One split-warp represents threads that are not stalled by memory latency, while the other one represents the threads that are still stalled. The former one can run ahead and potentially prefetch data that may also be needed by the slower threads. To support DWS, the *stack based reconvergence* implementation has to be modified. Upon warp subdivision, the reconvergence stack remains untouched to avoid enforcing certain execution orders to the branches. As a result, no current and future nested branch information is recorded by the stack. Instead, this information is stored in an additional structure called a *warp-split table* (WST). Each entry in the table represents a warp-split and includes multiple zones recording the warp-split's parent warp ID, the next PC, the active mask and the status. The multiple scheduler slots select warp-splits according to the WSTs and interweave the execution of them (the next warp-split will be issued once the current one is stalled). Even though DWS improves the memory latency hiding, it does not increase the SIMD pipeline utilization. In addition, the hardware overhead due to the extra scheduler slots and WSTs has to be carefully measured against the performance gained through DWS. HWS is another method of warp interweaving which pushes another warp during the idle cycles of the current warp. HWS increases the SIMD utilization by filling the holes in current cycle, while DWS only issues another warp-split after the current warp-split is stalled for some reason.

Simultaneous Branch and Warp Interweaving

Simultaneous branch and warp interweaving was proposed by Nicolas *et al.* [25] very recently. The main advantage is to allow two distinct instructions to be issued to disjointed subsets of the same row of SIMD cores instead of one single instruction, while carefully considering hardware overhead has to be considered carefully. They proposed two complementary techniques: simultaneous branch interweaving (SBI) and simultaneous warp interweaving (SWI). Similar to DWS, SBI enables an instruction scheduler to interweave the execution of instructions from different branches. However, SBI eliminates the constraint that only one instruction is issued and executed at any time. To support this, stack based reconvergence is not feasible anymore; instead, thread frontier based reconvergence [2] is adopted. It works by always scheduling the warp-split of the minimal PC. Even though thread frontier based reconvergence serializes the execution of divergent branches, it is amenable to parallel execution by relaxing the scheduling constraints [2]. SBI doubles the warp size to 64 and duplicates the instruction buffer, instruction decoder and register file. More importantly, a secondary instruction scheduler is added to enable the issuing of dual instructions. SWI complements SBI by scheduling other warps in the gaps left by the first scheduled warp. The secondary scheduler will feed instructions from another warp to the first issued warp as long as no lane conflict exists. In other words, the active masks of the two scheduler have no overlap. These two mechanisms work in two levels and can be integrated to extend the instruction throughput as much as possible, however a number of issues still need to be considered carefully. The first one is the hardware overhead. The architecture is similar to the multiple-issue SIMD organization. The control logic is nearly doubled for each SM. Additional memory spaces are also a considerable factor. The second issue is the cooperation pattern between the primary scheduler and the secondary scheduler, for example in the dynamic selection between the branch-level parallelism and the warp-level parallelism.

Large Warps Microarchitecture

Veynu *et al.* [23] suggest forming fewer but correspondingly larger warps and dynamically creating packed SIMD-width sized sub-warps from the active threads in a large warp. This leads to improved SIMD resource utilization in the presence of branch divergence. Similar to TBC, LWM relies on coarser scheduling units and a wider regrouping range. Sub-warp formation always occurs during the entire process, even for warps with no divergence, so LWM may degrade the speed of applications with few branch instructions. In addition, large warps may contain more branches and exacerbate idle periods imposed by branch divergence [26].

2.3 Summary

This chapter discusses earlier work done by other researchers on the branch divergence issue. The conventional methods such as branch predication make branch instructions viable on SIMD cores. The subsequent mechanisms further improve the performance by converging paths back together or eliminating divergence with extra hardware support. Warp related methods have been getting more and more attention in recent years and several methods have been proposed. This chapter analyzes some of these methods and discusses the advantages and disadvantages of them. Overall, HWS is orthogonal to the most recent and popular mechanisms and can be integrated with them to yield better SIMD efficiency without any modifications to the original programs.

3. Baseline GPU Architecture

This chapter provides the background of this work, including the essential concepts of parallel processor organizations, the difference between ILP and TLP, GPU computing model and streaming multiprocessor architecture. In addition, conventional branch divergence handling methods and DWF are also discussed.

3.1 Parallel Processor Organizations

Traditionally, the processor has been viewed as a sequential engine. Most programming languages specify algorithms as sequences of instructions. Processors execute machine instructions in a sequence one at a time. [27]. However, this view has never been completely true. As early as 1985 [28], the processors had already started to employ pipelining to overlap the execution of instructions, which is one of the techniques referred to as instruction-level parallelism (ILP). As computer technology evolved, many parallel architectures were developed. Based on Flynn's taxonomy [29], all parallel processor systems are classified into four categories, as shown in Figure 3.1:

1. **Single instruction single data (SISD):** a single processor executes a single instruction on data stored in a single memory. The typical example is a uniprocessor.
2. **Single instruction multiple data (SIMD):** a number of processing units execute the same instruction simultaneously on different sets of data stored in multiple processor associated memories. The typical example is a vector processor.

3. **Multiple instruction single data (MISD):** multiple processing units simultaneously execute different instructions on the same set of data. This organization has not been implemented, thus not appears in the figure.

4. **Multiple instruction multiple data (MIMD):** a set of processors simultaneously execute independent instructions on separate data sets. The typical example is a symmetric multiprocessors (SMP).

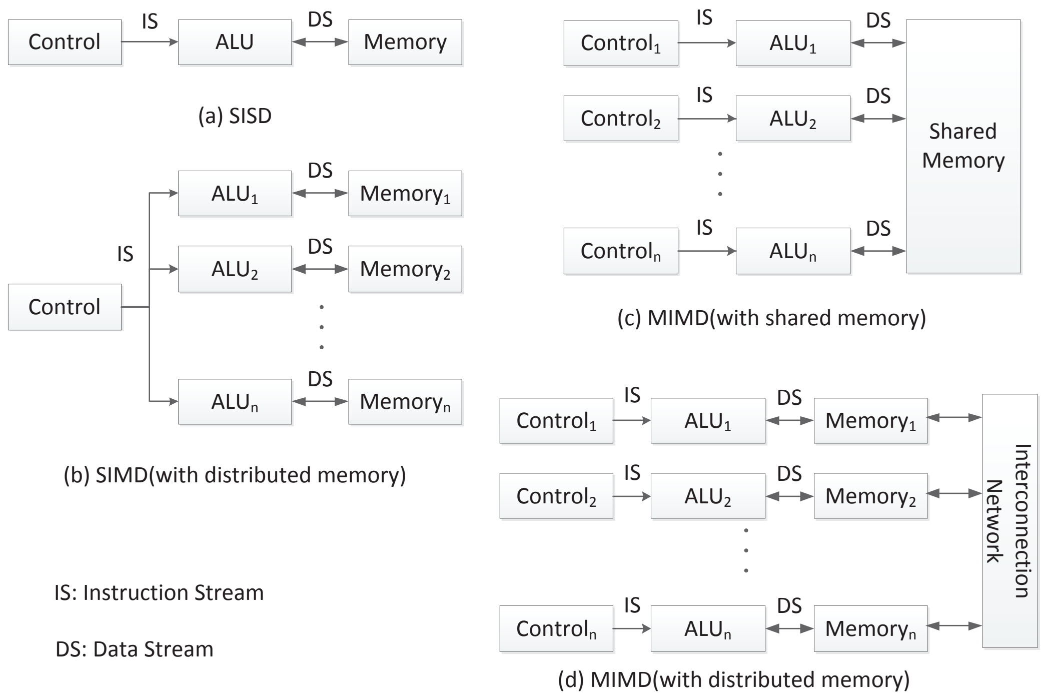


Figure 3.1 Parallel processor organizations.

SISD is the most conventional organization used. Machines with this architecture have only one instruction stream and only one data stream as shown in Figure 3.1(a). Another very common structure is MIMD. In recent years, multiprocessors have dominated the personal computer market. The organization they adopt is MIMD, which generally has multiple instruction streams and multiple data streams, as shown in Figures 3.1(c) and 3.1(d). Based on the memory structure, MIMD can be further divided into two categories [30]. The first uses shared memory among all processing units like Figure 3.1(c). The second omits shared memory, but locates a local memory for each processing unit called distributed-memory MIMD organization, as shown in Figure 3.1(d). All MIMD architectures have a common feature: they all build multiple control units associated with every processing unit. However, control units take a lot of room in a chip. This is one of the reasons that a multiprocessor with MIMD organization is not capable of integrating many processing units onto a single chip. To solve this problem, SIMD is introduced. The virtue of SIMD is that all parallel processing units share a single instruction stream but operate on different registers, as shown in Figure 3.1(b). For example, a single SIMD instruction may accomplish a vector addition of two sets of numbers within one execution cycle. The key advantage of SIMD is the reduction in the number of control units. This makes the integration of thousands of cores possible.

The most common variation of SIMD exists in almost every microprocessor today, and is based on the hundreds of multimedia extensions (MMX) and streaming SIMD extensions (SSE) of the x86 microprocessor [31]. The main reason is to improve performance of multimedia programs. These instructions are compiled to run on many ALUs simultaneously or on many narrower ALUs partitioned from a single wide ALU. For example, a 64-bit ALU can be divided into two 32-bit ALUs or four 16-bit ALUs or eight 8-bit ALUs as the extended SIMD instructions require.

Another variation of SIMD, which was also the first use of SIMD, is the vector architecture. It was popularized by Cray Computers [32], which built the fastest computers at the time. The vector architecture is featured in vector instructions, which are fetched and decoded by a shared unit, but operate on multiple ALUs with different vector elements. Other than scalar instructions, vector instructions use data-level parallelism extensively and greatly reduce the instruction fetch and decode bandwidth. The use of memory is also more efficient due to the predeterminable memory access pattern.

The GPU architecture is another major variation of SIMD organization. By e-quipping a computer with a GPU card, the graphics fraction of programs can be delivered to the GPU. As we know, the basic element of graphics processing is the pixel, and each pixel can be processed independently. To utilize this feature, the GPU has evolved to execute many threads corresponding to certain pixels in parallel. In recent years, GPUs have been improved enormously, especially driven by the rapidly growing game market. Even non-graphic applications have started to explore GPUs in order to accelerate processing. Many languages aiming to increase programmability have been invented, such as Brook [33], a streaming language for GPUs, and NVIDIA's CUDA, which enables programmers to write C programs on GPUs. The detailed GPU organization will be covered in 3.3 and 3.4.

3.2 Instruction-Level Parallelism and Thread-Level Parallelism

Instruction-level parallelism (ILP) is a method that overlaps instruction executions to improve the performance of a processor [28]. Many techniques exploiting ILP are employed in contemporary processors. The most common one is pipelining, but other techniques include branch prediction, out-of-order execution and data forwarding. In the last century, ILP was the key to achieve rapid performance improvements. However, the limitations of ILP have been getting more and more obvious in recent years. Designers have switched to the higher-level parallelism strategy: thread-level parallelism (TLP). A thread is a separate process with its own instructions and data. Each thread has all the program context to allow it to execute. Unlike ILP, which exploits implicit parallel operations within code sequence, TLP explicitly makes use of multiple threads of execution which are inherently parallel [28]. Furthermore, ILP and TLP can be employed at the same time, a technique called multithreading. Multithreading overlaps the execution of multiple threads on the sharing function units of a single processor. It can be divided into two classes. The first is fine-grained multithreading. In every cycle, threads that suffer from short or long stalls can switch to other free-to-go threads. This is efficient when threads frequently experience stalls. The baseline GPU architecture used in this work adopts this technique. The second class is coarse-grained multithreading, which switches threads only on long stalls. Since its performance is quite limited due to the frequent occurrence of short stalls, not many processors use it.

3.3 GPU Computing Model

Because of the wide use of NVIDIA's GPU devices in the general purpose computing field, this work adopts a similar computing model based on NVIDIA's CUDA programming model [3], as shown in Figure 3.2.

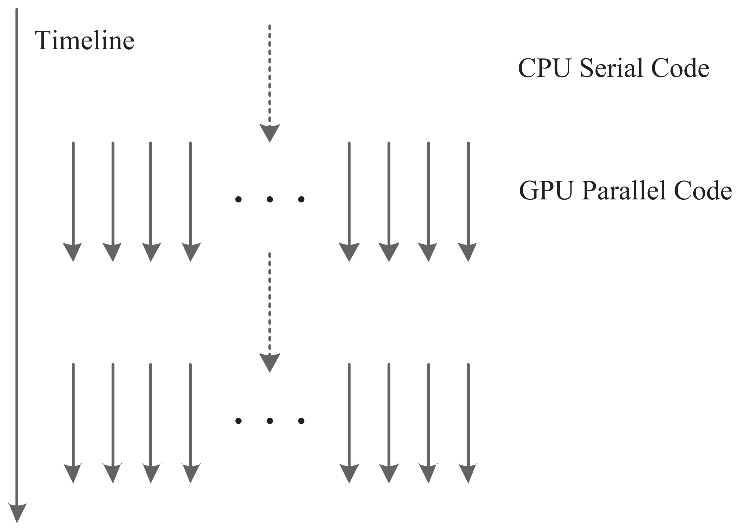


Figure 3.2 GPU computing model.

First, the program will be loaded into the CPU (also called the host). For CPU serial codes, the host will execute instructions in the traditional way. Once the GPU parallel codes are reached, the host will invoke the GPU (also called the device) and pass the parallel section to it. In the CUDA programming model, this parallel section is signaled by a kernel function. As an illustration, the following sample code performs vector addition [3]:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Every kernel function consists of an array of threads in a hierarchical pattern, as shown in Figure 3.3 [3]. Every kernel is mapped to a grid, which consists of multiple blocks organized as a maximum three-dimension array. Furthermore, each block is made up of a bunch of threads, which are also organized as an array with a maximum dimension of three.

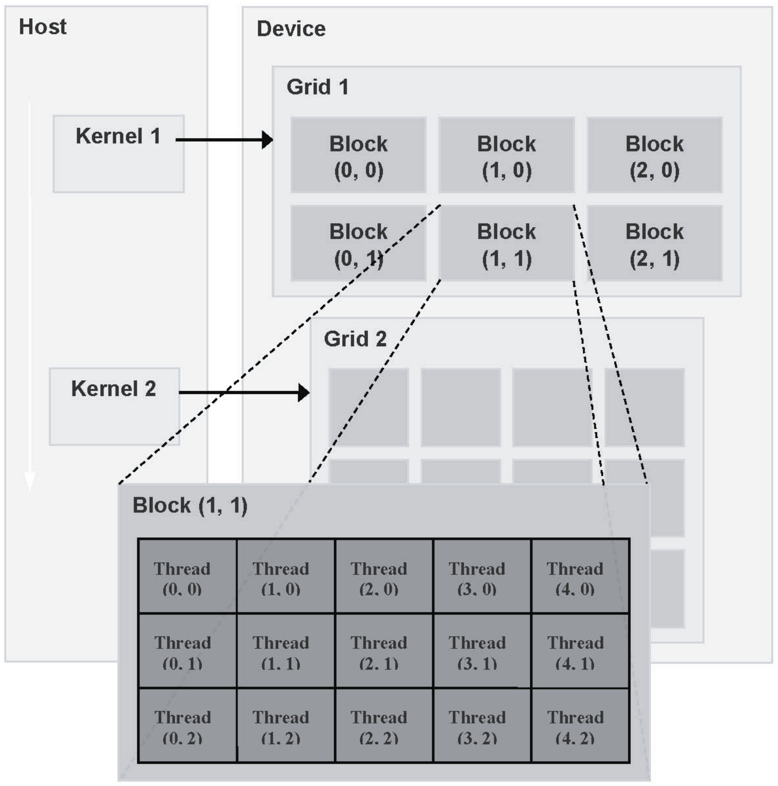


Figure 3.3 Thread hierarchy.

To facilitate a thread index, a built-in variable is introduced $threadIdx$. It is a 3-component vector and can be easily used to identify one-dimensional, two-dimensional or three-dimensional threads. In the example above, the kernel function $VecAdd$ is configured with N threads. These threads execute the same instruction: addition, but with different operands that are differentiated by the built-in variable $threadIdx.x$.

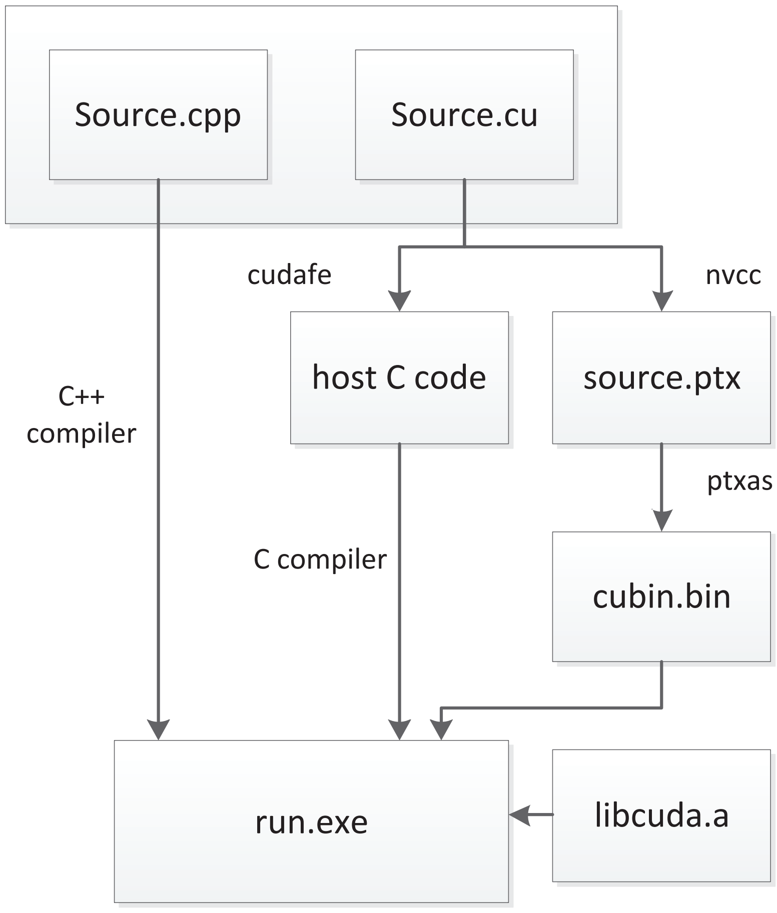


Figure 3.4 CUDA compute flow from the compilation view.

Figure 3.4 [34] describes the GPU compute flow from the compilation view. For different source codes, multiple compilers are used. The $Source.cu$ is a CUDA specified source file, which contains host C codes, as well as device C codes running in parallel on the GPU. These two groups of codes can be distinguished by $cudafe$. Then the host C code is compiled in the traditional way and device codes are compiled by $nvcc$ into parallel thread execution (PTX) assembly codes. Next, the PTX assembler compiles the PTX codes into GPU binary (labelled as ”cubin.bin” in Figure 3.4). Finally, all the host fragments and device portions, as well as the CUDA library (labeled as ”libcuda.a” in Figure 3.4), are lined together into a final executable program [3].

On the hardware side, the GPU is connected to the CPU through one or two PCI-E slots. Within the GPU, hundreds of processing cores are organized into a hierarchy. At the top level, the GPU is composed of an array of processors referred to as streaming multiprocessors (SMs) [3]. All SMs are connected to multiple memory modules through an interconnect network, as shown in Figure 3.5. The range of this work is within the scope of SM, which is further discussed in the next section.

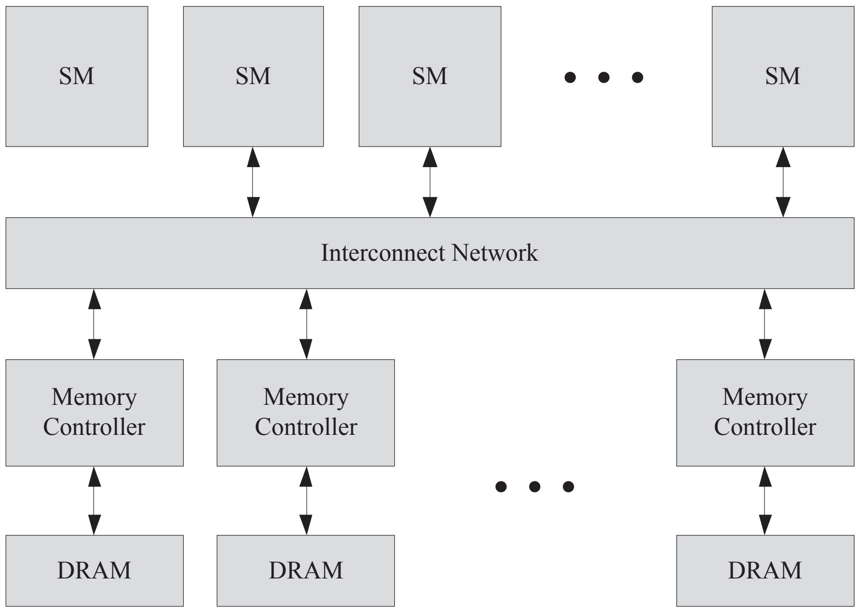


Figure 3.5 Baseline GPU architecture.

3.4 Streaming Multiprocessor Architecture

Figure 3.6 shows a single SM architecture [35]. It is mainly composed of a shared instruction fetch unit, an instruction decode unit, a highly banked register file and multiple ALUs. Before execution, individual threads are grouped into fixed-size warps [3], which are the granularity used for scheduling inside a SM. In the fetch stage, the scheduler selects a warp from the scheduling pool using a round-robin policy. Then the instruction cache is accessed and the instruction decode is performed. Next, multiple register values are read synchronously and then fed into the ALUs, where computation is finished in parallel. Once a warp reaches the final stage of the pipeline, it will be committed and put into the scheduling pool again for future scheduling. However, if any threads in a warp encounter a long latency operation (such as a DRAM access), the warp will be taken out of the scheduling pool until the warp is committed, and meanwhile other warps will be issued. As a result, long latency can be hidden and throughput loss can be reduced.

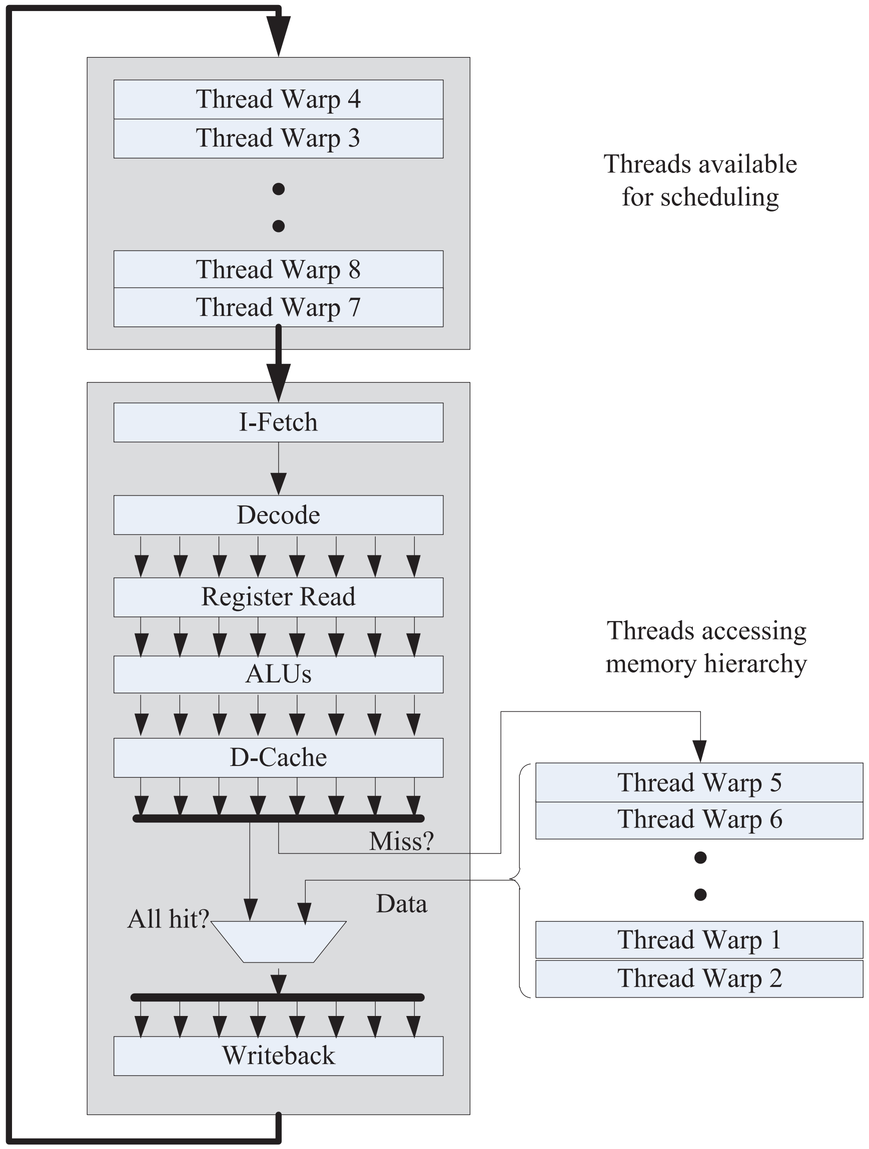


Figure 3.6 Streaming multiprocessor architecture.

3.5 Branch Divergence Handling

Branch divergence is a key issue for general purpose GPU applications. It occurs when threads within a warp take different paths. For equal length paths, an if-else branch instruction loses 50% efficiency. To facilitate user programming, contemporary GPUs allow threads to branch and execute independently, and therefore, threads with different paths can be directly serialized, as shown in Figure 3.7(a).

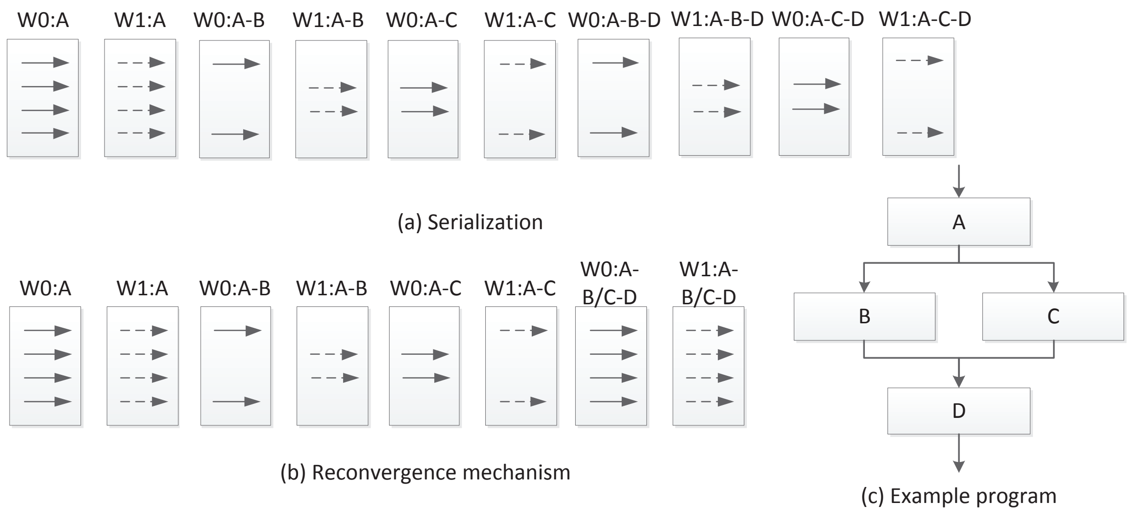


Figure 3.7 Conventional branch divergence handling.

Figure 3.7(c) gives the corresponding example program. Once the divergent point A is reached, the two warps $W0, W1$ are split into four fragments $W0 : A - B, W0 : A - C, W1 : A - B, W1 : A - C$. Next, the four segments continue executing till the end of the program, even though they have opportunities to converge (merge point D).

The serialization method is simple to implement, but not very efficient, thus most recent GPUs employ reconvergence mechanisms, as shown in Figure 3.7(b). The reconvergence point D is analyzed during the compilation period. Once code segments B and C finish, the threads in the fragmented warps go back to their original warps and carry on executing the rest of the instructions until another branch instruction is reached. This mechanism can be implemented by utilizing a branch synchronization stack which is used to manage independent threads that diverge and converge [27]. It should be noted that this reconvergence mechanism only functions within a warp. Different warps are independent in terms of branch handling.

The choice of the reconvergence point affects the overall performance. In this work, the immediate post-dominator [36] is chosen in the baseline GPU architecture according to standard practice. In graph theory, a node d dominates a node i if every path from the start node to i must go through d, written as d *dom* i. A node p is said to post-dominate a node i if all paths to the exit node starting at i must go through p. Similarly, the immediate post-dominator of a node i is the post-dominator of i that does not strictly post-dominate any other post-dominators of i. According to this definition, in the example program above, node D is the immediate post-dominator of A. It should be noted that the immediate post-dominators can be analyzed during compilation time.

With careful examination of the reconvergence process in Figure 3.7(b), further improvement can be made by noting that $W0 : A - B$ and $W1 : A - B$ execute the same program segment, and they both take half of the SIMD pipelines. Therefore, it is possible to merge them together and form a full warp, further saving execution time. This concept is the basic principle of a well-known branch handling method: dynamic warp formation (DWF).

3.6 Dynamic Warp Formation

Dynamic warp formation (DWF) is a well-known technique to improve GPU SIMD efficiency by forming new warps out of split warps in real time. During each schedule cycle, the thread scheduler will analyze all of the ready warps in the warp pool and try to form new warps. To combine two diverged warps, the following conditions must be satisfied: first, the two split warps must have an identical program counter, and second, no warp conflict can exist between the two warps. This occurs when two threads from different warps occupy the same scalar pipeline or lane. Since the register file is highly banked and every lane accesses one bank, if two threads access the same bank of the register file, bank conflict will occur and the two access operations must be serialized, which introduces a significant performance penalty. Figure 3.8 compares the warp structure under the PDOM mechanism and the DWF mechanism for the same example program. Figure 3.8(b) shows that $W0 : A - B$ and $W1 : A - B$ satisfy the two conditions discussed previously, and therefore, they are integrated into a new warp $W2 : B$. Similarly, $W0 : A - C$ and $W1 : A - C$ are combined into $W3 : C$. As a result, the throughput loss is eliminated for this example program.

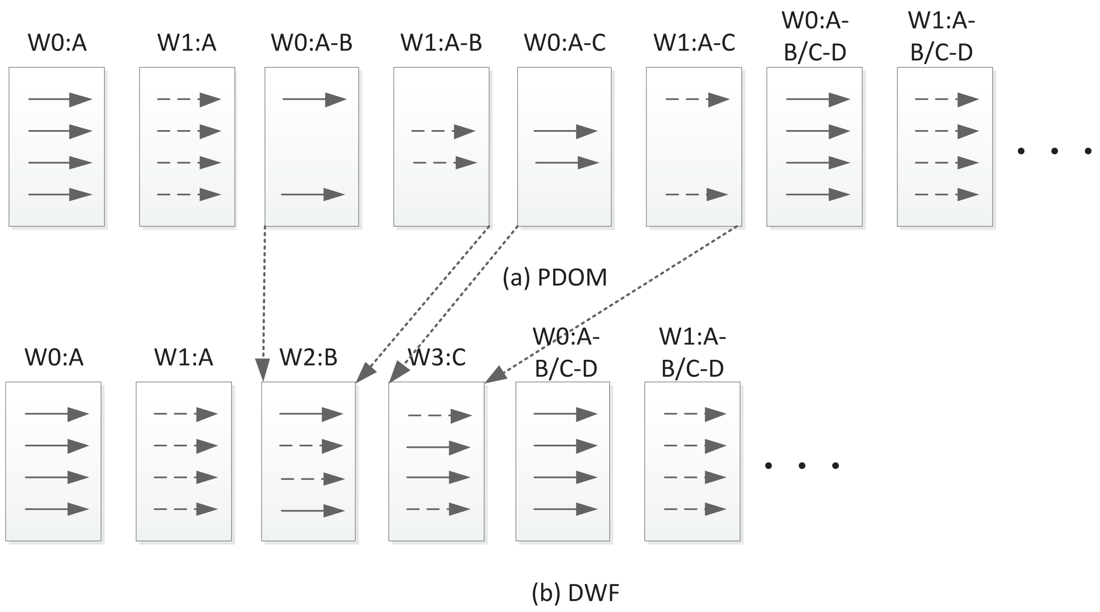


Figure 3.8 Comparison of PDOM and DWF.

DWF's use of the complementary scalar pipelines of two diverged warps introduces the following concern. Without DWF, thread IDs within a warp are logically consecutive. In other words, the difference between the smallest ID and the largest ID is less than warp size. Therefore, each thread's registers are at the same offset within the bank, and thus, one address decoder is sufficient. After introducing DWF, threads within a warp may come from different split warps, so the thread IDs cannot be guaranteed to be consecutive anymore. The offsets for different threads will vary. The register file has to be changed and equipped with multiple decoders for each bank.

The efficiency of DWF heavily depends on the number of threads with the same PC value. Suppose there are N threads with N different PCs, then the only warp structure is one active thread in each warp. DWF loses functionality in such a situation. To avoid this, all threads should have a similar rate of progress [35]. Thus, the warp scheduling policy needs to be carefully designed. According to [37], majority scheduling policy is the best one compared to other policies such as minority, time s-tamp, post-dominator priority or program counter priority. The majority policy tries to pick up the warps which have the most common PC in the warp pool and will continue to issue warps at this PC before switching to the second most common PC. Further discussion about DWF's performance will be presented in Chapter 4.

3.7 Summary

In this chapter, the GPU baseline architecture is described based on NVIDIA devices of compute capability 1.x. However, the method discussed in this thesis can be extended to other SIMD architectures. This chapter also describes the streaming multiprocessor architecture and explains how ILP and TLP are employed through fine-grained multithreading. Then conventional branch handling methods are discussed, emphasising reconvergence mechanisms. Finally, we introduce another advanced method, DWF, and explain how performance is improved by forming new warps.

4. Hybrid Warp Size Mechanism

The PDOM reconvergence mechanism ensures that resources are fully utilized after the reconvergence point, but the SIMD pipelines are still under-utilized between the divergence point and the reconvergence point. In this chapter, the HWS mechanism proposed to deal with this issue is described.

A contemporary SM creates, manages, schedules and executes threads in groups of 32 parallel threads called a warp. This fixed-size warp design simplifies hardware design, especially the warp scheduler. Every 4 cycles (32 threads ÷ 8 SIMD pipelines = 4 cycles), a warp that is ready for execution is selected by the warp scheduler and issued to the SIMD pipelines using a round-robin policy. This keeps the SIMD pipelines running at a uniform pace. However, when a warp meets a branch instruction and diverges into two separate warps, warp occupancy will deteriorate, as indicated in Figure 4.1(a). In the figure, each number represents an active thread within a warp, each letter represents a warp and each column represents an execution cycle. Figure 4.1(c) describes the corresponding program with branch divergence. For this example program, half of SIMD efficiency is lost due to this branch instruction.

The basis of the HWS mechanism is the dynamic adjustment of the warp size resulting in smaller warps to take fewer execution cycles. In this way, SIMD pipelines may start the next warp earlier and the throughput loss due to idle threads is reduced. This approach is shown in Figure 4.1(b), where the size of warp B is reduced to 8 threads, and the size of warp C is reduced to 24 threads. In this example, there are no resources wasted on idle threads and thus four execution cycles are saved.

SIMT GPUs are dramatically more efficient and flexible on control-intensive pro-

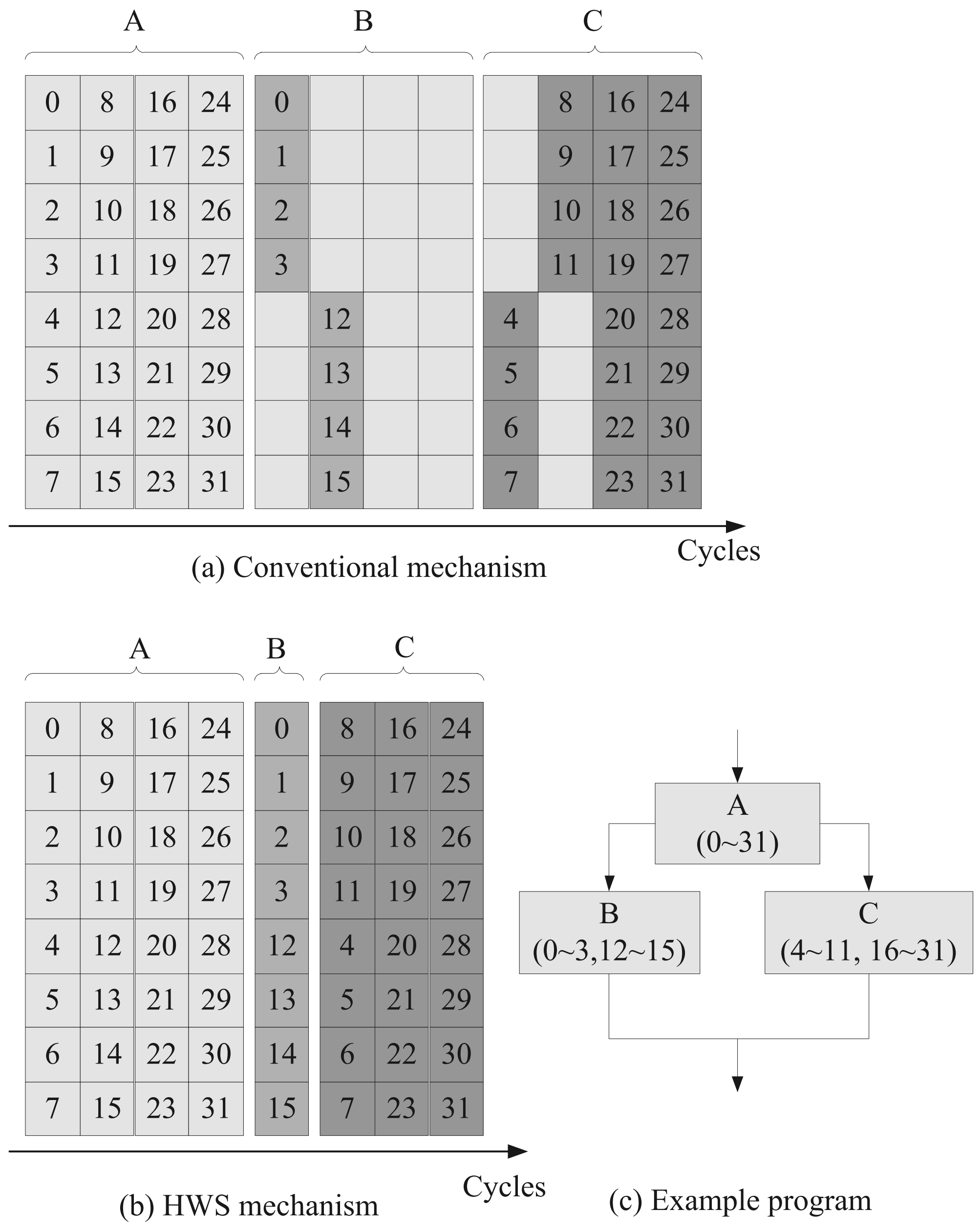


Figure 4.1 Branch divergence handling. (a) Conventional mechanism; (b) Hybrid warp size mechanism; (c) An example program. The numbers represent active thread IDs. The letters represent warp IDs.

grams than earlier GPUs, as their warps are much narrower than the SIMD width of prior GPUs [27]. The reason that HWS can bring performance improvement is the downsizing of the warp size on average. A suitable algorithm is needed to reduce the warp size and in this paper a squeeze algorithm is proposed, which is described in Section 4.1. The HWS warp scaling is discussed in Section 4.2. The variable warp size requiring a varying number of execution cycles must be handled by the warp scheduler and this is discussed in Section 4.3. The integration of HWS and DWF to yield better performance is discussed in Section 4.4.

4.1 Squeeze Algorithm

The main purpose of the proposed squeeze algorithm is to generate dense and small warps. In Figure 4.1, threads 12-15 are squeezed into the first quarter-warp, and therefore, the size of warp B is reduced to 8 threads. Similarly, warp C is reduced to 24 threads. The squeeze algorithm was designed with the following three objectives in mind:

1. Keep each thread in the same lane (represented by a row in Figure 4.1) to avoid bank conflicts when accessing the register file.
2. Make the modified warp as dense as possible.
3. Minimize movements to decrease the extra overhead and reduce additional memory divergence.

Based on these criteria, this work develops a squeeze algorithm, as shown in Figure 4.2. To minimize thread movement, the algorithm sorts in descending order the four quarter-warps by the number of active threads. Here the sorting operation is performed logically, not physically. No real thread movement is involved in this phase. The program just gives a quarter-warp ID to the proper quarter-warp according to the number of its active threads. For clarity, the IDs which the algorithm physically assigns to the four quarters are a, b, c and d. The next step is to take active threads in quarter 3 (the one with the fewest active threads) to feed corresponding holes (inactive thread places in the same lane) in quarter 0, then quarters 1 and 2. Next, the same operations are applied to quarter 1 and quarter 2. Even though sort operations increase the warp scheduler's complexity, they will reduce the amount of thread movements. In addition, they will ease the warp scaling operation, which is elaborated in Section 4.2. The conditional judgment in the diagram ensures no bank conflicts are generated due to squeezing. Therefore, all three criteria are satisfied.

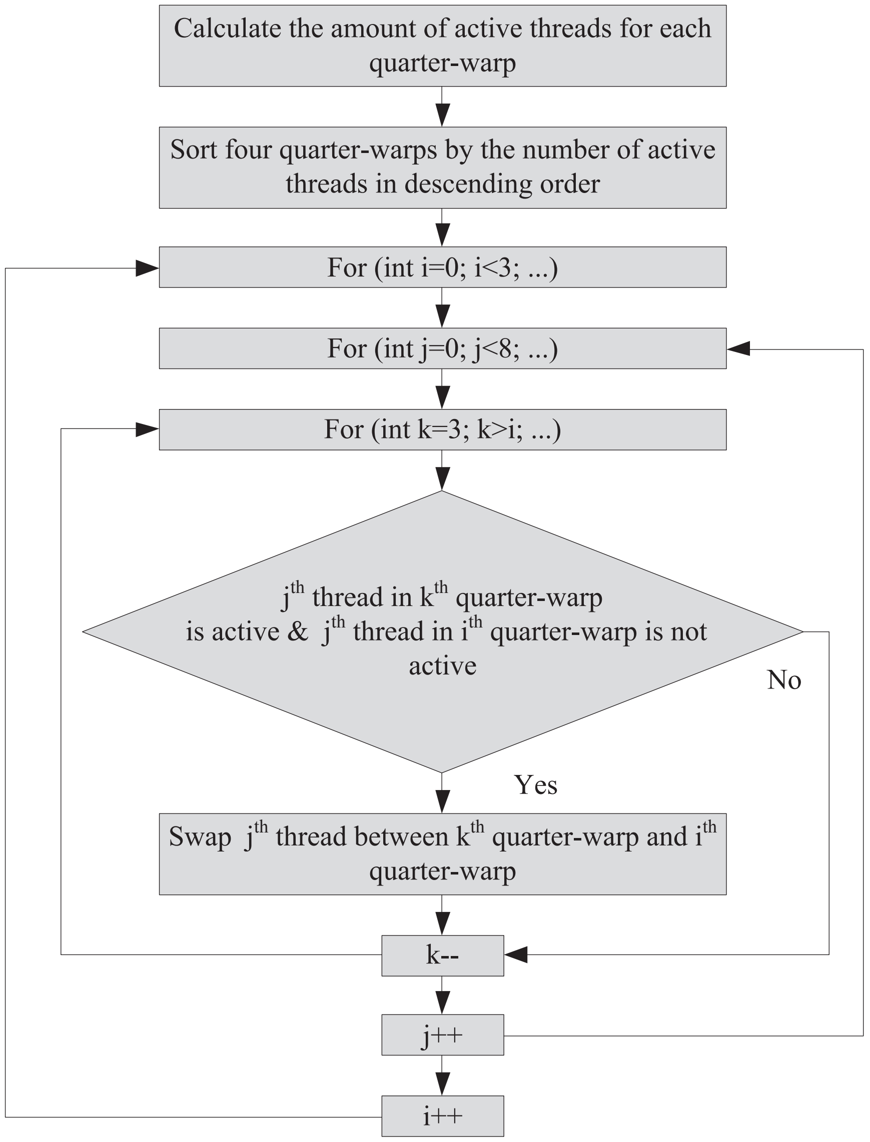

Calculate the amount of active threads for each quarter-warp
Sort four quarter-warps by the number of active threads in descending order
For (int i=0; i<3; ...)
For (int j=0; j<8; ...)
For (int k=3; k>i; ...)
jth thread in kth quarter-warp is active & jth thread in ith quarter-warp is not active
No
Yes
Swap jth thread between kth quarter-warp and ith quarter-warp
k--
j++
i++

Figure 4.2 Flow chart of the proposed squeeze algorithm.

Figure 4.3 describes how the squeeze algorithm compresses a warp step-by-step.

1) Sort the quarter-warps according to the number of active threads and label them #0, #1, #2 and #3, as shown in Figure 4.3.

2) Take the active threads from quarter-warp #3 to fill the holes in quarter-warp #0 making sure the threads are placed into a corresponding lane.

3) Similarly, take the active threads from quarter-warps #2 and #1 and move them to quarter-warp #0.

4) Repeat steps 2 and 3 using quarter-warp #1 as the recipient and then quarter-warp #2 as the recipient, if applicable.

In this example, the densest status was achieved after the second cycle. All of the active threads were squeezed into quarter-warps #0 and #1, and consequently, the next step, warp scaling can be simplified.

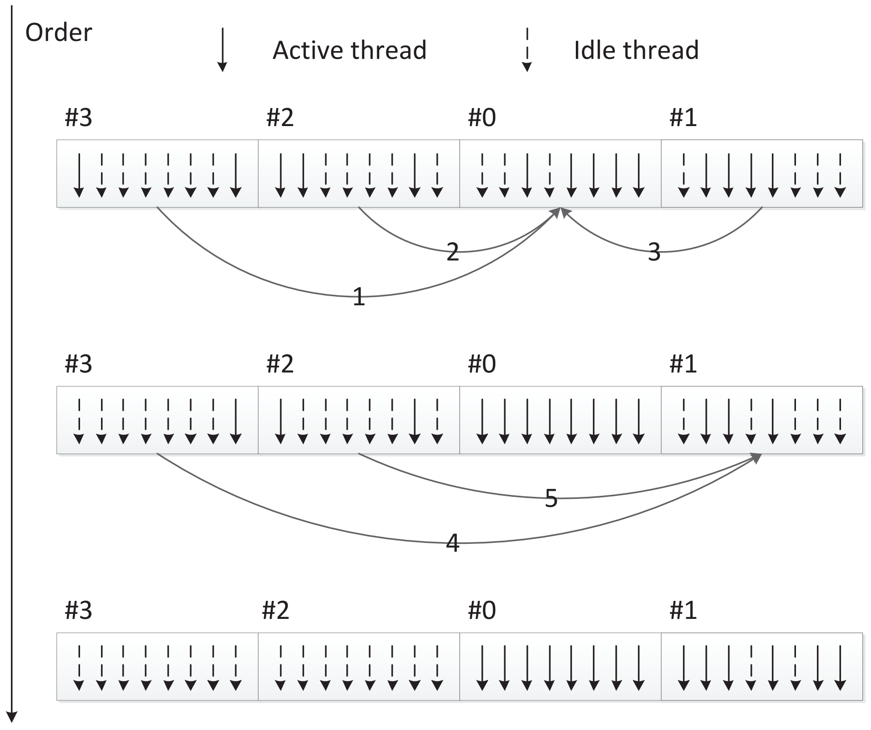


Figure 4.3 An example of the squeeze algorithm.

4.2 Warp Scaling

The squeeze algorithm compresses each warp, and therefore, increase the chance to downscale the warp size. The updated warp size has to be recorded to inform the issue logic about the length of the issue window. The following describes the detailed steps of this phase:

1) Create a quarter-warp mask word that contains four bits to indicate the status of each quarter-warp. A 1 indicates that the corresponding quarter-warp has at least one active thread, and a 0 corresponds to an idle quarter-warp. For the example in Figure 4.3, the mask is set to 0011, since quarter-warp #3 and quarter-warp #2 contain no active threads and quarter-warp #0 and quarter-warp #1 contain active threads. In this case, the warp size is set to 16 instead of 32.

2) Push the updated warp along with the quarter-warp mask into the corresponding entry in the warp pool.

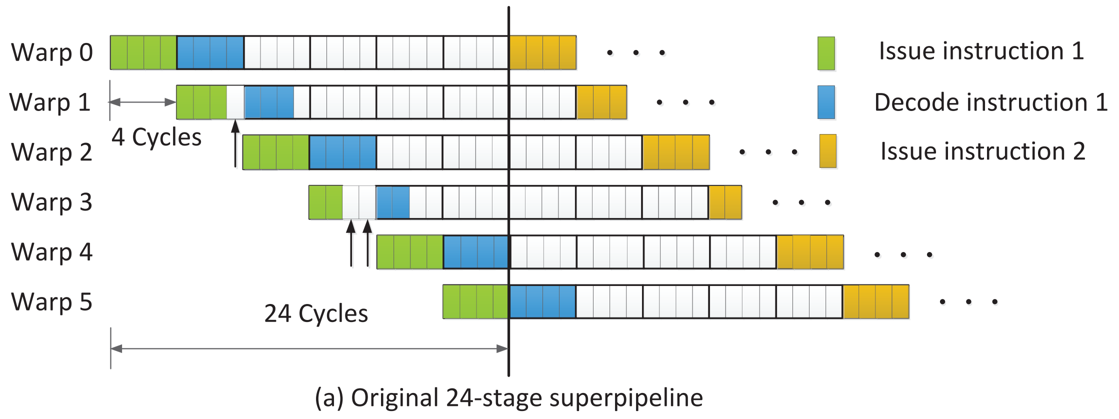
3) Pick up the largest warp size among the 28 standby warps from each SM and assign the new warp size to them.

Step 3 forces a synchronization of all the SMs. This requirement illustrates one limitation of the simulator we are using. For example, in this work 28 SMs are configured. If 27 of them have a warp size of 8 while the remaining one has a warp size of 32, the ultimate warp size for all the 28 warps will have to be set to 32. This disadvantage may be addressed in a future work.

Even though the hybrid warp size mechanism rearranges threads within a warp, it does not affect convergence at the merge point because the converged warp always has consecutive thread IDs.

4.3 Issue and Execution

Figure 4.4 compares the original pipeline with the HWS pipeline.



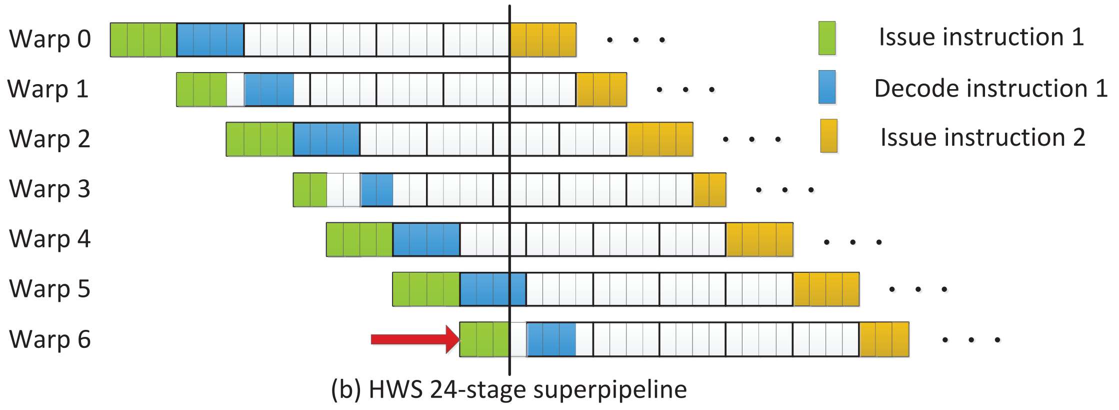


Figure 4.4 Comparison of the original 24-stage superpipeline and the HWS 24-stage superpipeline.

In this work, the pipeline of each SM is modeled as six logical stages (fetch, decode, execute, pre-memory, memory, write-back) with superpipelining of degree 4, as shown in the figure. Each warp is issued through four cycles, assuming the warp size is 32 and the SIMD width is 8. In Figure 4.4(a), the vertical arrows highlight a cycle when the issue is idle. This is due to the fixed warp size design. Conversely, HWS breaks the four-cycle rule and enables the scheduler to issue instructions even if the fourth issue cycle of the previous warp is not reached. This flexible issue mechanism guarantees that the instructions will enter into the pipelines as soon as possible, as shown in Figure 4.4(b). Except for the very first four cycles, the fetch stage and decode stage are both active for the subsequent cycles. Similar situations occur in the remaining four pipeline stages.

The read-after-write latency is 24 cycles, and 6 interweaved active warps (or 192 active threads) are needed to keep pipelines busy all the time, as specified in Chapter 1. In the example of Figure 4.4(a), the number of the active threads is 168 since 3 issue cycles are idle. HWS fills the gap by issuing the same instruction from the next warp, and therefore, when all of the instructions from warp 5 are issued, the 24-cycle latency is still not fully covered. HWS then issues another warp (indicated by the red arrow at the bottom). As a result, the pipeline is fully occupied by the 192 active threads from 7 warps. In all, three cycles are saved for the first instruction. In summary, for the HWS mechanism, the warp scheduler must extract the warp size information and calculate the next issue time, which varies between 1 and 4 cycles. Eliminating the fixed warp size will require extra hardware overhead.

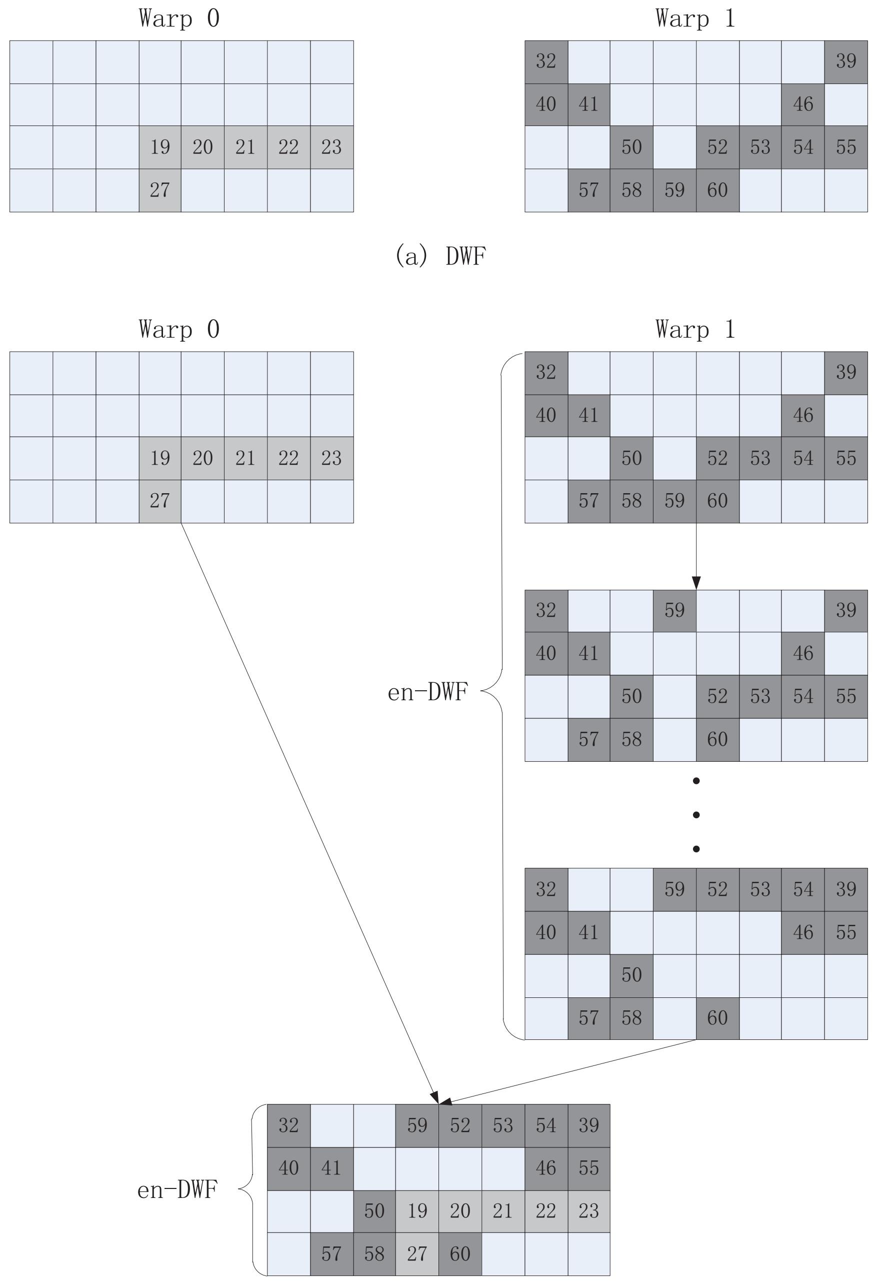
4.4 Integration of DWF and HWS

DWF assigns a thread to a warp only if that warp does not contain another thread in the same lane. However, each lane can hold a maximum of 4 threads for every warp (warp size 32 $\div$ SIMD width 8). This work modifies the DWF algorithm and assigns a thread to a warp as long as the number of active threads in that lane is less than four, as shown in Figure 4.5.

In Figure 4.5, even though threads 20-23 and threads 52-55 conflict, the threads 20-23 can be assigned to other quarter-warps of warp 0. The following describes the enhanced DWF algorithm, called en-DWF. For clarity, warp 1 is the incoming warp waiting to be regrouped with other warps and is called the male warp, while warp 0 is the selected warp from the warp pool offering the opportunity for combination and is called the female warp.

1) Starting from the first lane, scan each position to check if lane conflict exists (more than one active thread occupies the same position). Once a lane conflict is found, continue to the next step. For this example, there is no lane conflict

for the first three lanes. In the fourth lane, thread 27 and thread 59 are found in conflict.



(a) DWF

(b) en-DWF

Figure 4.5 An example of en-DWF.

2) Within the conflict lane, check if any position in the four quarter-warps is available. Here, availability is defined as no active threads existing in the male

or female warp. If any available position is found, go to the next step. In this example, the first position satisfies the condition and offers a chance for thread 59 to eliminate lane conflict.

3) Switch the contents of the available position with the conflict position. Thus, thread 59 is moved to the first quarter and leaves a hole in the previous location.

4) Repeat the steps above until the eighth lane is finished.

This method increases the possibility of warp integration. En-DWF performs the same as DWF if there is no lane conflict. If there are no available holes in the conflict lane, warp combination cannot be achieved. The male warp will be located in the warp pool as a new entry. Even though DWF and HWS both rely on rearranging threads to increase warp density and further improve SIMD efficiency, they are orthogonal methods and can be combined (called en-DWF&HWS here), as shown in Figure 4.6. The step-by-step operations are listed below:

1) Apply the squeeze algorithm to the male warp. In Figure 4.6, the same warp as in Figure 4.3 is used, so we directly get the resulting warp.

2) Apply en-DWF to the male warp. Rearrange the threads of the male warp to try to eliminate lane conflict.

3) Combine the two warps if all of the lane conflicts are gone.

4) Downscale the newly formed warp to reach the minimum size.

Compared to DWF, en-DWF&HWS increases the rate of warp formation, as well as the warp occupancy. It benefits from the looser condition of warp integration and has better tolerance of various warp sizes.

4.5 Summary

To this point, this thesis has discussed the various aspects of HWS, including the squeeze algorithm, warp scaling operations and how to issue and execute hybrid

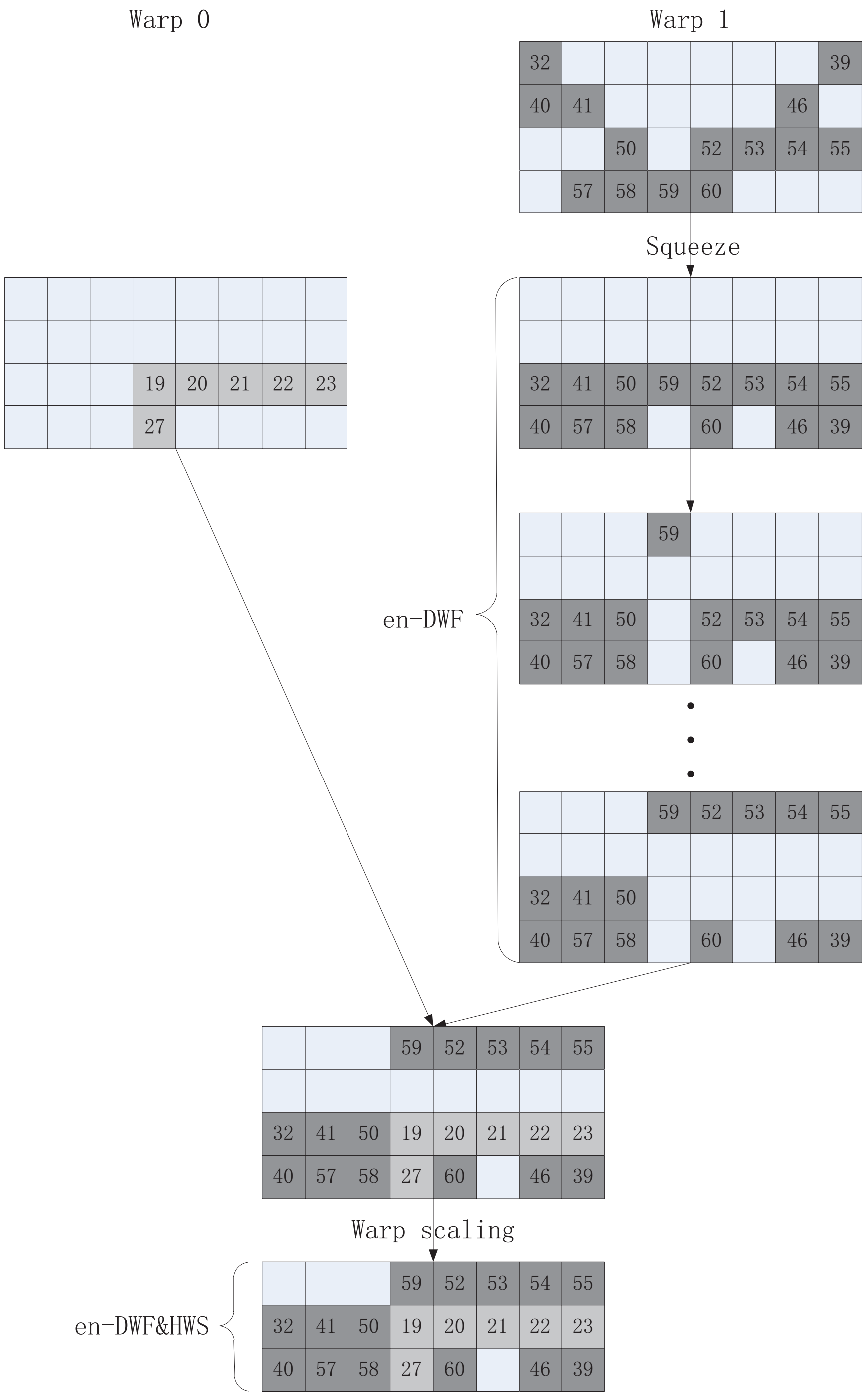


Figure 4.6   An example of en-DWF&HWS.

warps. This chapter also describes the implementation details of the enhancement to DWF, as well as the integration of DWF and HWS.

5. Methodology

This chapter describes the experiment method and specifies the system configurations adopted in this work. Then, the properties of the benchmarks used are listed to facilitate the result analysis and discussion.

5.1 GPGPU-Sim

A model of the mechanism proposed here is obtained by modifying a well-known general purpose GPU simulator, GPGPU-Sim (Version 2.1.2b). This simulator covers various aspects of a massively parallel architecture with highly programmable pipelines similar to those found in contemporary GPU architecture, such as CUDA [34], and can yield cycle-accurate performance statistics. In CUDA, the GPU device is invoked by the operating system through a system call. In GPGPU-Sim, this procedure is implemented by a spawn instruction, which signals the CPU host to launch a parallel compute kernel with predetermined configurations on the simulator [37]. By modifying the *common.mk* makefile used in the CUDA SDK, the simulator can make an application link to the customized CUDA library instead of to the original version, and execute parallel instructions according to the system configurations specified by a file named *gpgpusim.config*.

GPGPU-Sim includes two components: functional simulation and timing simulation. The functional model is designed for the parallel thread execution (PTX) codes, which are based on the CUDA instruction set architecture. The timing model is used to estimate how fast a program will run on the modeled platform. It contains the timing for the SMs, cache latency, interconnection network, memory controllers and graphics DRAM. Timing for the CPU and the communication cost between the CPU and GPU are not included. Although recent GPUs have enabled the concurrency of CPU and GPU, GPGPU-Sim assumes the CPU to be idle when the GPU is processing. Similar to other processor simulators, GPGPU-Sim separates the functional simulation from the performance simulation to facilitate other developers to quickly evaluate performance for novel ideas.

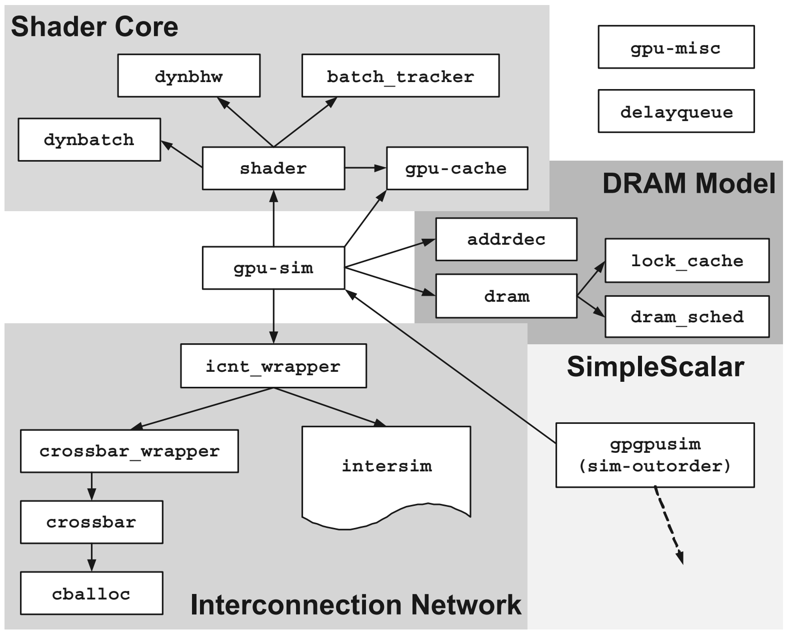


Figure 5.1 The overview of GPGPU-Sim.

As shown in Figure 5.1 [37], GPGPU-Sim contains three main modules: shader core (same as SM), interconnection network and DRAM. The SimpleScalar is used for modeling the CPU. The interconnection network module is designed for relaying messages between the SMs and the memory controllers. It models the traffic control and the timing for each message through it regardless of the content or size of the message. The DRAM module simulates the basic DRAM components such as the address decoder and the DRAM request scheduler. Moreover, it provides the DRAM access timing model, which is essential for developers to analyze performance bottlenecks. Different modules could be configured at various frequencies. The communication between the adjacent modules is implemented through the clock crossing buffers that are filled at the source domain's clock rate and drained at the destination domain's clock rate.

The SM module is the key component of the simulator. It simulates the SIMD pipeline in a similar way to the classic MIPS 5-stage in-order pipeline [27]. The following section discusses further the software design of the SM module, as shown in Figure 5.2 [37]. Note that these stages are simulated in reverse order to the real hardware pipeline in order to eliminate the need for two copies of each pipeline register. This is common among several processor simulators such as SimpleScalar [37].

Fetch

The fetch stage is the key stage in this work because the thread scheduling, as well as the thread issuing, is performed here. The thread schedule mechanism is vital to GPU performance since it crucially affects the SIMD throughput. At present, there are four schedulers: No reconvergence, PDOM, DWF and MIMD. Developers can specify any one of them by the configuration file $gpgpusim.config$ or the command line. The first three mechanisms have been discussed in Chapter 3. The fourth, the MIMD scheduler, can freely schedule threads into pipelines despite the program counter values. In the software, these four schedulers are implemented by four independent functions.

Issue policy is another important factor that contributes to final performance. Chapter 3 discusses the related policies and gives a recommended option: the majority policy, which picks up the warps with the most common PC and continues to issue warps at this PC before switching to other PCs.

Once the threads (labeled by $TIDs$ in Figure 5.2) are selected, their PC values are transferred to the fetch unit, which reads the corresponding instruction from the instruction cache. Meanwhile, the threads are locked until all of them have finished

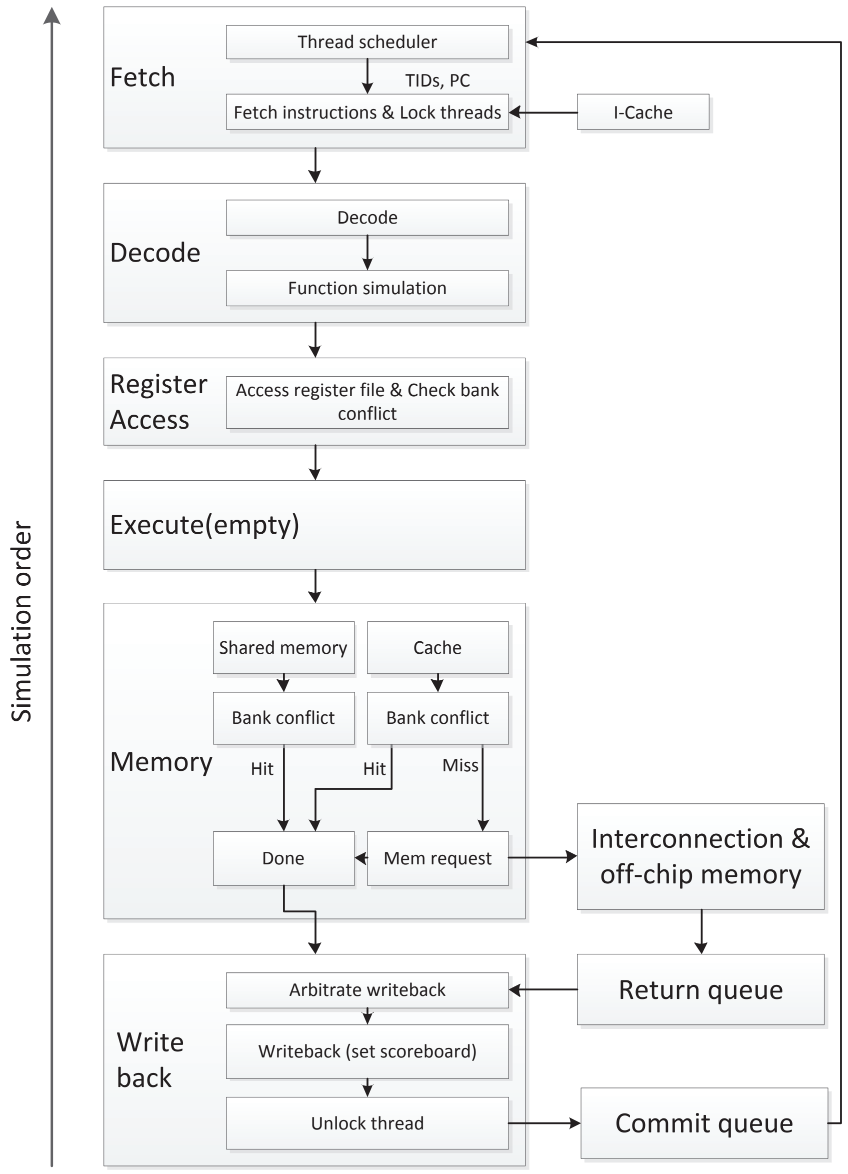


Figure 5.2 Pipeline stages of a SM.

execution without any outstanding stores or pending writes to local registers and have been committed during the last stage. The warp distribution information is also recorded by the simulator in this stage. Then the fetched instruction is put into the pipeline register of the decode stage.

Decode

In the classic MIPS pipeline, the decode stage mainly accomplishes the following two tasks:

1. Classify the instruction. Memory instructions are directed to the memory pipelines. Arithmetic instructions are processed in the default pattern. For branch instructions, a per-warp stack is used to handle the multiple paths. For our baseline architecture which employs the PDOM mechanism, at each divergence point a new entry is pushed to the top of the stack. Each entry includes the target branch PC, the active mask corresponding to the threads of that branch and their immediate reconvergence point PC. When the reconvergence point is reached, the stack pops the top entry.

2. Label the registers that will be used. A scoreboard is used here to label these reserved registers and indicate that they are in use. In the final stage, the reserved registers will be released.

In the GPGPU-Sim, the functional simulation is also performed in this stage. PTX codes generated by the NVCC or OpenCL compiler are simulated by the separate functional simulator. At the same time, much information is obtained such as memory space, memory address and next PC value. These can be used to facilitate the performance simulation.

Register Access

The main task in this stage is to guarantee that the banked register file,* which is shown in Figure 5.3, is read properly. The main issue is bank conflict. The register file bank depends on the thread ID. For each warp, if there is more than one thread accessing the same bank, then the multiple accesses will be serialized until each one is finished.

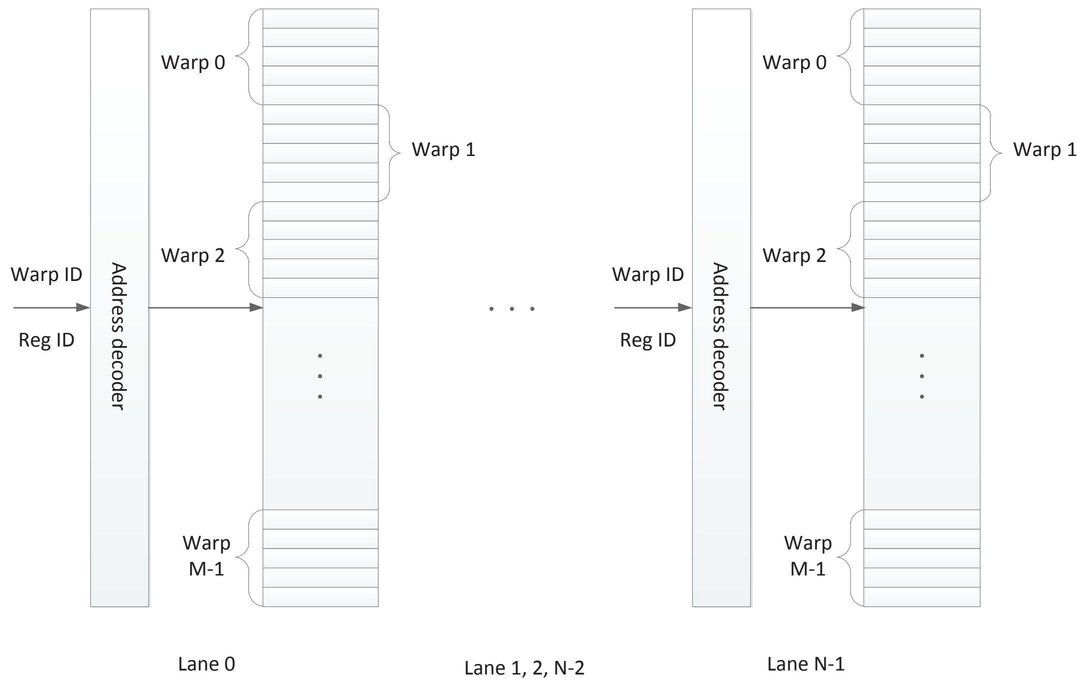


Figure 5.3 The banked register file.

Execute

The execute stage in the GPGPU-Sim is an empty stage since the functional simulation has been finished in the decode stage. Each SM contains eight streaming processors (SPs) and two special function units (SFUs), so it is a reasonable extension to add the timing for these modules in a future performance model, especially for those applications with many transcendental instructions.

*According to [38] the banked register file is a single ported RAM, with the appearance of a multiported register file using multiple banks through a patented technique called "operand collector".

Pre-Memory

The pre-memory is an optional stage designed to adjust the pipeline length. In the baseline model, this stage is empty.

Memory

GPGPU-Sim supports multiple memory spaces, as shown in Figure 5.3. Each thread owns a private data cache, which can be accessed in one cycle. Missed accesses are inserted into a FIFO miss queue and are managed through Miss Status Holding Registers (MSHR). A memory request is then added to the MSHR table and sent to the interconnection network. In addition, bank conflict is checked among the multiple data cache accesses within a warp.

The shared memory is a fast memory space that can be explicitly configured by programmers. It is shared within a thread block. It is also highly banked, so bank conflict is checked here. For NVIDIA GPUs, the shared memory has 16 banks with 16 KB per SM. Note that bank conflict is detected within each half-warp.

There are also two additional read-only memory spaces accessible to all threads: the constant and the texture memory spaces [3]. They facilitate the use of constants and some specific data types. However, once cache misses appear, the delay to access the constant or texture memory spaces in DRAM will be significant.

Write back

The write back stage mainly accomplishes the following tasks:

1. Arbitrate write-back between the threads from the memory stage and the thread-s from the return queue. In the GPGPU-Sim, the return queue has a higher priority, a condition which may generate better results because different threads can be balanced better and kept at a similar speed during the entire execution.

2. Update the registers using the output values and clear corresponding scoreboard

entries to indicate that the registers can be safely accessed without concern for dependency hazard.

3. Send the selected threads to the commit queue and unlock them to inform the thread scheduler that these threads are available for scheduling again.

5.2 System Configurations

For this work, the system configurations are shown in Table 5.1.

Table 5.1 System configurations.

#Streaming multiprocessors	28
Warp size	32
SIMD width	8
#Registers per SM	16384
Shared memory per SM (KB)	16
Constant cache per SM (KB)	8
L1 cache	None
L2 cache	None
#Memory modules	8
Bandwidth per memory modules	8(Bytes per cycle)
Memory mode	Perfect memory mode
Memory controller	Out of order
Warp scheduling policy	Majority + Round Robin
Branch divergence handling	Immediate post dominator (PDOM)
Interconnect topology	Mesh

To focus on the control flow issue, this work prevents the influence of the memory access by setting the simulator to the *perfect memory* mode, which means zero memory latency and no cache misses. The memory controller is mainly responsible for the scheduling of the multiple DRAM requests. The *out-of-order* mode means *First-Ready First-Come First-Serve*. It reorders the requests to prioritize those which access an already opened row [37]. This method takes advantage of the shared open row and reduces the overhead required to activate a new single row. Warps with different PCs are scheduled based on the majority policy. Warps with the same PC are scheduled following the round-robin policy. Mesh is a simple interconnection topology, even though the latency is relatively high.

5.3 Benchmarks

The same benchmarks in [34], listed in Table 5.2, are used here for comparison purposes. All benchmark source codes are obtained from GPGPU-Sim release. These benchmarks are developed and executed on GPU devices. Ali, *et al.* [34] then modified these applications and mapped them onto GPGPU-Sim.

Breadth-First Search (BFS) [11] is a fundamental algorithm in the graph processing field. It is an uninformed search method and searches all the nodes in a graph without considering the goal until it finds the target. Branch instructions such as $if - else$ are inevitable, and therefore, BFS loses much performance due to branch divergence. In our experiment, we test the BFS program on a random graph with 4096 nodes.

Black-Scholes (BS) [39] is a model that provides a partial differential equation for the evolution of an option price under certain assumptions. NVIDIA implemented it using CUDA. To allow for arbitrary numbers of options, each thread processes more than one index as required. Due to the existence of a closed-form expression, calculating option prices is not a difficult task. No branch instruction is involved here.

Neural Network (NN) [12] is a widely used pattern recognition method. In [12], it is used to recognize handwritten digits. In our experiment, 28 digits from the Modified National Institute of Standards Technology database of handwritten digits are tested in parallel on GPGPU-Sim. It is worth noting that the last two kernels contain blocks of only one thread each, a configuration which results in severe reduction of warp occupancy [34].

MUMmerGPU (MUM) is a GPGPU drop-in replacement for MUMmer, which is a sequence alignment program widely used in genotyping, genome resequencing, metagenomics and de novo genome assembly projects. It uses the GPUs to align simultaneously multiple query sequences against a single reference sequence stored as a suffix tree [13]. In our experiment, the first 140,000 characters of the Bacillus anchracis str. Ames genomes (a special sequence of genes) are used as the reference string. 50,000 25-character queries generated randomly using the complete genome are used as the seed [34]. Similar to BFS, MUM is also a branch intensive application, because both can carry out a large number of comparisons.

Table 5.2 Benchmark properties.

Benchmark	Description	Branch divergence intensity
BFS	Breadth-first search on a graph	High
BS	Financial options pricing	Low
NN	Neural network algorithm for recognizing handwritten digits	Medium
MUM	Sequence alignment program	High
RAY	Rendering graphics with near photo-realism	Medium
STO	A library that accelerates hashing-based primitives	Low
LPS	3D Laplace solver	High
NQU	N-Queen solver for a chess puzzle	High
AES	Advanced Encryption Standard algorithm to encrypt and decrypt files	Low
LIB	Monte Carlo simulations	Low

Ray Tracing (RAY) is the core method of photorealistic rendering using global illumination simulation [40]. In [40], each pixel is rendered by a scalar thread in

CUDA. Since the rendering operation for each pixel depends on the objects it hits and in real life these objects are always varied, branch divergence here is a considerable problem. In our experiment, we render a 256×256 graph with up to 5 levels of reflections and shadows.

StoreGPU (STO) is a library that accelerates a number of hashing based primitives popular in distributed storage system implementations [41]. In our experiment, the input size is set to 192KB. Sliding-window hashing is adopted based on the MD5 algorithm [41]. The main challenge of this application is memory management rather than flow control. Branch divergence is barely involved here.

3D Laplace Solver (LPS) is an important mathematical tool since the Laplace equation can describe the properties of electric, gravitational and fluid potentials. [14] implemented it on CUDA with careful memory management, but branch divergence still exists due to a complex and nested variable index. In our experiment, the grid is set to 100×100×100 and one iteration is performed.

N-Queen Solver (NQU) tackles the classic puzzle of placing N queens on an $N \times N$ chess board where no queen can capture another [15]. [15] uses a simple backtracking algorithm to enumerate all possible solutions. The CPU is used to generate a large number of configurations for upper rows and the GPU is responsible for the remaining rows. In our experiment, N is set to 10. It is worth noting that for most of the time only a single thread is active.

Advanced Encryption Standard (AES) is the most widely adopted modern symmetric key encryption standard [42]. [42] implemented it on CUDA with multiple optimizations. In our experiment, we encrypt a 256×256 image with a 128-bit encryption key.

LIBOR Monte Carlo (LIB) implemented a LIBOR market model and tested it using the Monte Carlo method on CUDA [43]. The parallel random numbers are generated by a Sobol's quasi-random generator. Most of the processing time is spent on computation and memory operations. Branch divergence is not a key issue here.

We use the default inputs, 4096 paths for 15 options.

For further benchmark properties, such as grid/block dimensions, instruction counts and memory utilization, refer to [34].

6. Results

This chapter evaluates various aspects of our experimental results. First, the potential performance improvement brought by HWS is theoretically estimated. Then the effect of the squeeze algorithm is analyzed and the performance improvement resulting from en-DWF over DWF is discussed. Next, the measured results on the baseline architecture are described, with emphasis on the speedup of HWS. Also, the simulation results are verified with the estimated results. Finally, this chapter examines the DWF-enabled platform and discusses the performance after integrating DWF and HWS.

6.1 Estimated Performance

The following factors need to be considered carefully to theoretically estimate the HWS performance improvement. The first is the warp distribution which can be obtained through the simulator. However, the detailed statistics of how threads are distributed within each warp are not accessible. For example, suppose there is a warp containing 24 threads. If all these threads are located in the first three quarters, then one cycle can be saved by applying HWS. Conversely, if those four quarters all contain at least one active thread, then no improvement can be achieved through HWS. The second factor is the effect of the squeeze operations. Suppose those 24 threads spread across the four quarters. After the squeeze operations, they may be compressed into three quarters and leave a quarter free. Or they may still occupy all four quarters. For simplicity, it is assumed that threads are distributed optimally. Thus, based on the warp distribution and the ideal speedup for each fraction, the following equation [28] can be applied to get the final estimated speedup.

$$Speedup_{overall} = \frac{1}{(1 - \sum_i Fraction^i_{enhanced}) + \sum_i \frac{Fraction^i_{enhanced}}{Speedup^i_{enhanced}}} \qquad (6.1)$$

Here, i ranges from 0 to 2 and $Fraction^i_{enhanced}$ represents the percentage of warps with 1, 2 and 3 active quarter-warps. The corresponding values of $Speedup^i_{enhanced}$ are 4, 2 and 1.33. This is a simple model, regardless of the problem discussed previously and the change to the memory access pattern, and therefore, it is foreseeable that estimated values are higher than simulation outcomes in some degree. Section 6.4 verifies this by comparing the estimated speedups for the ten benchmarks with the experimental results. First, the contributions of different portions of HWS will be evaluated.

6.2 Effects of the Squeeze Algorithm

The squeeze algorithm is the first step in the HWS. It makes warps denser and facilitates the scaling operation, however it is not an indispensable part of it. Figure 6.1 shows the performance gain of the squeeze operations. Note that enhancement to DWF is not included. On average, PDOM&HWS obtains 0.6% improvement and DWF&HWS gets 2.2% gain. This verifies that the randomness of thread distribution is raised by DWF. In other words, by dynamically forming new warps, threads spread in a more irregular and random way. Therefore the squeeze operations function better in this kind of organization. For benchmarks RAY and LPS under DWF, instructions per cycle (IPC) are degraded, not improved. The reason is that after applying squeeze operations, the merge conflict rate is increased for these benchmarks. More under-filled warps cannot merge into new dense warps, and the performance loss from this overtakes the advantage of the squeeze operations. It is suggested the squeeze algorithm be combined with en-DWF to address this issue.

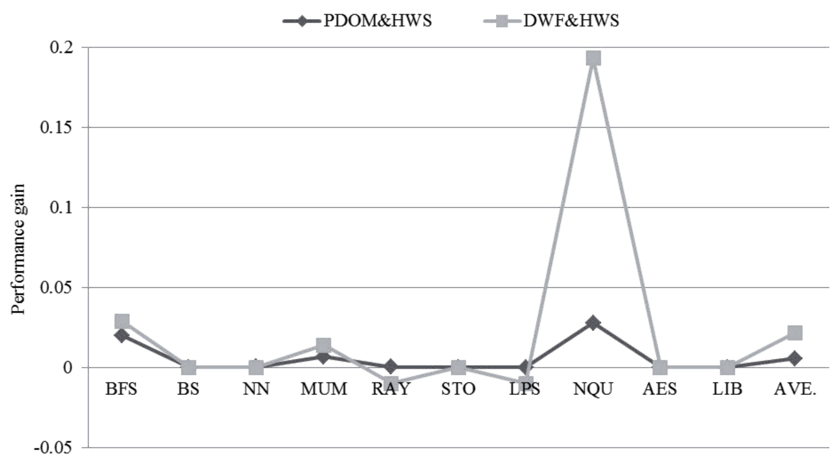


Figure 6.1 Performance gain of the squeeze algorithm.

6.3 Effects of the Enhancement to DWF

The DWF algorithm was modified in Chapter 4 to produce the en-DWF algorithm. Figure 6.2 shows the performance improvement due to this modification. On average 8.8% improvement is achieved, compared to the original DWF mechanism, and there is no performance loss occurs for any applications. This is due to the increased merge rate and the decreased number of combined warps for en-DWF. This also verifies that thread distribution within an under-filled warp due to branch divergence is similar to others' distribution in some degree, and the modification to DWF produces a better SIMD efficiency.

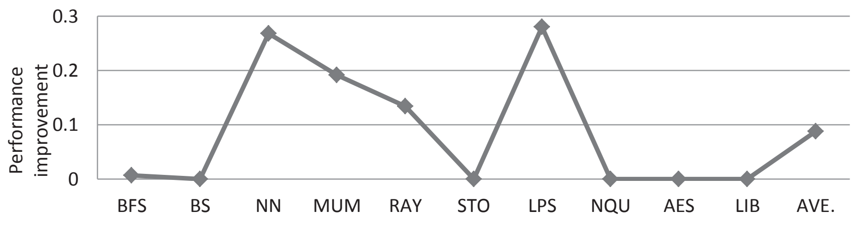


Figure 6.2 Performance improvement of en-DWF.

6.4 Measured Performance

Figure 6.3 compares the estimated speedup and the measured speedup that the HWS mechanism contributes to the baseline architecture which employs the immediate post-dominator (PDOM) branch divergence handling policy. The correlation coefficient between them is 0.85. Since the squeeze algorithm is lane aware, it is impossible to obtain the expected speedup for every under filled warp. For example, if there is a warp with 4 active threads, ideally, the warp size will be scaled to 8. However, if two of the four threads are located in the same lane, the warp size will be scaled to 16 instead of 8. As a result, the measured speedup will be less than the estimated speedup, as shown in Figure 6.3. The only exception is the benchmark NQU. Notice that the estimated speedup is calculated based on the assumption that warps are assigned to all SMs evenly and the warp distribution within each SM is the same. However, at the end of NQU, only one SM is employed and warp distribution within this SM is shown in Table 6.1. From the last column we can see that a higher speedup is generated based on the distribution of the last stage. This higher value results in the final excess of the measured speedup over the estimated speedup.

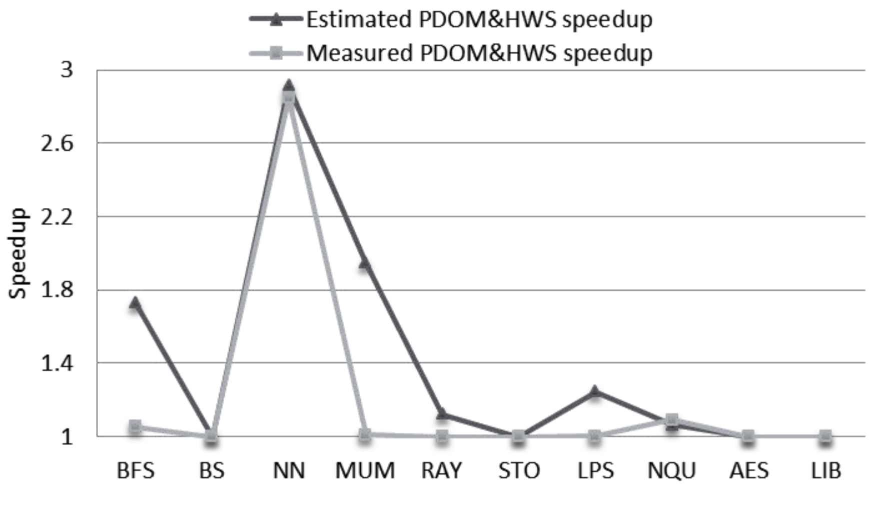


Figure 6.3 Estimated versus measured speedup for PDOM&HWS.

Table 6.1 Warp distribution of NQU.

Speedup for each fraction	4	2	1.33	0	Speedup
Warp distribution for all stages	0.062	0.026	0.006	0.906	1.065
Warp distribution at the last stage	0.058	0.068	0.067	0.807	1.104

Figures 6.4 and 6.5 show the performance for the ten benchmarks in terms of IPC and speedup. In Figure 6.5, the weight of control flow instructions is also presented. First we will discuss the benefit HWS brings to PDOM. For the ten applications, a speedup of 1.20 is achieved on average. Benchmarks BFS, NN and NQU obtain relatively more improvement. Figure 1.1 shows that these three applications all have a considerable number of low occupancy warps, which means more throughput loss can be retrieved by HWS. In Figure 6.5, notice that it is the large control flow portion in BFS and NQU that leads to the low occupancy. For NN, although branch instructions only occupy 6%, this program has large portions of the codes spent on a single thread [34], which leads to 95% of warps having less than 4 active threads. As a result, HWS takes advantage of these under-filled warps and yields a significant speedup: 2.85. For benchmarks BS, STO, AES and LIB, because all warps have a warp occupancy of 29-32, there is no space available to improve performance through HWS.

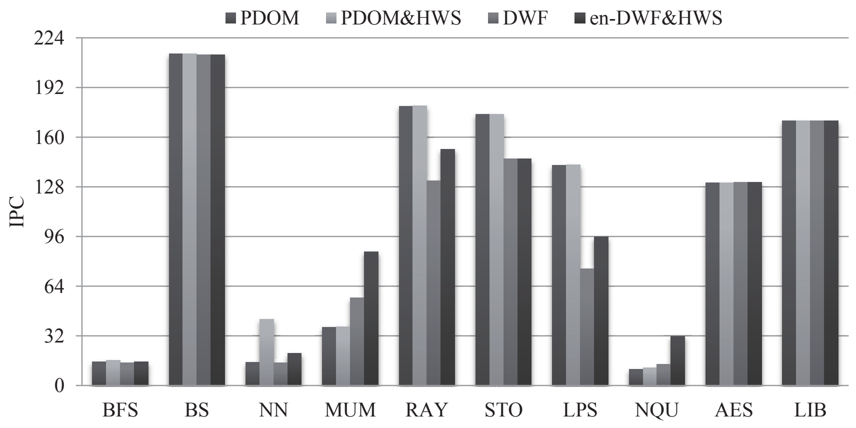


Figure 6.4 IPC comparison.

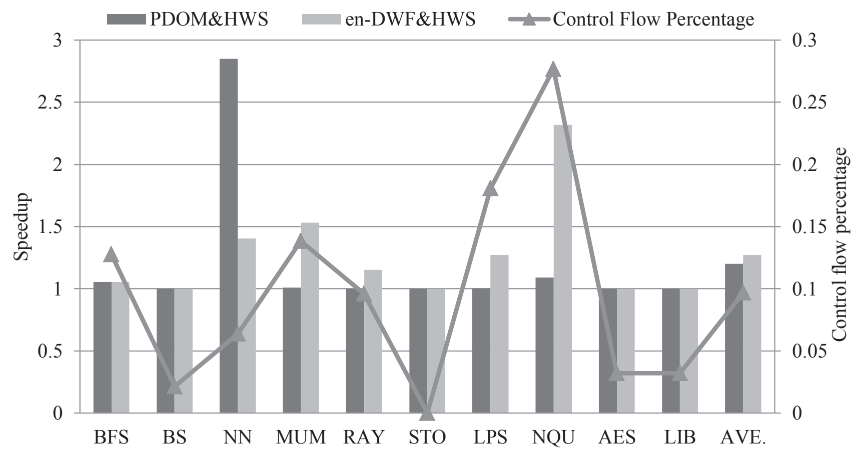


Figure 6.5 Speedup comparison and the percentage of control flow. PDOM&HWS represents the speedup HWS brings to PDOM; en-DWF&HWS represents the speedup HWS brings to DWF.

When integrated with DWF, HWS generates a speedup of 1.27 on average. For each benchmark, HWS performs better when integrated with DWF than with PDOM. Moreover, a benchmark that gains significant speedup with PDOM&HWS will very likely be improved by en-DWF&HWS since warp occupancy mainly depends on the properties of the application itself, not on the branch handling mechanism it adopts.

6.5 Summary

This chapter discusses the simulation results of the various aspects of HWS. It calculates the theoretical improvement of HWS and verifies that it agrees with the simulation results. The contribution of the squeeze algorithm is also evaluated. For the baseline platform, 0.6% improvement is achieved with squeeze operations. This value goes up to 2.2% when DWF is employed. Furthermore, en-DWF is shown to be more efficient than DWF by 8.8% on average for the selected benchmarks. Finally, the detailed performance metrics are listed for the four platforms: PDOM, PDOM&HWS, DWF and en-DWF&HWS. There is an average speedup of 1.20 on the baseline platform and 1.27 on the DWF-enabled architecture.

7. Conclusions and Future Work

In this thesis, a novel technique is proposed for GPU SIMD control flow handling: hybrid warp size mechanism which abandons the fixed-size warp design and allows mixed-size warps to be scheduled and executed by hardware. When branch divergence occurs, warps are squeezed with lane awareness, and then warp sizes are downscaled wherever possible. Based on the updated warp sizes stored in the warp pool, the warp scheduler calculates the number of cycles the current warp needs and issues the next warp accordingly. As a result, hybrid warps are pushed into pipelines as soon as possible and more pipeline stages are overlapped. The proposed technique was evaluated using a modified version of the famous GPU simulator, GPGPU-Sim. The estimated theoretical potential improvement was calculated for HWS and compared with the simulation results. The correlation coefficient between them was 0.85. For the baseline architecture, an average speedup of 1.20 was achieved for a set of general purpose GPU applications.

The well-known branch handling mechanism, DWF, was analyzed and it was found that the efficiency of warp formation could be further improved. The warp formation pattern was modified to allow thread migration before integration and to avoid conflicts as much as possible. Consequently, more warps could be combined and fewer warps were scheduled, resulting in a better SIMD utilization. According to the simulations, 8.8% improvement was obtained compared to the original DWF method.

In addition, this work was compared with other previous methods and it was found that they are orthogonal. Therefore, it is worthwhile to analyze further the advantages and disadvantages of these methods with the goal of combining them in some useful way. As an example, a proposed implementation of the integration of DWF and HWS was presented. For clarity, the next warp that is waiting for regrouping with the other warps is defined as a male warp, and the selected warp from the warp pool offering opportunity of combination is defined as a female warp. Most operations including the squeeze operations and the en-DWF operations happen on the male warp. The simulation shows that the set of benchmarks are accelerated on average by a factor of 1.27.

Future work involves implementing HWS on hardware, including the squeezer, warp size analyzer, and the modification to the warp pool and other modules. This would provide a means to estimate the area overhead and the extra power consumption.

The location of the squeeze algorithm is another issue which needs further exploration. In this work, when integrating with DWF, HWS applied squeeze operations to the male warp before trying to combine with the female warp. It also works if the squeeze operations are performed on the updated warp (the combination of the male and female warps). However, it might be difficult to prove which one is better.

A. Squeeze Algorithm Codes

```
//Initialization
for(unsigned i=0;i<4;i++){
    thd_count_quar[i]=0; //Record thread counts for each quarter
    flag_quar[i]=i; //Record quarter IDs for sort operations
}

//Calculate thread amount for each quarter
for(unsigned i=0;i<4;i++){
    for(unsigned j=0;j<8;j++){
        if(tid_original[i*8+j]!=-1)
        thd_count_quar[i]++;
    }
}

//Sort four quarters according to thread amount
for(unsigned i=0;i<3;i++){
    for(unsigned j=i+1;j<4;j++){
        if(thd_count_quar[i]<thd_count_quar[j]){
            int temp_count;
            int temp_flag;
            temp_count=thd_count_quar[i];
            temp_flag=flag_quar[i];
```

```
                thd_count_quar[i]=thd_count_quar[j];
                flag_quar[i]=flag_quar[j];
                thd_count_quar[j]=temp_count;
                flag_quar[j]=temp_flag;
            }
             }
}

for(i=0;i<warp_size;i++){ //Warp_size equals to 32
    tid_squeez[i]=tid_original[i];
}

//Start from the quarter with the most active threads
for(unsigned i=0;i<3;i++){
    int l=flag_quar[i];

    //No holes can be filled in this quarter
    if(thd_count_quar[l]==8) continue;

    for(unsigned j=0;j<8;j++){
        //Current position already contains an active thread
        if(tid_original[l*8+j]!=-1) continue;

        //Scan from the quarter with the least active threads
        for(unsigned k=3;k>i;k--){
            int p=flag_quar[k];

            //No thread could be taken to fill the hole
            if(tid_original[p*8+j]==-1) continue;
```

```
                    tid_squeez[l*8+j]=tid_squeez[p*8+j];
                    tid_squeez[p*8+j]=-1;
                    break;
                     }
                }
        }
```

B. en-DWF Codes

```
for(unsigned i=0;i<8;i++){ //Process 8 lanes separately
  for(unsigned j=0;j<4;j++){ //Process 4 holes in each lane
    unsigned test_place=j*8+i;

    //Only process conflict places
    //m_occ_ext represents the active mask for the female warp
    if(m_occ_ext[test_place]>0 && tid_squeez[test_place]>-1){
      for(unsigned k=0;k<4;k++){
        if(k!=j){
          unsigned scan_place=k*8+i;

          //Check if there is a place that contains
          //holes both in the male and female warps
          if(m_occ_ext[scan_place]==0 && tid_squeez[scan_place]==-1){
            tid_squeez[scan_place]=tid_squeez[test_place];
            tid_squeez[test_place]=-1;
            break;
          }
        }
      }
    }
  }
}
```

C. Warp Size Analyzer Codes

```
unsigned simd_width=8;
unsigned warp_size_this_shader=32;

for (unsigned i=0;i<warp_size/simd_width;i++) {
    int hws_flag=0;

    for (unsigned j=0;j<simd_width;j++){
        unsigned tem_index=i*simd_width+j;
        if(tid_squeez[tem_index]<0) hws_flag++;
    }

    //Current quarter contains no active threads
    if(hws_flag==8) warp_size_this_shader-=8;
}

//Force a synchronization to all the SMs
if(warp_size_this_shader>warp_size_each_loop) {
    warp_size_each_loop=warp_size_this_shader;
}
```

References

[1] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7 –17, 2011.

[2] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili, "Simd re-convergence at thread frontiers," in *the 44th Annual IEEE/ACM International Symposium on Micro*, pp. 477–488, ACM, 2011.

[3] NVIDIA Corporation, *CUDA C Programming Guide, 4.1 edition*, 2011.

[4] Khronos Group, *The OpenCL Specification*, 2010.

[5] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH*, pp. 18:1–18:15, ACM, 2008.

[6] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," *Micro, IEEE*, vol. 30, no. 2, pp. 7 –15, 2010.

[7] NVIDIA Corporation, *NVIDIA's next generation CUDA computer architecture: Fermi whitepaper*, 2009.

[8] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *Micro, IEEE*, vol. 32, no. 2, pp. 28 –37, 2012.

[9] NVIDIA Corporation, *CUDA C Best Practice, 4.1 edition*, 2011.

[10] N. Corporation, "Cuda zone." `http://www.nvidia.com/cuda`, 2012.

[11] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *the 14th International Conference on High Performance Computing*, pp. 197–208, Springer-Verlag, 2007.

[12] Billconan and Kavinguy, "A Neural Network on GPU." `http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU`, 2012.

[13] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, 2007.

[14] M. Giles, "Jacobi iteration for a Laplace discretisation on a 3D structured grid." `http://people.maths.ox.ac.uk/gilesm/codes/laplace3d/laplace3d.pdf`, 2012.

[15] Pcchen, "N-Queens Solver." `http://forums.nvidia.com/index.php?showtopic=76893`, 2012.

[16] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick, "The Illiac IV system," *Proceedings of the IEEE*, vol. 60, no. 4, pp. 369 – 388, 1972.

[17] Advanced Micro Devices, Inc., "ATI CTM Guide," 2006.

[18] R. A. Lorie and H. R. Strong, "Method for conditional branch execution in SIMD vector processors," 1984.

[19] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *the 33rd Annual IEEE/ACM International Symposium on Micro*, pp. 159–170, ACM, 2000.

[20] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graph*, vol. 24, no. 3, pp. 434–444, 2005.

[21] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *the 31st Annual International Symposium on Computer Architecture*, pp. 52 – 63, 2004.

[22] W. Fung and T. Aamodt, "Thread block compaction for efficient simt control flow," in *the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 25 –36, 2011.

[23] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *the 44th Annual IEEE/ACM International Symposium on Micro*, pp. 308–317, ACM, 2011.

[24] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *the 37th Annual International Symposium on Computer Architecture*, pp. 235–246, ACM, 2010.

[25] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," tech. rep., 2012.

[26] A. K. Ahmad Lashgar, Amirali Baniasadi, "Investigating Warp Size Impact in GPUs," tech. rep., 2012.

[27] W. Stallings, *Computer Organization and Architecture*. Pearson Education, Limited, 2012.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (4th edition)*. Morgan Kaufmann, 2007.

[29] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948 –960, 1972.

[30] B. Parhami, *Computer Architecture: From Microprocessors To Supercomputers*. Oxford Series in Electrical and Computer Engineering, Oxford University Press, 2005.

[31] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design, Academic Press, 2012.

[32] D. Calahan and W. Ames, “Vector processors: models and applications,” *IEEE Transactions on Circuits and Systems*, vol. 26, no. 9, pp. 715 – 726, 1979.

[33] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” in *SIGGRAPH*, pp. 777–786, ACM, 2004.

[34] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 163 –174, 2009.

[35] W. Fung, I. Sham, G. Yuan, and T. Aamodt, “Dynamic warp formation and scheduling for efficient GPU control flow,” in *the 40th Annual IEEE/ACM International Symposium on Micro*, pp. 407 –420, 2007.

[36] S. S. Muchnick, *Advanced compiler design and implementation.* San Mateo: Morgan Kaufmann, 1997.

[37] W. Fung, “Dynamic warp formation: exploiting thread scheduling for efficient MIMD control flow on SIMD graphics hardware,” Master’s thesis, University of British Columbia, Vancouver, BC, Canada, 2008.

[38] UBC Computer Architecture Group, “GPGPU-Sim 3.x Manual.” `http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual#Register_Access_and_the_Operand_Collector`, 2012.

[39] NVIDIA Corporation, “NVIDIA CUDA SDK code samples.” `http://developer.nvidia.com/cuda-cc-sdk-code-samples`, 2012.

[40] Maxime, “NVIDIA CUDA Community Showcase.” `http://www.nvidia.com/object/cuda-apps-flash-new.html#state=detailsOpen;aid=21efc800-8959-11dd-ad8b-0800200c9a66`, 2012.

[41] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, “Storegpu: exploiting graphics processing units to accelerate distributed storage

systems," in *the 17th International Symposium on High Performance Distributed Computing*, pp. 165–174, ACM, 2008.

[42] S. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *IEEE International Conference on Signal Processing and Communications (ICSPC)*, pp. 65 –68, 2007.

[43] M. Giles and X. Su, "Libor monte carlo application." `http://people.maths.ox.ac.uk/gilesm/hpc/`, 2012.