The Dissertation Committee for Veynu Tupil Narasiman
certifies that this is the approved version of the following dissertation:

An Enhanced GPU Architecture for Not-So-Regular Parallelism with Special Implications for Database Search

Committee:

Yale N. Patt, Supervisor

Derek Chiou

Mattan Erez

Donald S. Fussell

Michael C. Shebanow

An Enhanced GPU Architecture for Not-So-Regular Parallelism with Special Implications for Database Search

by

Veynu Tupil Narasiman, B.A.; B.S.E.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2014

Dedicated to all who have helped, guided, and supported me along the way.

Acknowledgments

This dissertation would not have been possible without the help and support of many. Here is my attempt to acknowledge their contributions.

First, I would like to thank my advisor, Professor Yale N. Patt, for sparking my interest in computer architecture several years ago when I was an undergraduate enrolled in his Introduction to Computing (EE 379K, a.k.a EE 306) and Computer Architecture (EE 360N) courses, and for his guidance since then and throughout my graduate student years. Dr. Patt always stressed the importance of not only being a good computer architect, but also a good human being, advice I will continue to follow throughout my career.

This dissertation would certainly not be possible without Dr. Michael C. Shebanow. I thank Mike for agreeing to work with me several years ago back when I hadn't the slightest clue about GPUs, and for mentoring me during summer internships at NVIDIA and serving on my committee. I am especially thankful for our weekly phone meetings. I remember thinking for hours after our phone conversations about something Mike said that I did not quite understand or agree with only to come to the same conclusion every time - Mike was right all along. Such wisdom is rare and I am very fortunate to have had a true scholar like Dr. Shebanow as a mentor. Thank you Mike.

I would also like to thank Professors Mattan Erez, Derek Chiou, and

Donald Fussell for their feedback and discussions, and for serving on my committee. Their input has improved the quality of this dissertation.

Many current and former HPS members have contributed to this dissertation and my life as a graduate student. I especially thank:

- Dr. Francis Tseng for first introducing me to research when I was a young graduate student who had just finished courses and was searching for a topic.
- Dr. Tse-Yu Yeh for mentoring me early on during my first few summer internships as a graduate student.
- Dr. Chang Joo Lee for introducing me to his research and for his work as a co-author of my first paper.
- Rustam Miftakhutdinov for his creativity and discussions on research, his help as a co-author of my first paper, and also for providing a fun and interesting work atmosphere.
- Jose Joao for maintaining (and teaching me about) our research group's IT infrastructure, and for our enjoyable experience and discussions as teaching assistants for EE 306.
- Dr. Carlos Villavieja for discussions on research, proofreading my entire dissertation, weekly squash matches, and very enjoyable dinners at his place.

- Khubaib, Faruk Guvenilir, Milad Hashemi, and Ben (Ching-Pei) Lin for providing a great research environment, a fun and interesting office to work in, and for their help proofreading this dissertation.

I also thank other HPS graduates including Professor Onur Mutlu, Professor Moinuddin Qureshi, Professor Hyesoon Kim, Dr. Aater Suleman, Dr. Eiman Ebrahimi, Danny Lynch, Santhosh Srinath, and Linda Hastings for providing a rich and enjoyable research environment during my years as a graduate student at UT Austin.

I deeply thank the Cockrell Foundation for providing me with several scholarships and fellowships throughout my undergraduate and graduate education. I also thank NVIDIA for the graduate fellowship I was awarded in 2007-2008.

I must also thank my family, for without them none of this would be possible. I thank my mother and father, Ranjana and Tupil Narasiman, for encouraging and nurturing my natural interests in math and science from an early age, and my older sister Tara and younger brother Vijay for their friendship and encouragement.

Finally, I thank my wife, Regina Koshy, for her never ending support, incredible patience, and constant belief that I could do this. Thank you Regina.

An Enhanced GPU Architecture for Not-So-Regular Parallelism with Special Implications for Database Search

Veynu Tupil Narasiman, Ph.D.
The University of Texas at Austin, 2014

Supervisor: Yale N. Patt

Graphics Processing Units (GPUs) have become a popular platform for executing general purpose (i.e., non-graphics) applications. To run efficiently on a GPU, applications must be parallelized into many threads, each of which performs the same task but operates on different data (i.e., data parallelism). Previous work has shown that *some* applications experience significant speedup when executed on a GPU instead of a CPU. The applications that benefit most tend to have certain characteristics such as high computational intensity, regular control-flow and memory access patterns, and little to no communication among threads. However, *not all parallel applications have these characteristics.*

Applications with a more balanced compute to memory ratio, divergent control flow, irregular memory accesses, and/or frequent communication

(i.e., not-so-regular applications) will not take full advantage of the GPU's resources, resulting in performance far short of what could be delivered. The goal of this dissertation is to enhance the GPU architecture to better handle not-so-regular parallelism.

This is accomplished in two parts. First, I analyze a diverse set of data parallel applications that suffer from divergent control-flow and/or significant stall time due to memory. I propose two microarchitectural enhancements to the GPU called the *Large Warp Microarchitecture* and *Two-Level Warp Scheduling* to address these problems respectively. When combined, these mechanisms increase performance by 19% on average.

Second, I examine one of the most important and fundamental applications in computing: *database search*. Database search is an excellent example of an application that is rich in parallelism, but rife with not-so-regular characteristics. I propose enhancements to the GPU architecture including new instructions that improve intra-warp thread communication and decision making, and also a row-buffer locality hint bit to better handle the irregular memory access patterns of index-based tree search. These proposals improve performance by 21% for full table scans, and 39% for index-based search.

The result of this dissertation is an enhanced GPU architecture that better handles not-so-regular parallelism. This increases the scope of applications that run efficiently on the GPU, making it a more viable platform not only for current parallel workloads such as databases, but also for future and emerging parallel applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to their massive computational horsepower, Graphics Processing Units (GPUs) have become a popular platform for executing general purpose (i.e., non-graphics) parallel applications. For an application to run efficiently on a GPU, the application must be able to be parallelized into many (e.g., thousands) threads, where each thread performs the same fundamental task (i.e., executes the same static image of code) but operates on different input data. This form of parallelism is known as data parallelism.

Parallel programming paradigms such as CUDA (Compute Unified Device Architecture) [41] and OpenCL [25] allow programmers to express such parallelism in applications by creating thousands of parallel threads, each of which executes the same piece of static code known as a kernel. Previous work [49, 19] has shown that *some* applications experience an order of magnitude speedup when run on a GPU instead of a CPU.

GPUs are able to achieve such speedups by exploiting parallelism in two major ways. First, threads executing the same code are grouped into fixed sized batches known as *warps*.[1] These warps are executed on a processing

[1]Warp size for current NVIDIA [41] GPUs is 32 threads.

core that employs a scalar front end (fetch and decode) and a SIMD (Single Instruction, Multiple Data) backend. The number of threads in a warp is usually equal to the SIMD width of the core so that a warp can execute an instruction for all its threads across the SIMD resources in parallel. Note that this style of processing is most efficient when threads in a warp have regular (i.e., identical) control flow behavior and regular (i.e., contiguous) memory access patterns.

Second, GPUs concurrently execute many warps on a single core. For example, 32 warps, each with 32 threads, can all be assigned to execute on the same core. When one warp is stalled, other warps can execute which helps tolerate data dependencies, branch penalties, and most importantly long latency operations (e.g., cache misses). Long latencies can be almost completely hidden for applications with high computational intensity and regular (e.g., streaming) memory access patterns.

Given such a microarchitecture, it is obvious that for an application to achieve significant speedup from GPU execution there needs to be an abundance of data parallelism available. Furthermore, the applications that benefit most from GPU execution tend to have certain characteristics such as high computational intensity, regular control flow behavior and memory access patterns, and little to no fine grain communication among threads. However, *not all parallel applications have these characteristics.* Parallel applications with a more balanced compute to memory ratio, divergent control flow behavior, irregular memory access patterns, or frequent fine grain communication among

threads will not make full use of the GPU, resulting in performance far short of what could be delivered. I call such applications *not-so-regular*[2] parallel applications. The goal of this dissertation is to enhance the GPU architecture to better handle not-so-regular parallelism.

1.1 The Problem: Regular versus Not-so-Regular Parallelism

As previously stated, the applications that benefit the most from GPU execution have certain characteristics such as:

1. *High computational intensity*, more specifically a high compute operation to memory operation ratio for the instructions in the kernel that all threads execute. The massive computational resources of the GPU can only be kept busy if the application requires several computations to be done. A low compute to memory ratio will result in severe stalling without any computation available to hide the latency, leaving the computational resources idle. Therefore, applications with low computational intensity will not benefit as much from GPU execution.

2. *Regular (i.e., non-divergent) control flow* behavior among the parallel threads. Grouping threads into warps is efficient if those threads remain on the same dynamic execution path (i.e., same Program Counter)

[2]I use the term not-so-regular, as opposed to irregular, since the data parallel nature of the application (thousands of threads executing the same kernel) implies some regularity in the first place.

throughout their execution. Although this may hold true for existing graphics applications (i.e., shaders), many general purpose parallel applications exhibit more complex control flow behavior due to frequent conditional branches in the code. Although all threads start out at the same Program Counter (PC), conditional branch instructions can cause threads in a warp to take different dynamic execution paths, or *diverge*. Since existing GPU implementations allow a warp to have only one active PC at any given time, these implementations must execute each path sequentially. This leads to lower utilization of SIMD resources while warps are on divergent control-flow paths because the warp must execute with a fewer number of active threads than the SIMD width of the core. Therefore, applications with severe divergent branching will not benefit as much from GPU execution.

3. *Regular (i.e., contiguous/streaming) memory access patterns* among the parallel threads. Regular memory access patterns are important for applications ported to the GPU for two main reasons: First, when threads in a warp access memory, if all threads access the same region in memory (i.e., a cache line), the requests can be *coalesced*,[3] and therefore serviced in a single memory transaction. If the threads need to access different regions of memory, a problem known as *memory divergence*, the warp will stall until all the memory transactions have completed. Second, con-

[3]coalescing requests means combining the requests from different threads into a single wide memory transaction.

tiguous/streaming memory access patterns can efficiently use the memory bandwidth provided on GPUs. The reason has to do with *page or row buffer locality*. When a warp accesses memory, an entire page/row (∼4 Kilobytes) is loaded into a per bank buffer (called the row buffer) even though the warp requests just a small fraction of that row (∼128 bytes). Subsequent accesses to that same row (i.e., row hits) can be read quickly and efficiently since 1) they can be pipelined, and 2) they have a short latency compared to requests that map to a different row (i.e., row conflict). Applications with streaming access patterns have high row buffer locality and therefore make efficient use of memory bandwidth. However, applications with more complex memory access patterns will suffer from significant stalling and poor memory bandwidth utilization, resulting in little to no benefit from GPU execution.

4. *Little to no fine-grain communication* among parallel threads. GPU execution works very well for applications with several identical but independent tasks. However, when threads have to frequently and quickly communicate values amongst each other (e.g., to make a collective decision during a search), existing GPU architectures have to go through several roundabout steps to make this happen due to the lack of fast and efficient communication across threads. Communication between threads in different warps must be carefully guarded by synchronization primitives which leads to overhead and inefficiency. Even intra-warp communication (i.e., across SIMD lanes), which obviates the need for

such synchronization since threads in the same warp are executed in lockstep, is limited and can be a bottleneck for certain applications such as the database search algorithms presented in Chapter 6. This leads to performance degradation for parallel applications requiring frequent fine grain communication among threads.

To summarize, applications with these previously discussed characteristics take full advantage of the GPU's resources, resulting in significant (e.g., orders of magnitude) speedup over single-threaded CPU execution. However, applications that suffer from one or more of the not-so-regular characteristics do not take full advantage of the GPU, resulting in performance far short of what could be delivered.

1.2 The Solution: Enhancements to the GPU Architecture

My proposals to improve GPU performance for not-so-regular parallel applications can be divided into two parts. First, I analyze a diverse set of data parallel applications that suffer from divergent control flow and/or significant memory stall time. I propose two microarchitectural enhancements to the GPU called the *Large Warp Microarchitecture* and *Two-Level Warp Scheduling* to address these problems respectively. Second, I examine one of the most important and fundamental applications in computing: *database search*. Database search is an excellent example of an important application that is rich in parallelism, but is rife with several of the not-so-regular char-

acteristics previously discussed. I propose new instructions that improve the intra-warp communication and decision making needed by database search, and also a row buffer locality hint bit to better handle the irregular memory access patterns of index-based tree search.

1.2.1 The Large Warp Microarchitecture and Two-Level Warp Scheduling

To alleviate the performance penalty due to branch divergence, I propose the *Large Warp Microarchitecture* (LWM). Existing GPU cores simultaneously execute many warps each with a modest number of threads (usually equal to or close to the SIMD width of the core). Instead, I propose creating fewer but correspondingly larger warps (that have a significantly larger number of threads than the SIMD width of the core), and dynamically creating SIMD width sized sub-warps from the active threads in a large warp. The key insight is that even in the presence of branch divergence, there will likely be a large number of active threads in the large warp. These active threads can be dynamically grouped together into fully populated sub-warps that can better utilize the SIMD resources on GPU cores.

To alleviate stall time due to memory latency for applications with a more balanced compute to memory ratio, I propose a novel two-level round-robin warp instruction fetch scheduling policy. Two-level scheduling splits all concurrently executing warps into fetch groups (e.g., 32 warps could be split up into 4 fetch groups of 8 warps each). The scheduling policy selects a single fetch

group and prioritizes warps from that fetch group over warps in other fetch groups. Warps within the same fetch group are scheduled in a round-robin fashion. This continues until all warps in the currently prioritized fetch group are stalled on a long latency operation. At this point, the next fetch group is selected (making the fetch group that used to be most prioritized now least prioritized) and the policy repeats. Note that the scheduling policy within a fetch group is round-robin, and switching from one fetch group to another is also done in a round-robin fashion (hence two-level round-robin). The key insight is that each fetch group reaches a long latency instruction at different points in time; as such, when warps in one fetch group are stalled, warps from another fetch group can execute, thereby hiding the latency. The overall result is reduced idle cycles (i.e., when all warps are stalled) leading to performance improvement over traditional pure round-robin scheduling policies.

I show that when combined, the Large Warp Microarchitecture and Two-Level Warp Scheduling significantly improve computational resource utilization resulting in a 19% performance improvement on average over traditional GPU architectures on a set of general purpose parallel applications.

1.2.2 New Instructions and Row Buffer Hint Bits for Fast Database Search

Database search represents an excellent example of an application with lots of parallelism, but suffers from several of the not-so-regular characteristics previously described. I target two basic types of database search in this thesis:

1) full table scan, and 2) index-based search. Full table scan is basically a filter operation that requires an entire column (or columns) of a table to be read, tested, and conditionally written to the output. Full table scans are costly operations but are required when the data being searched is unordered (i.e., no index available). As will be shown in Chapter 6, conditionally accumulating data to produce the output during a full table scan is problematic for SIMD architectures such as GPUs since it requires fast communication across SIMD lanes (e.g., intra-warp or intra-wavefront communication in NVIDIA/AMD terminology). I propose adding a *conditional accumulate* instruction to GPU ISAs, and the hardware to efficiently execute it, which accomplishes in one instruction what would otherwise take several instructions. My evaluation shows a 21% speedup for full table scans that produce large outputs.

Index-based search is a much more efficient way of searching but requires additional storage to index the data. Database indexes are typically organized as tree structures which are traversed during the search. Unlike full table scans which stream through data, the memory access patterns for tree traversal are unpredictable and highly irregular. Significant programmer effort has been devoted to optimizing tree search on GPUs [22, 26] resulting in very sophisticated algorithms and data layouts that attempt to remove some of the not-so-regular characteristics by efficiently utilizing SIMD hardware and exposing cache and memory (i.e., page/row buffer) locality. I show that despite these efforts, much of the data locality (specifically row-buffer locality) created by reorganizing the data is destroyed due to interference when

multiple index-based queries are executed simultaneously. I propose adding a programmer controlled row buffer locality hint bit to load instructions which the memory controller can use to better exploit row buffer locality. In addition, I propose a new *tree traverse* instruction which leverages the same hardware needed by the *conditional accumulate* instruction. During index-based tree search, threads within a warp must communicate values to collectively decide which path to go down next. *Tree traverse* accelerates this process. Therefore, subsequent requests are exposed to the memory system sooner allowing the memory controller to more efficiently exploit row-buffer locality. Combing row buffer locality hints with the new *tree traverse* instruction improves index-based query throughput by 39%.

1.3 Thesis Statement

The GPU architecture can be enhanced through mechanisms such as large warps, two-level warp scheduling, new instructions supporting fast intra-warp communication/decision making, and row buffer locality hint bits to better handle not-so-regular parallelism, thereby increasing the scope of applications that can be executed efficiently on a GPU, and in so doing, making the GPU a more viable platform not only for current parallel workloads such as databases, but also for future and emerging parallel applications.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides background information on GPU architectures. Chapter 3 discusses related work on executing not-so-regular parallel applications on GPUs. Chapter 4 describes the Large Warp Microarchitecture and Two-Level Warp Scheduling. Chapter 5 presents the results of those two mechanisms. Chapter 6 discusses related work and background information on executing database workloads on GPUs. Chapter 7 describes my proposals to speed up database search on GPUs. Chapter 8 presents the results of those proposals. Lastly, Chapter 9 concludes this thesis and discusses future research directions.

Chapter 2

Background on GPU Architectures

In this chapter, I present background information that provides some context for the rest of this dissertation. I first describe the architecture of a Graphics Processing Unit (GPU), with specific attention to the overall GPU core microarchitecture and pipeline, memory model, and existing conditional branch handling mechanisms. Although the exact GPU architecture and microarchitecture varies from design to design (e.g., NVIDIA GPUs versus AMD GPUs versus Intel GPUs etc.), I have tried to capture the basic essence of the GPU architecture and represent the baseline GPU used in evaluating the proposals presented in this dissertation.

2.1 Background on GPU Core Microarchitectures

In this section, I first describe in detail the microarchitecture of a single GPU core.[1] The GPU core is the basic building block of the entire GPU since many such cores may be replicated on the GPU chip with shared access to a last level cache (LLC) and memory.

[1]The term "GPU core" closely resembles a single Streaming Multiprocessor (SM) in NVIDIA's terminology [36]. It should not be confused with the NVIDIA term "CUDA core", which I refer to as simply a Functional Unit (FU).

2.1.1 GPU Core Pipeline

Figure 2.1 illustrates the baseline microarchitecture of a single GPU core composed of a scalar front end (fetch, decode) and a SIMD (Single Instruction Multiple Data) backend. As mentioned before, GPU programming models allow the programmer to create thousands of threads, each executing the same code. Before execution, those threads are grouped into fixed size SIMD batches called warps (by NVIDIA) or wavefronts (by AMD). Each warp consists of threads with consecutive thread IDs and the number of threads in the warp is equal to the SIMD width of the core (N in Figure 2.1). Many warps (M warps in Figure 2.1 for a total of M $\times$ N threads) are assigned to execute concurrently on a single GPU core.

In the fetch stage, the warp scheduler selects a warp from the set of ready warps. The baseline fetch scheduling policy uses a round-robin scheduler giving equal priority to each warp [14, 29]. Associated with each warp is a warp ID, a bit vector called the active mask, and a single Program Counter (PC). Each bit in the active mask indicates whether the corresponding thread in that warp is active. When a warp is originally created, all of its threads are active[2]. However, conditional branch instructions may lead to branch divergence which causes some threads to become temporarily inactive. This will be discussed in more detail in Section 2.1.3.

GPU cores process warps in a fashion similar to barrel processing[54, 51]

[2]If the total number of threads created by the programmer is not a multiple of the warp size, then one warp may be created without all threads active.

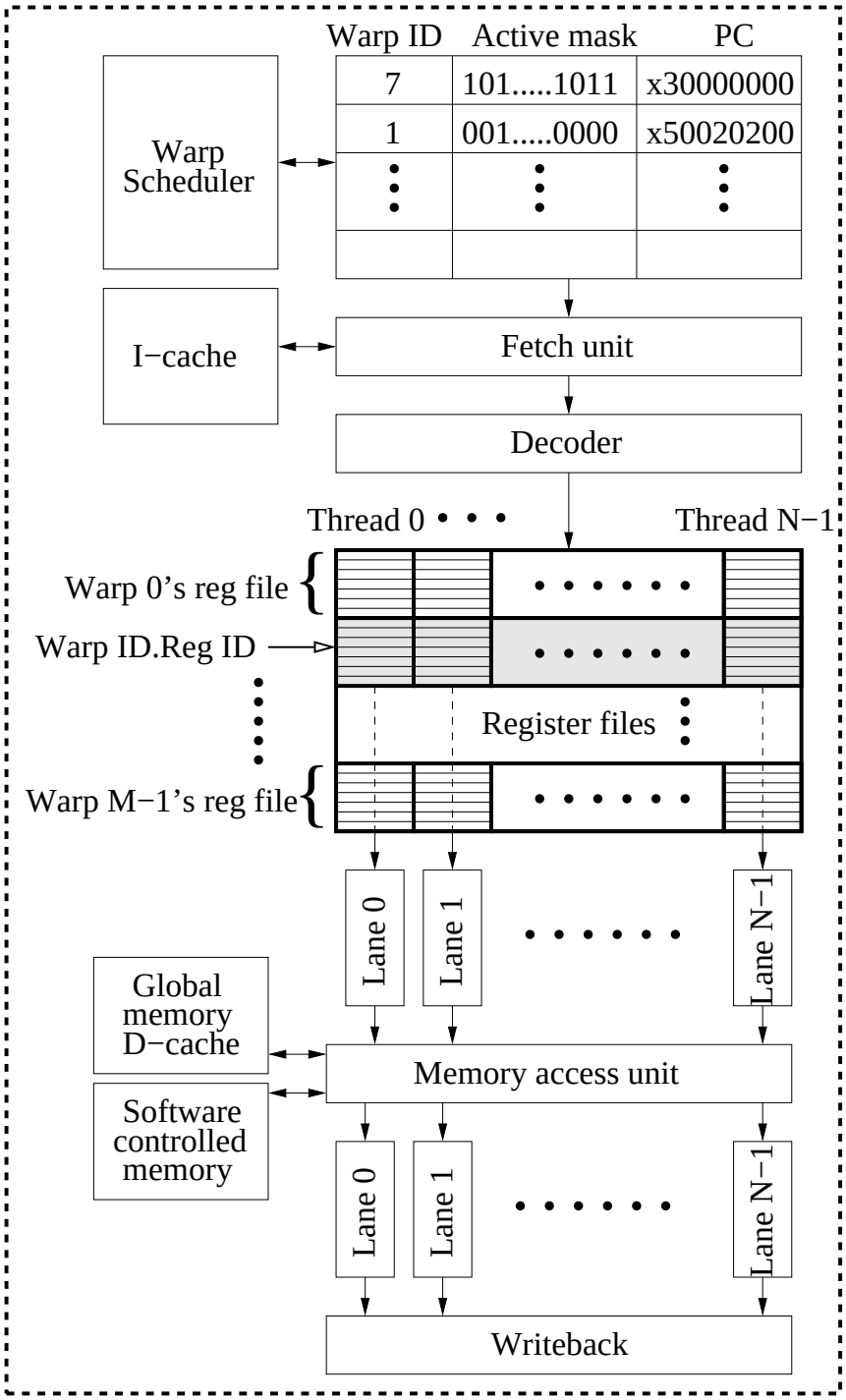


Figure 2.1: GPU core pipeline

where once a warp is selected in the fetch stage, it cannot be selected again (i.e., it is not ready) until that warp goes through the entire pipeline and completes execution of the instruction that was just fetched. Such a barrel processing model obviates the need for complex dependency checking logic and also branch prediction and misprediction recovery hardware, allowing more transistors to be spent on computational resources (i.e., functional units). Note that some GPU architectures relax this barrel processing model slightly by allowing a warp to be re-fetched sooner rather than waiting until the previous instruction completes execution. However, they still impose a several-cycle minimum warp re-fetch latency to avoid the need for branch prediction and misprediction recovery hardware.

After a warp is selected by the scheduler, the instruction cache is accessed and the fetched instruction is decoded, thereby completing the scalar portion of the pipeline. Next, register values for all threads in the warp are read in parallel from the register file indexed by the warp ID and the register ID as shown in Figure 2.1. These register values are fed into the SIMD backend of the pipeline (i.e., the computational resources or functional units) and are processed in parallel across multiple SIMD lanes. Once a warp reaches the final stage of the pipeline, the results of the instruction are written back to the register file (if necessary) and the warp's PC and active mask are updated before the warp is again considered for scheduling.

2.1.2 Memory Model

Figure 2.1 also illustrates the memory model for the baseline GPU core. All threads have access to global memory and data from memory can be cached on the chip. An entire cache line is read (or written) in parallel in a single memory transaction. Therefore, a warp accessing memory can be satisfied in a single transaction if all threads in the warp access data within the same cache line. If the threads within a warp access different cache lines (i.e., memory divergence), those accesses will be serialized resulting in stalls in the pipeline. For best performance, GPU programmers are encouraged to avoid memory divergence as best they can by restructuring the algorithm and/or layout of the input data to guarantee that threads within the same warp read contiguous and aligned values from global memory.

If one or more threads in the warp access a line not present in the cache, the warp stalls until all requested lines are fetched from memory. This can take several hundreds of clock cycles [41]. However, during this time, the warp is removed from the pipeline and put aside, allowing other warps to flow through the pipeline in an effort to hide memory latency.

In addition to the global memory data cache, there is also a software controlled on-chip scratchpad memory that can be used to avoid costly accesses to main memory. This on-chip scratchpad memory is usually highly banked (one bank per SIMD lane) so that threads in a warp can read (or write) data from (or to) this memory in parallel as long as there are no bank conflicts [41, 40]. This organization allows for more flexible gather/scatter

memory access patterns compared to global memory, but is still subject to bank conflicts. This memory corresponds to local memory in OpenCL [25] or shared memory in CUDA [41] since threads concurrently executing on the same GPU core may use this memory to communicate values. However, such communication must be carefully guarded by synchronization primitives inserted by the programmer.

2.1.3 Conditional Branch Handling on SIMD processors and GPUs

In this section I discuss the various mechanisms existing GPUs use to execute conditional code (e.g., if-then-else statements).

2.1.3.1 Predication

Predication, introduced by Allen et al. [3], turns control flow dependencies into data dependencies by associating instructions with a true/false predicate bit that enables/disables the destination write of that instruction. Today's GPU Instruction Set Architectures (ISAs) support predicated execution and extensively use it to execute simple conditional code blocks. GPU ISAs are equipped with per-thread predicate bits which can be sourced by subsequent instructions to enable or disable their execution. The following example illustrates how a simple if statement can be implemented through predication. Consider the following high level pseudo-code:

```
if(x < y){
  x++;
}
```

This simple if statement can be predicated as follows:

```
    CMP_LT  P1, R1, R2
@P1 ADD     R1, R1, #1
```

The first assembly language instruction compares the values in registers R1 and R2, and sets the 1-bit predicate P1 to 1 (i.e., true) if in fact the value in R1 is less than the value in R2. The next instruction increments R1, but *only* if P1 is true as denoted by the @P1 syntax preceding the ADD opcode. Recall that all 32 threads in a warp are executing this code sequence in lockstep. Therefore, the result of the compare instruction is 32 individual predicate bits (which can be viewed as a single 32-bit wide predicate register for the warp) some of which are true and the others false. By guarding the add instruction with predicate P1, only those threads in the warp for which the if condition evaluates to true execute the ADD instruction and write back the new incremented R1 value. The other threads mask off the ADD operation essentially treating it as a NOP. This leads to underutilization of the GPU core's computational resources since only a fraction of the SIMD lanes will be active when executing the predicated ADD instruction.

Another issue with predication is that even if *all* threads in the warp

fail the if condition and therefore must to skip the instruction(s) inside the if block, those instruction(s) will still be fetched, decoded, and flow through the pipeline but will be completely masked off at the execution and write back stages of the pipeline which is very wasteful and inefficient. This is why predication is used only for very simple and short conditional code constructs like the previous example. The two mechanisms discussed in the next sections do not suffer from this problem.

2.1.3.2 Multiple Program Counters

Another approach to support conditional code on SIMD processors, described by Diamos et al. [11] and implemented in Intel's Sandy Bridge GPUs [20], is to have multiple Program Counters (PCs) per warp (one for each thread in the warp, i.e., one per SIMD lane), and in addition a single global PC for the entire warp. For every instruction executed, the per-thread PC is compared to the global warp PC and only if they are equal (i.e., that thread is active) is the instruction corresponding to that thread (i.e., SIMD lane) executed, otherwise it is masked. Furthermore, only the active threads in a warp will have their per-thread PCs incremented to the next instruction when executing non-control-flow instructions. The warp global PC is also incremented to the next instruction. The per-thread PCs of inactive threads remain unchanged.

Special if-then-else control flow instructions must be added to the ISA to support this style of execution. For example, for a simple if statement

(such as the one from Section 2.1.3.1), the conditional branch instruction that a compiler would normally produce is replaced with a special *if* instruction. The semantics of this instruction update the per-thread PCs with the target address only for those threads that are to skip the if block. All other per-thread PCs are updated with the fall through path. The warp global PC is also updated with the fall through path, unless all threads that executed the if instruction are to skip the if block, in which case the warp global PC is also updated with the target address. In this manner, as instructions inside the if block are executed, the threads that skip the if block would have their instructions masked since their per-thread PC would not match the warp global PC. Once the last instruction in the if block completes, all per-thread PCs and the warp global PC will be the same (all equal to the PC of the first instruction after the if block). Execution will continue with the warp fully active (all per-thread PCs match the warp global PC). However, while instructions in the if block were executing, SIMD resources were underutilized just as was the case for predication.

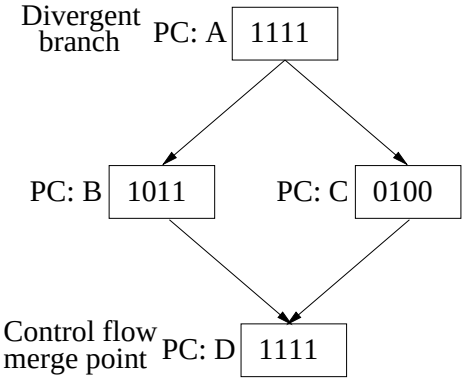
2.1.3.3 Stack-based Reconvergence on GPUs

The most common way to execute conditional code on GPUs, stack-based reconvergence, was originally proposed by Levinthal and Porter [30] in the CHAP graphics processor. Stack-based reconvergence is currently used in GPUs from both NVIDIA and AMD, and is also the branch-handling mechanism implemented for the simulated baseline GPU evaluated in this disserta-

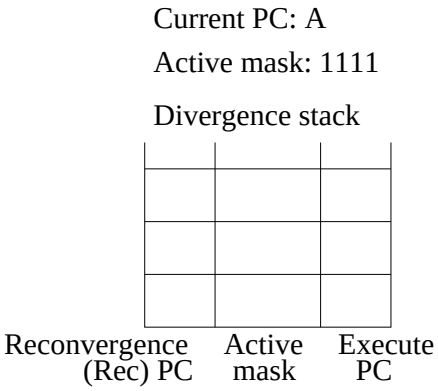
tion.

Figure 2.2 illustrates the baseline branch handling mechanism currently employed by GPU cores. In this example, there is only a single warp consisting of four threads, each of which is executing the same static code whose control flow graph is shown in Figure 2.2(a). Since a warp can only have a single active PC at any given time, when branch divergence occurs, one path must be chosen first and the other is pushed on a divergence stack associated with the warp so that it can be executed later. The divergence stack is also used to bring the warp back together once the divergent paths have been executed and all threads have reached a control flow merge (CFM) point. A divergence stack entry consists of three fields: a re-convergence PC, an active mask, and an execute PC. The execute PC and active mask fields will be copied into the current PC and current active mask of the warp when the divergence stack is popped. The re-convergence PC is used to indicate when the divergence stack should be popped. Executing the divergent paths serially but then re-converging at the CFM point can be accomplished as follows:

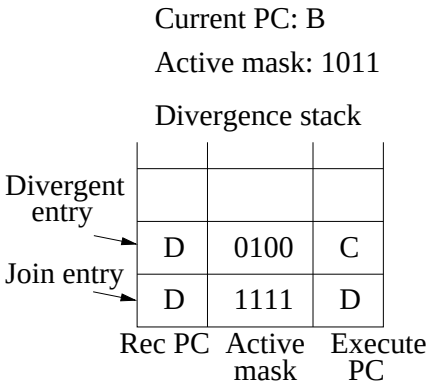
1) When the branch outcomes for all threads in the warp are not the same (i.e., a divergent branch), push a *join* entry onto the divergence stack. The join entry has both the re-convergence PC and execute PC equal to the compiler identified control flow merge (CFM) point of the branch. The active mask field is set to the current active mask (i.e., the active mask when the branch instruction was executed). This entry will be used to bring the warp back together (i.e., all four threads active) after the divergent paths are exe-



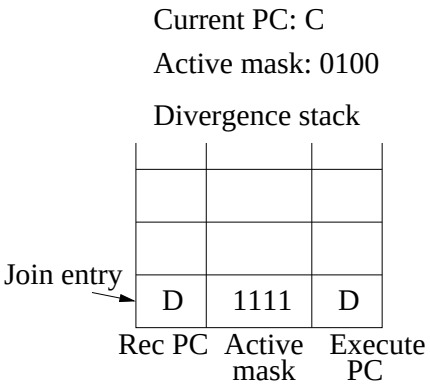
(a) Control flow graph



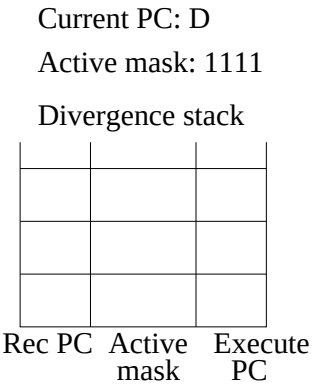
(b) Initial state



(c) After executing A



(d) After executing B



(e) After executing C

Figure 2.2: Stack based re-convergence for baseline GPU cores

cuted and all threads reach the control flow merge (CFM) point. Next, one of the two divergent paths is selected to execute first and the current PC and active mask of the warp are updated accordingly. Lastly, another entry, the *divergent* entry, is pushed on the divergence stack. The execute PC and active mask of this entry correspond to the divergent path that was not selected to be executed first. The re-convergence PC is set equal to the CFM point of the divergent branch.

2) When an instruction from a warp reaches the last stage of the pipeline, the warp's re-convergence stack is accessed to see if the next PC of the warp is equal to the re-convergence PC at the top of the stack. If so, the entry is popped, and the active mask and execute PC fields of the entry become the active mask and PC of the warp.

Figures 2.2(b) through (e) show the state of the current PC, the current active mask, and the divergence stack for a warp at relevant points in time as it executes the control flow graph of Figure 2.2(a). Inside each basic block of Figure 2.2(a) is a bit vector indicating whether or not the corresponding thread in the warp needs to execute the instructions in that basic block, i.e., the current active mask of the warp. The SIMD lanes are fully utilized as the instructions in block A execute but are underutilized as the divergent paths (blocks B and C) execute. Once all threads reach block D, the warp is restored to four active threads and execution again proceeds efficiently. However, the under-utilization of SIMD resources before re-convergence at the control flow merge point can lead to significant performance degradation, just as was the

case for the previous branch handling mechanisms described in Sections 2.1.3.1 and 2.1.3.2.

In summary, all of the branch handling mechanisms previously discussed allow GPUs to execute conditional code correctly, but not very efficiently. Performance suffers for applications with frequent conditional branches since only a fraction of the SIMD resources are utilized when conditional code executes. The rest of the functional units are wasted since their corresponding destination writes are disabled by either false predicate bits, per-thread PC to global warp PC mismatches, or disabled active mask bits.

Chapter 3

Related Work on Executing Not-So-Regular Parallel Applications on GPUs

3.1 Related Work on Optimizing Applications for the GPU at the Software Level

There has been plenty of work [49, 19] porting general purpose parallel applications to the GPU and optimizing them at the software level to remove some of the not-so-regular characteristics and achieve high performance. However, most of the applications successfully ported to the GPU tend to be computationally intensive grid-structured problems (i.e., stencil computations) with very regular control flow and memory access patterns. For such workloads, parallel threads not only start out executing the same code but remain on the same path (i.e., little or no branch divergence) throughout their execution. Furthermore, the programmer can organize data in memory in a SIMD friendly manner to avoid memory divergence and can also make efficient use of the GPU's on-chip memory since memory access patterns are known up front. Using such optimizations, previous work has shown that GPU execution can offer orders of magnitude speedup over single-threaded CPU execution. However, applications with input dependent branches and memory accesses cannot be optimized by the programmer since the dynamic execution

paths and memory access patterns of the parallel threads are not known until run-time. Hwu et. al [19] call "kernels with a large number of data-dependent control flows unsuitable for the GPU." They also admit to avoiding "applications with data-dependent memory-access patterns" since "such a kernel is unlikely to outperform a CPU."

Another example of optimizing parallel applications at the software level to make them better suited for GPU execution is the index-based tree search algorithm [26] I study in detail in Chapter 6. However, as I show in Chapter 7, such state-of-the-art algorithms remove *some* but not *all* of the not-so-regular characteristics.

In summary, parallel applications that suffer from not-so-regular parallelism do not execute efficiently on the GPU and cannot be completely optimized statically at the software level. Dynamic mechanisms are needed to improve performance for such applications.

3.2 Related Work on Executing Conditional Code on SIMD processors

Using a bit mask to execute conditional code in processors that exploit SIMD parallelism is an old concept. The Illiac IV [8] had a *mode bit* per Processing Element (PE) which either turned on or off a PE during execution of a single instruction. Likewise, CRAY-1 [48] had a *vector mask* register which was used to vectorize loops with if/else statements. These bit masks are akin to the *active mask* bits on GPU cores.

Allen et al. [3] introduced the idea of predicated execution which converts control flow dependencies into data dependencies. As discussed in Chapter 2, predicated execution has been extensively used on GPUs to implement conditional code. Predicated execution allows GPU cores to implement conditional code without branch instructions but does not deal with the problem of underutilized SIMD resources when SIMD lanes are masked off by predicate bits.

Smith et al. [52] introduced the concept of *density-time execution* whereby the time taken to execute a masked vector instruction is a function of the number of true values in the mask. False values in the vector mask register are skipped, thereby reducing the number of cycles it takes to execute the vector instruction. Rather than skipping false values, my proposal, the Large Warp Microarchitecture, finds active operations from threads in a large warp to fill the holes caused by branch divergence. Additionally, Smith et al. [52] studied several vector ISA alternatives for executing conditional code on vector processors. One of the methods uses compress operations which compacts data from memory or a register into a destination register under the control of a vector mask register. This is similar to the conditional accumulate instruction I propose in Chapter 7. However, unlike vector compress, conditional accumulate contiguously stores data directly in memory under the control of input predicate bits, and also counts the number of data items actually written to memory.

Kapasi et al. [23] introduced *conditional streams*, which allow stream

processors to filter an input stream before it is processed. However, this mechanism requires 1) communication between different SIMD lanes, and 2) effort from the programmer to declare conditional streams and implement new kernels to perform the filtering. In contrast, my approach, the Large Warp Microarchitecture 1) does not require communication between SIMD lanes and 2) is a pure hardware mechanism and therefore does not require any programmer effort.

Krashinsky et al. [28] proposed the Vector-Thread architecture (VT), which employs a control processor and a vector of virtual processors (VPs). The control processor uses vector-fetch commands to broadcast the same instruction to all the VPs. However, if divergence occurs, each VP also has the ability to direct its own control flow with thread-fetch commands. In this sense, the architecture is not strictly SIMD. In contrast, the Large Warp Microarchitecture is strictly SIMD and tolerates branch divergence by dynamically breaking down large warps into efficiently packaged sub-warps.

Meng et al. [31] proposed Dynamic Warp Subdivision (DWS) whereby upon divergence, two warp-splits are formed which can be scheduled independently. Although this does not increase SIMD resource utilization, it may increase memory-level parallelism since both sides of a divergent branch are executed concurrently. As such, DWS is orthogonal to the Large Warp Microarchitecture (LWM) and can be employed on top of the LWM by splitting a large warp upon branch divergence.

Diamos et al. [11] proposed SIMD re-convergence at thread frontiers.

Recall from Chapter 2 that after branch divergence, existing GPUs reconverge threads at the compiler identified control flow merge point of the divergent branch. However, Diamos et al. [11] propose reconverging at points earlier than the control flow merge point which improves performance for applications with unstructured control flow. This work is orthogonal to my proposal, the Large Warp Microarchitecture, and can be applied on top of it to further improve performance.

Fung et al. [14, 15] proposed Lane-Aware Dynamic Warp Formation (DWF), and were the first to come up with the idea of combining threads from different warps to address underutilized SIMD resources due to branch divergence on GPU cores. Therefore, I discuss their approach, and its limitations, in detail in the following section.

3.2.1 Dynamic Warp Formation

Basic Idea behind DWF: In DWF, warps start out exactly as in the baseline GPU core with consecutive threads grouped into a warp, and many such warps assigned to concurrently execute on a single core. However, in DWF, when a warp retires an instruction, it tries to merge its active threads with another warp that is in the list of warps waiting to be scheduled (the list of warps waiting to be scheduled is called the warp pool). In order to merge threads, there must be a warp in the warp pool whose PC value matches the next PC of the warp being retired. If such a warp is found, then merging is attempted. An active thread in the retiring warp can only be merged if the

same exact bit position in the active mask of the warp in the warp pool is zero (i.e., empty), hence the term lane-aware. For example, if the retiring warp's active mask is 1010 and the warp in the warp pool has an active mask of 1100, then only one thread can be merged. After merging, the warp in the warp pool would have an active mask of 1110, and the retiring warp would enter the warp pool with an active mask of 1000. Furthermore, if another warp retires whose next PC is the same as the previous two warps, the retiring warp can only be merged with the younger of the two warps in the warp pool (i.e., the warp with an active mask of 1000) since the hardware only allows merging with the youngest matching warp in the warp pool. I subsequently refer to this as *local merging*.

Divergence Handling in DWF: When a warp retires a conditional branch, there are potentially two different next PCs and active masks for the retiring warp. In the baseline GPU core, as well as the Large Warp Microarchitecture, one of the PCs and the corresponding mask is pushed on a divergence stack (after a join entry is first pushed on the divergence stack). However, DWF searches for a matching warp in the warp pool at both PCs and attempts to merge each active mask with the corresponding matching warp in the warp pool (if found). DWF does not use a divergence stack but rather relies on the local merging described above to bring warps back to their original thread count once all threads reach the control flow merge point of the divergent branch.

Scheduling in DWF: In addition to lane-aware merging, Fung et

al. [14, 15] also propose several scheduling policies for warps in the warp pool and show that the scheduling policy used is critical to the effectiveness of DWF. The key idea behind their scheduling policies is to keep a large group of threads together (i.e., at the same PC) in order to maximize the potential for merging threads. The two best performing policies are *majority* and *post dominator priority (PDPRIO)*. In the majority policy, the most common PC among all warps in the warp pool is calculated and all warps at that most common PC are scheduled before the next majority PC is calculated. In PDPRIO, warps with threads that have passed fewer post dominators (i.e., control flow merge points) are prioritized.

DWF has three main limitations 1) The local merging in DWF can be inefficient and often results in lost opportunities to increase SIMD efficiency, 2) DWF can cause significant additional memory divergence not found in the baseline GPU core, and 3) The scheduling policies described above have several unintended effects with certain types of control flow, which leads to not only lost opportunities for merging but also poor memory locality. I elaborate on these issues below.

Local Merging: As previously mentioned, DWF only permits a retiring warp to be merged with the youngest matching warp in the warp pool. This can result in inefficient merging by leaving holes in the active mask of older matching warps in the warp pool. This is especially problematic when diverged threads reach a control flow merge point. Both the baseline GPU core and the Large Warp Microarchitecture use a divergence stack to restore

each warp back to its original thread count when the threads reach a control flow merge point. However, DWF does not use a divergence stack and therefore relies on local merging to bring warps back to their original active thread count. Local merging is inefficient because it can merge a retiring warp only with the youngest matching warp in the warp pool. As a result, warps may not be brought back to their original thread count until several instructions after the control flow merge point has been reached. This reduces SIMD efficiency and can offset any benefit DWF achieved before the control flow merge point was reached, resulting in performance degradation compared to the baseline GPU core.

Additional Memory Divergence: Since DWF reassigns threads to warps during merging, after the first divergent branch, warps in DWF may no longer contain consecutive threads. This reassignment can persist even after threads reach a control flow merge point, causing additional memory divergence not found in the baseline GPU core. Recall that since the baseline GPU core uses a divergence stack to restore a warp to its original state after divergence, the thread to warp assignment never changes and therefore warps always consist of consecutive threads. In summary, DWF can destroy the coalesced memory accesses the programmer originally created since it reassigns the thread to warp assignment and never restores it.

Inefficiency of Scheduling Policies: The scheduling policies behind DWF try to keep a large number of threads together in order to maximize opportunities for merging threads. However, the proposed scheduling poli-

cies cannot always achieve this goal. As Fung et al. [14, 15] point out, the majority scheduling policy suffers from the problem of starving threads that take rarely executed paths. These threads must eventually be executed and SIMD resource utilization will be very low when they do. Starving threads also present a problem with memory locality. When the starved threads eventually execute, data that used to be present in the cache may not be there anymore and previously opened DRAM row buffers may now be closed. The PDPRIO scheduling policy can also have a similar effect. For applications with loop divergence (i.e., where each thread iterates over a loop for a different number of iterations) and also applications with imbalanced nested branching, the number of post dominators passed is not a good indication of which threads need to catch up. Therefore, PDPRIO can also result in excessive separation of threads for such applications. In summary, it is difficult to find an ideal scheduling policy that keeps threads together especially in the presence of biased branches, loop divergence, and/or imbalanced nested branching.

Due to these reasons, DWF has been superseded by Thread Block Compaction [13] (TBC). TBC has many similarities to my proposal, the Large Warp Microarchitecture [33], which will be described in detail in Chapter 4. I also quantitatively compare my work to TBC and describe the pros/cons of each approach in Chapter 5.

3.3 Related Work on Thread/Warp Scheduling

There has been previous work which analyzed and proposed scheduling policies for threads on Multi-Threaded (MT) or Simultaneously Multi-Threaded (SMT) cores [2, 55]. However, none of these policies were designed for scheduling warps on GPUs. GPU scheduling is unique in that warps tend to have much data locality among them. Also, GPUs support many more warp contexts compared to MT and SMT cores and allow zero-cycle context switching among all concurrently executing warps.

Lakshminarayana et al. [29] evaluate several possible fetch scheduling policies for GPUs. However, most of the scheduling policies they evaluate result in warps progressing uniformly through the program (similar to pure round-robin). In contrast, my two-level policy allows warps to arrive at a long latency instruction slightly apart from each other in time thereby effectively hiding long latencies and improving computational resource utilization.

Subsequent to two-level scheduling, Rogers et al. [47] proposed Cache-Conscious Wavefront Scheduling. This work proposes to reduce the number of active warps that are scheduled in order to avoid L1 cache thrashing when too many warps are simultaneously executing. In addition, this work studies other warp scheduling policies such as Greedy-Then-Oldest (GTO) scheduling and also two-level scheduling using GTO scheduling at each of the two levels instead of round-robin. This shows that having two levels of scheduling is a general concept that can be applied in combination with future warp scheduling proposals to further improve performance.

3.4 Related Work on Irregular Memory Access Patterns

As mentioned before, irregular memory access patterns are problematic for GPU cores. Recall that in traditional GPU cores, when threads in a warp want to access data in different regions of memory (i.e., different cache lines), the warp is stalled until all requests are serviced. If the multiple cache lines are all resident in the cache, multiple serial cache accesses must occur before the warp can continue execution. If none of the lines are in the cache, the warp is stalled and put aside and must wait until all memory requests are serviced before it can resume execution. In the worst case, a single warp of 32 threads can request data that is stored in 32 different cache lines (none of which are in the cache) and would be stalled a very long time until all 32 memory requests are serviced. To make matters more complicated, a warp with memory divergence can have some of the data cached, and the rest in memory. In such a case, the entire warp is still stalled until all requests to memory are serviced (even though some of the threads were ready to continue immediately). Clearly, applications with irregular memory access patterns are a problem for traditional GPU cores.

Previous work has been proposed to improve efficiency when the memory divergence is such that some threads in a warp have the data cached, and others do not. Tarjan et al. [53] proposed Adaptive Slip and Meng et al. [31] proposed Dynamic Warp Subdivision (DWS) which was also discussed in Section 3.2. In both cases, instead of stalling the entire warp, those threads whose

data was cached continue executing and will later reconverge with the threads that were stalled. This mechanism works well when only a few warps (e.g., less than 8) are concurrently executing on a core, but offers little improvement and is impractical when lots of parallelism is available (32+ warps executing on a single core). Furthermore, neither mechanism applies to the cases where memory divergence occurs but all the requested data is either cached or all the data is un-cached. Lastly, neither proposal targets another problem associated with irregular memory access patterns: poor row buffer locality. In contrast, my proposal to speed up index-based tree search specifically targets improving row-buffer locality and therefore is orthogonal to these proposals.

Michelogiannakis et al. [32] proposed Collective Memory Scheduling (CMS). This work attempts to improve performance by better handling the non-streaming memory access patterns that result from parallelizing stencil-based applications in a tile based manner across multiple cores. CMS coordinates accesses from different cores reading different tiles in an effort to improve row-buffer locality which may exist in the requests from different cores. Unlike my row buffer hint proposal, CMS applies to only tiled-based applications and also attempts to improve row-buffer locality by coordinating accesses from different cores rather than accesses from the same instruction stream.

Chapter 4

The Large Warp Microarchitecture and Two-Level Warp Scheduling

In this chapter I describe two proposals to improve computational resource utilization on GPU cores: the Large Warp Microarchitecture, and Two-Level Round-Robin Warp Scheduling. These proposals address two not-so-regular parallel application characteristics, divergent control flow and limited computational intensity, respectively. I first motivate the need for such mechanisms, then describe each mechanism in detail, and lastly discuss how the two proposals can be combined.

4.1 Motivation

The Problem: Underutilized Computational Resources: Despite the massively parallel architecture of GPU cores, the computational resources on a GPU core are often underutilized. For example, grouping threads into warps is only efficient if those threads remain on the same dynamic execution path (i.e., same PC) throughout their execution. Although this holds true for graphics applications, many general purpose parallel applications exhibit more complex control flow behavior among the parallel threads due to

frequent conditional branches in the code. Conditional branch instructions can cause threads in a warp to take different dynamic execution paths, or *diverge*. As described in Chapter 2, since existing GPU implementations allow a warp to have only one active PC at any given time, these implementations must execute each path sequentially. First, the warp executes the threads that follow the taken path of the branch (the not taken threads are masked off). Then the warp executes the threads that took the not taken path (masking off the taken path threads). This leads to lower utilization of SIMD resources while warps are on divergent control-flow paths because the warp must execute with a fewer number of active threads than the SIMD width of the core. This loss of efficiency continues until the divergent paths finish and a control flow merge point is reached. At this time, the warp is brought back to its original active thread count (i.e., the active thread count before the divergent branch instruction) and execution proceeds efficiently.

Another example of unused computational resources occurs when a GPU core is unable to effectively hide the latency of long latency operations (i.e., memory accesses) with computation from other warps. The warp instruction fetch scheduling policy employed on a GPU core can significantly affect the core's ability to hide such latencies. For example, commonly-employed scheduling policies that give equal priority to each warp (i.e., round-robin scheduling) tend to result in all the warps arriving at the same long latency operation at roughly the same time. Therefore, there are no other warps to execute to hide the latency with computation. On the other hand, allowing

warps to progress at very different rates can result in starvation and destroy the data locality among the warps. For example, data brought into the cache and row buffers opened by one warp are likely to be accessed again by other warps. However, allowing warps to progress very unevenly may destroy this locality.

Figure 4.1 illustrates the unused computational resources for a set of general purpose parallel benchmarks. Each benchmark is represented by a stacked bar indicating the percentage of cycles a certain number of the functional units are active. In this experiment, the SIMD width and warp size is 32, and 32 warps are concurrently executing on a single GPU core using a round-robin scheduling policy.

Branch divergence results in a reduction of the number of active threads

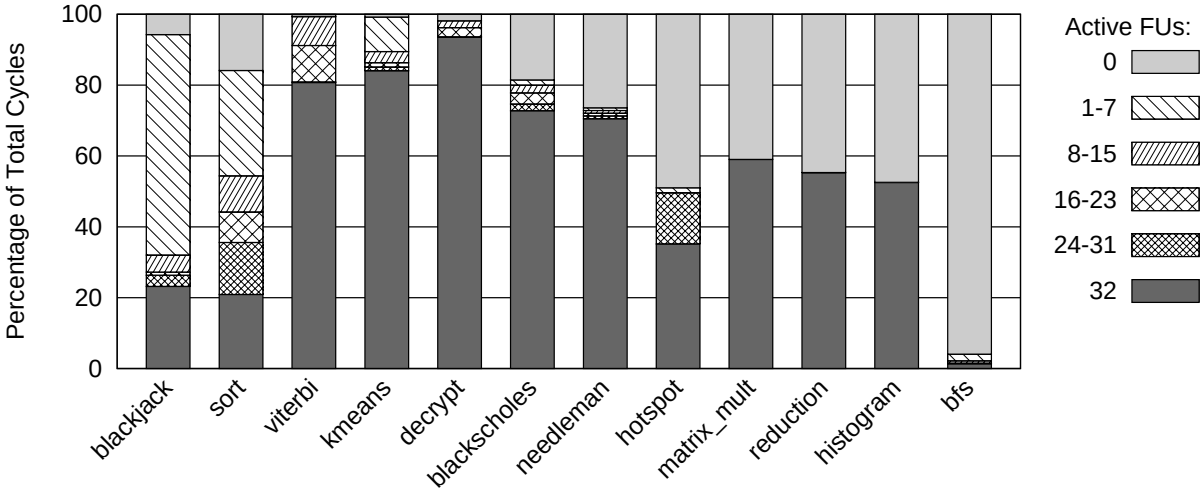


Figure 4.1: Computational resource utilization, SIMD width/warp size is 32

in a warp which leads to underutilization of the computational resources. The leftmost benchmarks suffer from this problem indicated by a large percentage of cycles where only a fraction of the FUs are active. On the other hand, the rightmost benchmarks suffer less from branch divergence but rather experience a significant fraction of cycles where none of the FUs are active (idle FU cycles). The main reason for these idle cycles is that all (or most) warps are stalled waiting on a long latency operation (e.g., a cache miss). Even with so many warps concurrently executing, several benchmarks show a significant fraction of idle cycles. For example, *bfs* stalls approximately 95% of the time.

Given this underutilization of computational resources, I propose the Large Warp Microarchitecture to improve performance for parallel applications that suffer from branch divergence. I also propose two-level warp scheduling to better overlap computation and memory latency which improves performance for those applications with a more balanced or low compute to memory ratio.

4.2 The Large Warp Microarchitecture

To alleviate the performance penalty due to branch divergence, I propose the *Large Warp Microarchitecture* (LWM). While existing GPUs assign several warps to concurrently execute on the same GPU core, I propose having fewer but correspondingly larger warps. The total number of threads and the SIMD width of the core stay the same. The key benefit of having large warps is that fully populated sub-warps can be formed from the active threads in a large warp even in the presence of branch divergence.

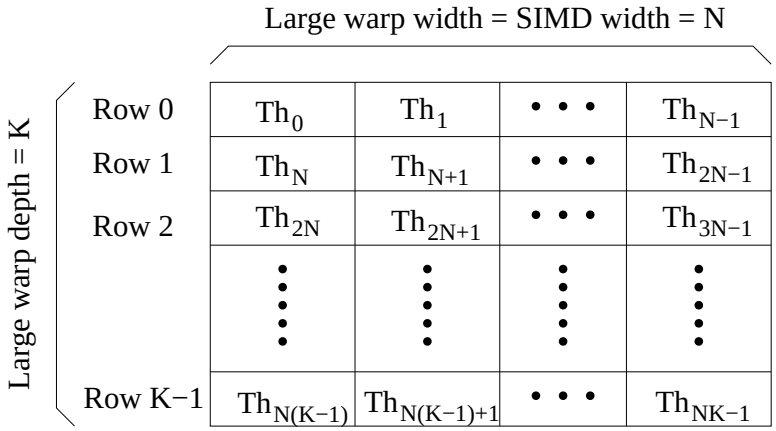


Figure 4.2: Large warp active mask

A large warp is statically composed of consecutive threads and has a single PC. It also has an active mask organized as a two dimensional structure where the number of columns is equal to the SIMD width of the core. Figure 4.2 shows the organization of the active mask of a large warp of size K $\times$ N threads executing on a core with a SIMD width of N. Each cell in Figure 4.2 is a single bit indicating whether or not the corresponding thread is currently active. Notice that the actual storage cost does not change compared to the baseline. The baseline GPU core would have K separate N-bit wide active masks instead (i.e., K separate warps).

Once a large warp is selected in the fetch stage, the instruction cache is accessed at the PC of the large warp and the instruction is decoded in the following cycle just as in the baseline GPU core. In parallel with decode, SIMD-width sized sub-warps are created which then flow through the rest of the pipeline. Figure 4.3 shows the hardware structures added to support the

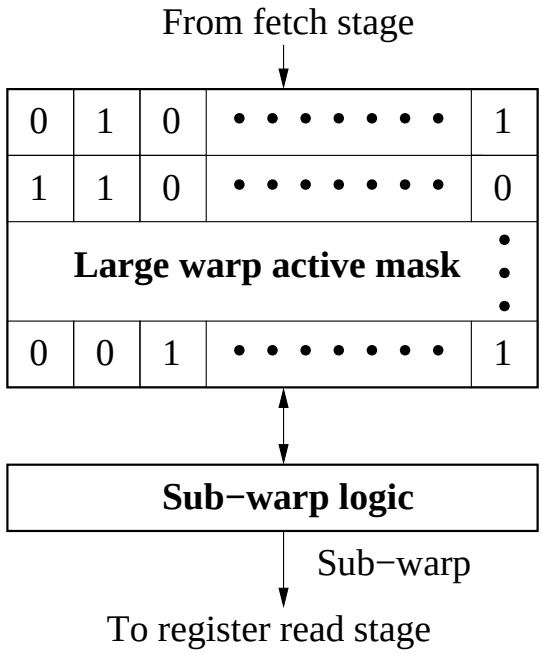
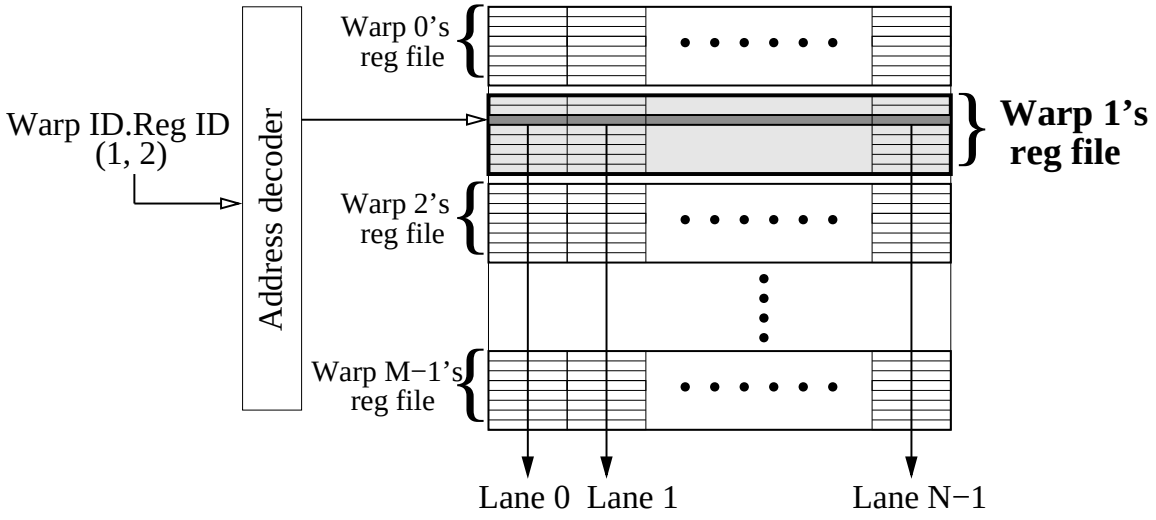


Figure 4.3: sub-warping logic

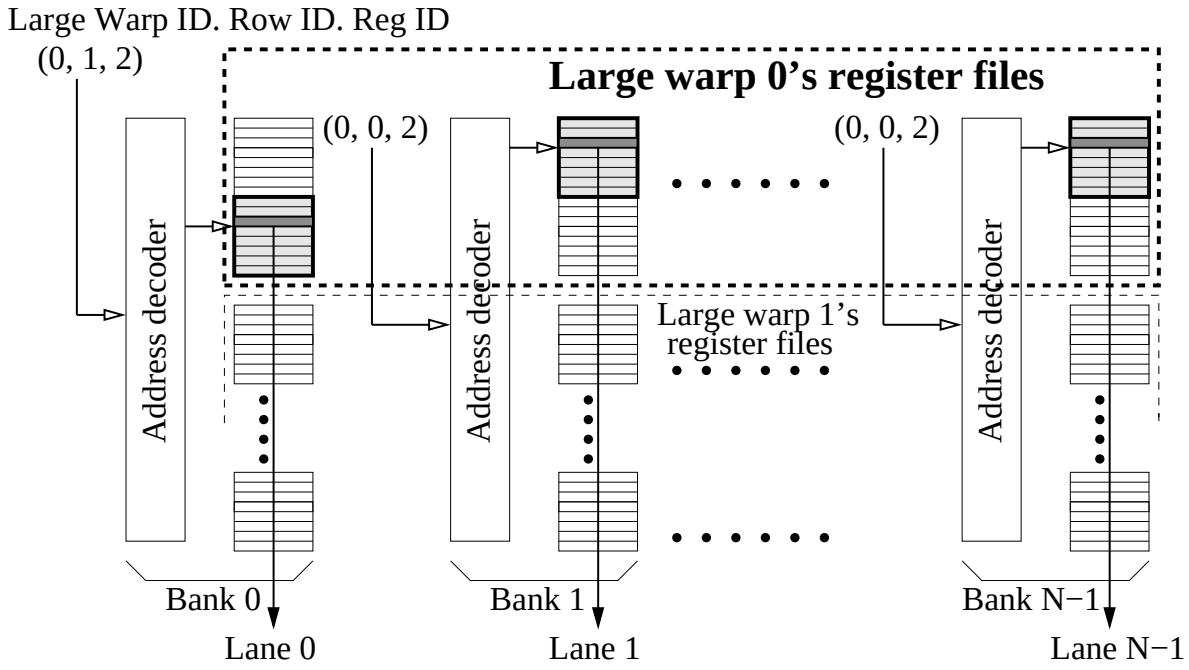
Large Warp Microarchitecture (LWM). When forming sub-warps, the goal is to pack as many active threads as possible into a sub-warp to best utilize the SIMD resources further down the pipeline. To accomplish this, specialized sub-warping logic examines the two dimensional active mask of the large warp and aims to pick one active thread from each column.

4.2.1 Sub-warp Creation

When determining how to pack active threads into a sub-warp, the design of the register file must be taken into consideration since it is imperative that the register values for a sub-warp can be sourced in parallel. Figure 4.4(a) shows the design of the register file for the baseline microarchitecture (no large



(a) Baseline register files



(b) Large warp microarchitecture register files

Figure 4.4: Large warp vs. baseline register file design

warps). Since consecutive threads are statically grouped into warps and this assignment never changes, the register file can be conceptually designed as a very wide single banked structure indexed by warp ID concatenated with the register ID as shown in Figure 4.4(a).[1] However, having a single address decoder does not give enough flexibility for the LWM to pack threads into sub-warps. Ideally, any set of active threads should be allowed to be packed into a sub-warp. However, this would require the register file to have a number of ports equivalent to the SIMD width of the core. Such a design would require considerable increase in area and power. Therefore, I use the register file design from Jayasena et al. [21] and Fung et al. [14, 15] and shown in Figure 4.4(b). The register file is split up into separately indexable banks, one bank per SIMD lane. This requires a separate address decoder per bank as shown in Figure 4.4(b). Each bank is indexed by the large warp ID, a new field called the row ID, and the register ID. The concatenation of the large warp ID and the row ID is the same width as the warp ID in the baseline GPU core. The new row ID field corresponds to which row of the two-dimensional large warp active mask the corresponding thread was selected from. Recall that during sub-warp creation, sub-warp logic attempts to select a single active thread from each column of the large warp active mask. As such, the threads selected and packed into a sub-warp may come from different rows. This design is cost-effective and provides much more flexibility in grouping active threads

[1]Such large SRAMs cannot be built for timing/energy reasons [4], so even the baseline register file is slightly banked.

into sub-warps than the baseline register file. Using this design, threads can be grouped into a sub-warp as long as they come from different columns in the large warp's active mask.

Figure 4.5 illustrates the dynamic creation of sub-warps from a large warp of 32 threads executing on a core with a SIMD width of four. Due to branch divergence, the large warp is shown with only a subset of its threads active. Each cycle, the hardware searches each column of the active mask in parallel for an active thread. The selected threads are grouped together into a sub-warp. Once an active thread is selected, the corresponding bit in the active mask is cleared. If there are still active threads remaining, a stall signal is sent to the fetch stage of the pipeline since the large warp has not yet been completely broken down into sub-warps. Once all bits in the active mask have been cleared, sub-warping for the current warp is complete and sub-warping for the next large warp (selected in the fetch stage) begins. Figure 4.5 illustrates how a large warp is dynamically broken down into four sub-warps over four successive cycles.

Note that the baseline GPU core would form eight different warps of four threads each rather than grouping all 32 threads into a large warp. Therefore, while the divergent code executes, SIMD resources will be underutilized since each warp contains fewer active threads than the SIMD width. However, with large warps, the inactive slots are filled with active threads during sub-warp creation. Therefore, only four efficiently packed sub-warps are created and SIMD resources are better utilized.

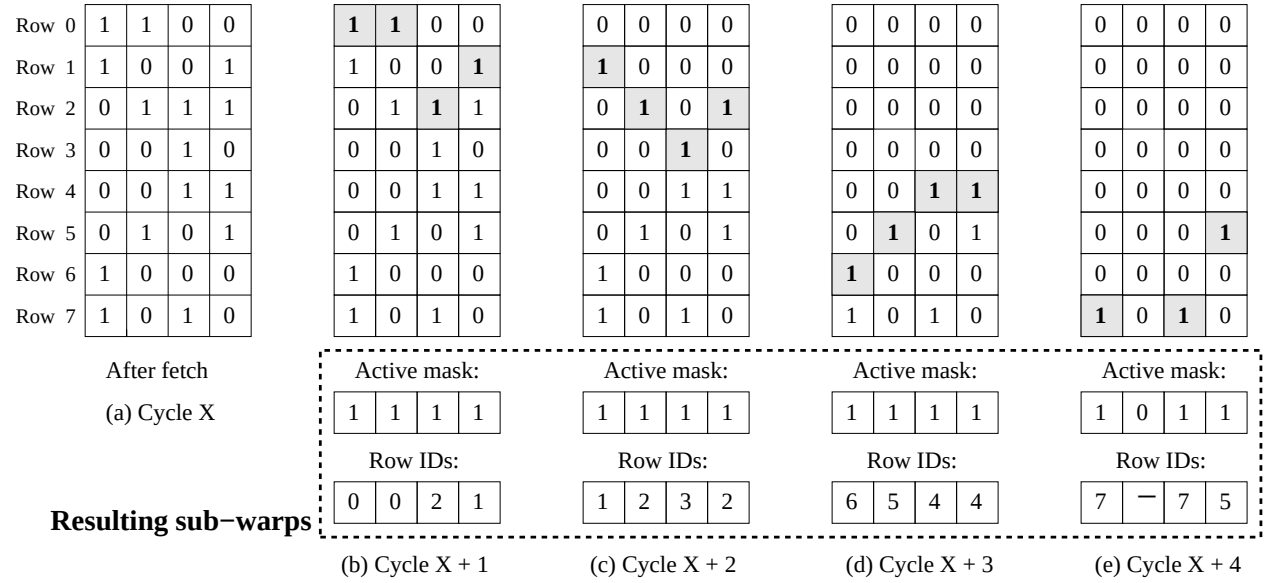


Figure 4.5: Dynamic creation of sub-warps

4.2.2 Barrel Processing with Large Warps

In the baseline GPU core, multiple warps are executed in a barrel processing fashion such that once a warp is selected by the scheduler in the fetch stage, it is not considered again for scheduling until the warp completes execution. For the Large Warp Microarchitecture, I impose a similar, but slightly relaxed model. Once a large warp is selected, it is not reconsidered for scheduling until the first sub-warp completes execution. This re-fetching policy requires hardware interlocks to ensure that register dependencies are not violated. Rather than having full register dependency checking hardware for each thread, I employ a single bit per thread to ensure that if a thread has been packed into a sub-warp that has not completed execution, it will not be packed into another sub-warp (corresponding to the next instruction) until

the previous sub-warp completes.

I use this relaxed re-fetching policy with one exception: conditional branch instructions. When a large warp executes a conditional branch, it is not re-fetched until *all* sub-warps complete. I do this since it is not known whether or not a large warp diverged until *all* sub-warps complete[2]. As such, re-fetching a large warp after the first sub-warp completes requires speculation and complicates the implementation without providing much benefit since an instruction from a different large warp can be executed instead to avoid bubbles in the pipeline.

4.2.3 Divergence and Reconvergence

Large warps handle divergence and reconvergence much the same way that baseline warps do. However, as mentioned before, when a large warp executes a branch instruction, it is not known for sure whether or not the large warp diverged until the last sub-warp completes execution. Therefore, the new active mask and the active masks to be pushed on the divergence stack are buffered in temporary active mask buffers. Once all sub-warps complete execution, the current active mask and PC of the large warp are updated and divergence stack entries are pushed on the large warp's divergence stack (if in fact the large warp diverged). The divergence stack is popped the same as the

[2]If any of the sub-warps before the last sub-warp diverges, we know at that instant that the large warp will diverge. Therefore an optimization could be applied here to re-fetch the large warp sooner in such cases. However, for ease of implementation, the default policy of waiting until the last sub-warp completes is always used.

baseline GPU core described in Chapter 2.

4.2.4 Unconditional Branch Optimization

When a warp in the baseline GPU core executes an unconditional control flow instruction (i.e., a jump), only a single PC update is needed. The same is true for large warps and therefore there is no need to create multiple sub-warps when a large warp executes a jump instruction. Creating multiple sub-warps is wasteful since successive sub-warps would just overwrite the same value to the PC that the previous sub-warp wrote. Thus, sub-warping for a large warp executing a jump instruction completes in just a single cycle, allowing sub-warping for the next large warp to begin sooner. Note that for a large warp size of 256 threads and a SIMD width of 32, this optimization saves 7 cycles (assuming that the large warp is fully populated) because it creates only one sub-warp instead of 8.

This optimization is a specific example of a more general optimization which applies to any instruction where the outcome is identical for all threads in the large warp (i.e., a scalar instruction [10]). The compiler could help identify other such scalar instructions (not just unconditional branches) that would also benefit from this optimization. In the evaluation presented in the next chapter, I only apply the optimization to jump instructions and leave identifying other scalar instructions as a future research direction which I discuss in Chapter 9.

4.2.5 Memory Divergence Optimization

As mentioned before in Chapters 2 and 3, a well known software technique employed by many GPU programmers is to coalesce accesses to global memory [41, 40, 27]. Many consider this the single most important performance consideration for programming GPUs [40]. Therefore, GPU programmers take special effort to ensure that consecutive threads access consecutive memory locations. If the addresses are not consecutive (i.e., divergent), each warp will require multiple transactions to global memory which can significantly degrade memory throughput and therefore performance. The same problem exists for accessing data that has been cached. If all addresses do not map to the same cache line, multiple serial cache accesses must be made due to cache port contention.

The LWM can cause additional memory divergence not found in the baseline since non-consecutive threads can be grouped together into a sub-warp during dynamic sub-warping. To avoid this, when a large warp is executing an instruction that accesses global memory, sub-warps are formed corresponding to each row of the active mask (i.e., no packing). Although this does not pack threads into sub-warps as efficiently, it avoids the additional memory divergence created if the regular sub-warping mechanism was used. I evaluate the effect of this optimization and the unconditional branch optimization in Chapter 5.

This completes the description of the Large Warp Microarchitecture (LWM). Detailed evaluation of the LWM is presented in the next Chapter.

4.3 Two-level Warp Scheduling

In this section I describe my other proposal, two-level warp scheduling, which aims to better overlap computation and memory latency, thereby improving performance for applications with a more balanced or low compute to memory ratio.

GPU cores concurrently execute many warps on the same core which helps avoid stalls due to long latency operations. However, the warp instruction fetch scheduling policy employed on the GPU core can considerably affect the core's ability to hide long latencies. Therefore, I propose a new two-level round-robin scheduling policy which more effectively hides long latencies and therefore reduces idle functional unit (FU) cycles. I first describe my new scheduling policy in the context of the baseline GPU core (not the LWM) and later describe how the two can be combined.

The baseline GPU core uses a round-robin warp instruction fetch policy giving equal priority to all concurrently executing warps [29, 14]. This policy results in warps progressing through the program at approximately the same rate which can be beneficial since warps tend to have a lot of data locality among them.[3] When one warp generates a memory request, other warps are likely to produce memory requests that map to that same row buffer. This row buffer locality can be exploited as long as the requests are generated close

[3]GPU programmers are encouraged to make consecutive threads access consecutive memory locations so that memory accesses can be coalesced [41, 27], therefore requests from different warps often have significant spatial locality.

enough to each other in time. A fair round-robin policy allows this to happen whereas a scheduling policy that results in very uneven warp progression could destroy such locality. However, a pure round-robin scheduling policy also tends to make all warps arrive at the same long latency operation at roughly the same time. Since all (or most) of the warps are stalled, there are not enough active warps to hide the long latency with computation, resulting in several idle FU cycles.

To this end I propose a two-level round-robin scheduling policy. The policy groups all concurrently executing warps into fixed size fetch groups. For example, 32 warps could be grouped into 4 fetch groups (with fetch group IDs 0–3) each with 8 warps. The scheduling policy selects a single fetch group to prioritize (let's say fetch group 0) and warps in that fetch group are given priority over warps in other fetch groups. More specifically, fetch group 0 is given the highest priority, fetch group 1 the next highest, and so on. Warps within the same fetch group have equal priority and are scheduled in a round-robin fashion amongst each other. Once *all* warps in the highest prioritized fetch group are stalled on a long latency operation, a fetch group switch occurs giving fetch group 1 the highest priority, fetch group 2 the next highest, and so on. Fetch group 0, which used to have the highest priority, now has the lowest.

Note that the scheduling policy within a fetch group is round-robin, and switching fetch group priorities is also round-robin (hence the name two-level round-robin). Prioritizing fetch groups prevents all warps from stalling at

the same time. Instead, a smaller subset of warps (i.e., a fetch group) arrives at the stall leaving other warps to execute while warps in one fetch group are stalled. Since both levels of scheduling are round-robin, row-buffer locality among warps remains high just as in conventional round-robin scheduling.

Figure 4.6 shows execution on a GPU core (a) with round-robin scheduling, and (b) with two-level scheduling. In this example, there are 16 total warps. With round-robin, all warps progress evenly through the compute phase of the program but then all stall waiting on data from memory. However, two-level scheduling with 2 fetch groups of 8 warps each reduces the number of idle cycles as shown in Figure 4.6(b). With two-level scheduling, warps in fetch group 0 proceed through the computation in half the time it took all 16 warps and therefore reach the stalling point sooner. Since all warps in fetch group 0 are stalled, a fetch group switch occurs and warps in fetch group 1 begin to execute their compute phase while requests created by fetch group 0 are serviced. This results in better overlap between memory latency and computation thereby improving performance.

For two-level scheduling to be effective, the fetch group size must be set judiciously. The fetch group size should have enough warps to keep the pipeline busy in the absence of long latency operations. Recall from Chapter 2 that the baseline GPU core uses a barrel processing model where once a warp is selected in the fetch stage, it is not reconsidered for scheduling until it completes execution. Therefore, the minimum fetch group size is equal to the number of pipeline stages. Having too large a fetch group size limits the

effectiveness of two-level scheduling for two reasons: 1) A larger fetch group takes longer to progress through computation than a smaller fetch group and therefore will not reach the stalling point as soon, and 2) A larger fetch group implies a greater number of warps stalling at the same time, leaving fewer warps to hide the latency. I evaluate the effect of fetch group size in the results presented in Chapter 5.

4.4 The Large Warp Microarchitecture and Two-Level Scheduling

The Large Warp Microarchitecture (LWM) and two-level scheduling can be combined. In the results presented in Chapter 5, I show that when the LWM is evaluated independently, the best performing large warp size is 256

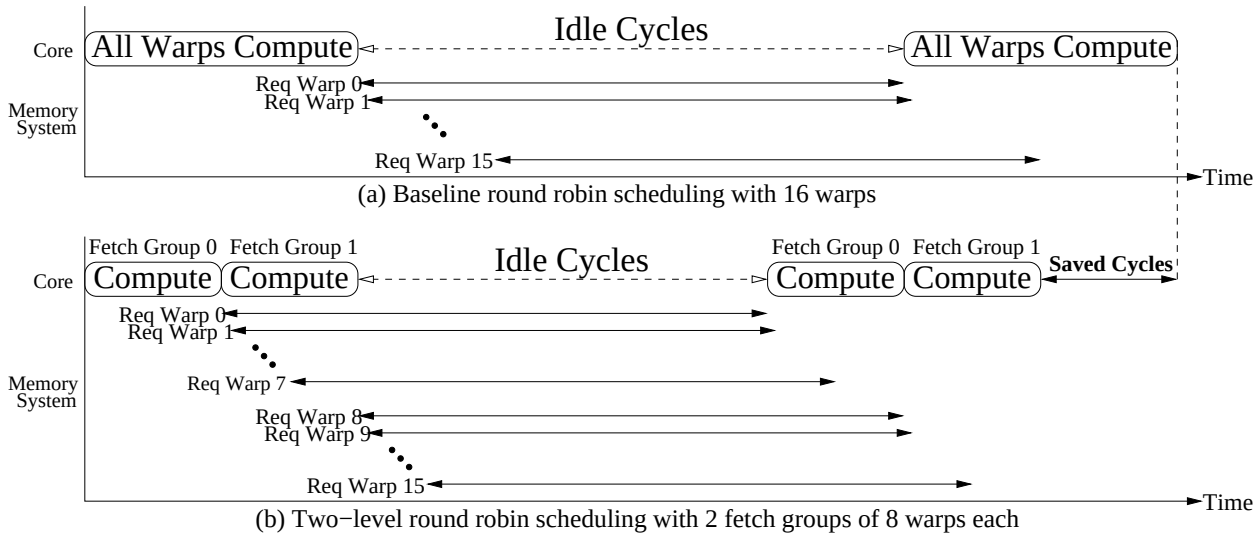


Figure 4.6: Baseline round-robin vs two-level round-robin scheduling

threads. Likewise, the best performing fetch group size for two-level scheduling is 8 regular-sized warps (i.e., 256 total threads since there are 32 threads per warp in the baseline). Therefore, when the two mechanisms are combined, the best performing configuration is a large warp size of 256 threads, and a fetch group size of only a single large warp.

However, the combination of the large warp re-fetch policy for branch instructions (i.e., waiting for all sub-warps to complete) and two-level scheduling with a fetch group size of one large warp can be problematic and limit the effectiveness of the combination. For example, consider an application that has no stalls after a short warm-up period. With no stalls, two-level scheduling continues prioritizing a single large warp until the entire program completes. Only then will a fetch group switch occur. This will ultimately result in a single large warp having to execute the entire program with no other large warps active (since they all finished executing the program). Having only one large warp active for such a long time is problematic since every time a conditional branch instruction is fetched, the large warp must wait until all sub-warps complete before being re-fetched thereby introducing several bubbles in the pipeline. Note that this is not an issue for two-level scheduling alone (i.e., with regular sized warps). Two-level scheduling in isolation would also result in a single fetch group left to execute, but a single fetch group (i.e., 8 regular sized warps) is still enough to effectively utilize the pipeline. Therefore this problem is unique to the combination of large warps and two-level scheduling where the fetch group size is a single large warp.

To alleviate this, I implement a two-level scheduling timeout rule whereby if a single large warp is prioritized for more than *timeout* instructions, I preemptively invoke a fetch group switch. This rule only applies to the combination of the LWM and two-level scheduling with a fetch group size of one large warp. Note that for most applications, the timeout is never invoked. However, for branch-intensive applications with few long latency stalls (e.g., the *blackjack* benchmark used in my evaluations), this rule helps significantly by bounding the time only a single large warp is active. I found empirically that 32K instructions works well for the timeout period.

4.5 Hardware Cost

Most of the hardware cost for the LWM comes from restructuring the register file. As shown in Section 4.2, instead of a single address decoder, the LWM requires a separate address decoder per SIMD lane. Previous work [21, 14, 15] estimates that such a design results in little die area increase. Jayasena et al. [21] propose stream register files with indexed access which require dedicated row decoders for each bank instead of a single shared decoder. They show that this results in an 11% to 18% increase in register file area which corresponds to a 1.5% to 3% increase in chip area of the Imagine processor [24]. Fung et al. [15] show similar results with an estimated 18.7% increase in register file area, corresponding to 2.5% of GPU area.

In addition to the register file overhead, there are a few extra storage structures required by the LWM not present in the baseline GPU core. As

explained in Section 4.2.1, sub-warp creation is done in parallel with decode by searching the columns of the two dimensional active mask of a large warp and clearing the bits corresponding to the selected threads. The bits in the large warp's actual active mask cannot be cleared and therefore a copy must be made before the large warp can be broken down into sub-warps. For large warps of size 256 threads, this corresponds to 256 bits of storage. In addition, as explained in Section 4.2.3, the Large Warp Microarchitecture uses temporary active mask buffers while executing branch instructions. Since a temporary buffer is required for each path of the divergent branch, this corresponds to 512 bits of storage. Lastly, the LWM requires extra bits and some simple logic for dependency handling as explained in Section 4.2.2. This additional storage amounts to just a single bit per thread. The total additional storage cost is 256+512+1024 bits (i.e., 224 bytes).

Two-level warp scheduling requires negligible storage cost. The only change is a simple logic block in the fetch stage implementing two-level round-robin scheduling.

Chapter 5

Evaluation of Large Warps and Two Level Scheduling

In this chapter I quantitatively evaluate both the Large Warp Microarchitecture and two-level scheduling. I first present my evaluation methodology, and then present overall performance results. I also present detailed evaluation of each mechanism independently.

5.1 Evaluation Methodology

I use a cycle-level simulator that simulates parallel x86 threads, each executing the same compute kernel. For the results presented in the rest of this chapter, I simulate a single GPU core concurrently executing 1024 threads. Table 5.1 presents the system parameters used in my simulations for the baseline GPU core.

Since x86 does not have instructions to aid with branch divergence/reconvergence of parallel threads like GPU ISAs do [39], I created instrumentation tools to identify conditional branch instructions and their control flow merge points. I used a similar procedure to identify barrier synchronization points since x86 does not support single instruction barrier synchronization

Scalar front end	1-wide fetch, decode stages, round-robin warp scheduling 4KB single-ported I-Cache
SIMD back end	In order, 5 stages, 32 parallel SIMD lanes
Register file and on-chip memories	64KB register file 128KB, 4-way, 1-cycle D-Cache, 1 read/write port, 128-byte line size 128KB, 32-banked scratchpad memory (128 bytes per thread)
Memory system	Open-row, first-come first-serve scheduling, 8 banks, 4KB row buffer 100-cycle row-hit, 300-cycle row-conflict, 32 GB/s memory BW

Table 5.1: Baseline GPU core and memory configuration

Benchmark	Description	Input Set
blackjack	Simulation of card game	52-card deck per thread
sort	Parallel bucket sort of a list of integers	1M random integers
viterbi	Algorithm for decoding convolutional codes	4M encoded bits
kmeans	Partitioning based clustering algorithm	16K 1-dimensional 8-bit points
decrypt	Advanced Encryption Standard decryption	256K AES encrypted bytes
blackscholes	Call/put options pricing	Initial data for 512K options
needleman	Optimal alignment for 2 DNA sequences	2 DNA Sequences, length 2048
hotspot	Processor temperature simulation	512x512 grid of initial values
matrix_mult	Classic matrix multiplication kernel	2 256x256 integer matrices
reduction	Reduce input values into single sum	32M random boolean values
histogram	Binning ASCII characters	16M ASCII characters
bfs	Breadth first search graph traversal	1 million node arbitrary graph

Table 5.2: Benchmarks

present in GPU ISAs [39].

I created parallel applications adapted from existing benchmark suites including Rodinia [9], MineBench [34], and NVIDIA's CUDA SDK [37] in addition to creating one of my own (blackjack). Each benchmark was parallelized using POSIX threads (Pthreads) and compiled with Intel's ICC compiler. I optimized each benchmark for GPU execution using principles found in [49] and [27]. Each benchmark runs to completion and consists of 100-200 million

dynamic instructions across all 1024 threads. Table 5.2 lists the benchmarks (along with input set) used in this study.

The metric I use to compare performance is retired *instructions per cycle* (IPC). Note that when a warp (or a sub-warp) executes an instruction, I treat each active thread in the warp as executing a single instruction. Therefore, if the warp (or sub-warp) size is 32, the maximum IPC is 32.

5.2 Results

In this section I provide the overall results of the Large Warp Microarchitecture and two-level scheduling. I also compare my work to recent state of the art work in divergent control flow handling on GPUs, Thread Block Compaction (TBC) [13]. TBC groups multiple regular-sized warps into a block and synchronizes them at every branch instruction. Special hardware is used to dynamically create new warps (i.e., compact into fewer warps) based on the active mask of all warps in the block after the divergent branch.

5.2.1 Overall IPC Results

Figures 5.1 and 5.2 show IPC and functional unit utilization for the baseline (32 warps of 32 threads each, round-robin scheduling), Thread Block Compaction (TBC), Large Warp Microarchitecture (LWM), two-level scheduling (2Lev), and Large Warp Microarchitecture combined with two-level scheduling (LWM+2Lev). Note that the SIMD width (32) and total thread count (1024) is the same for each configuration. For TBC, I used a block size of 256

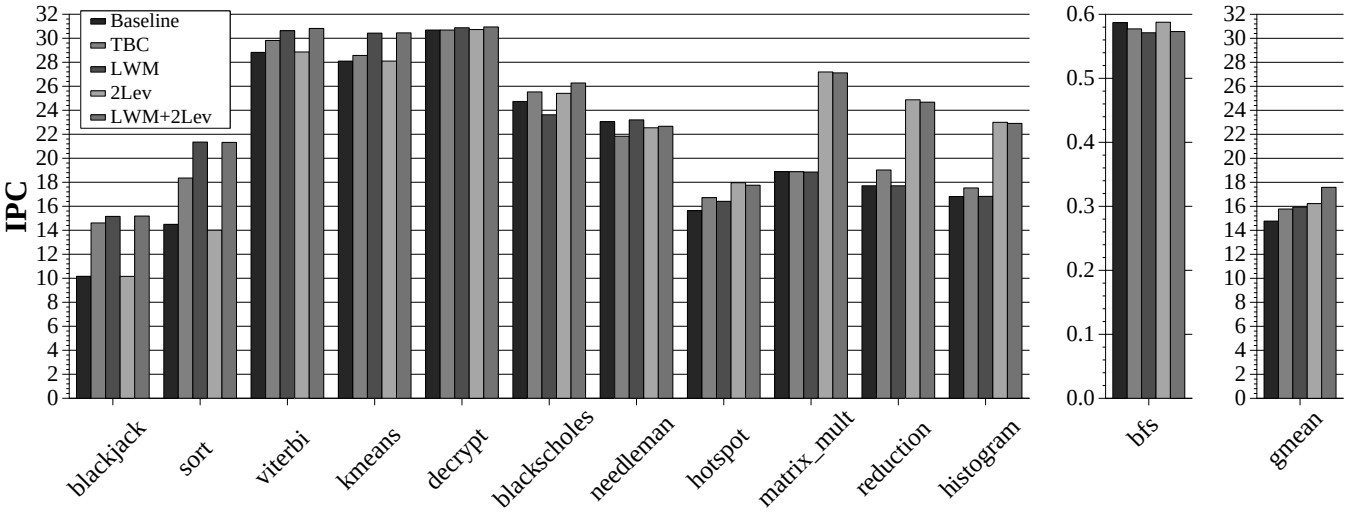


Figure 5.1: Performance

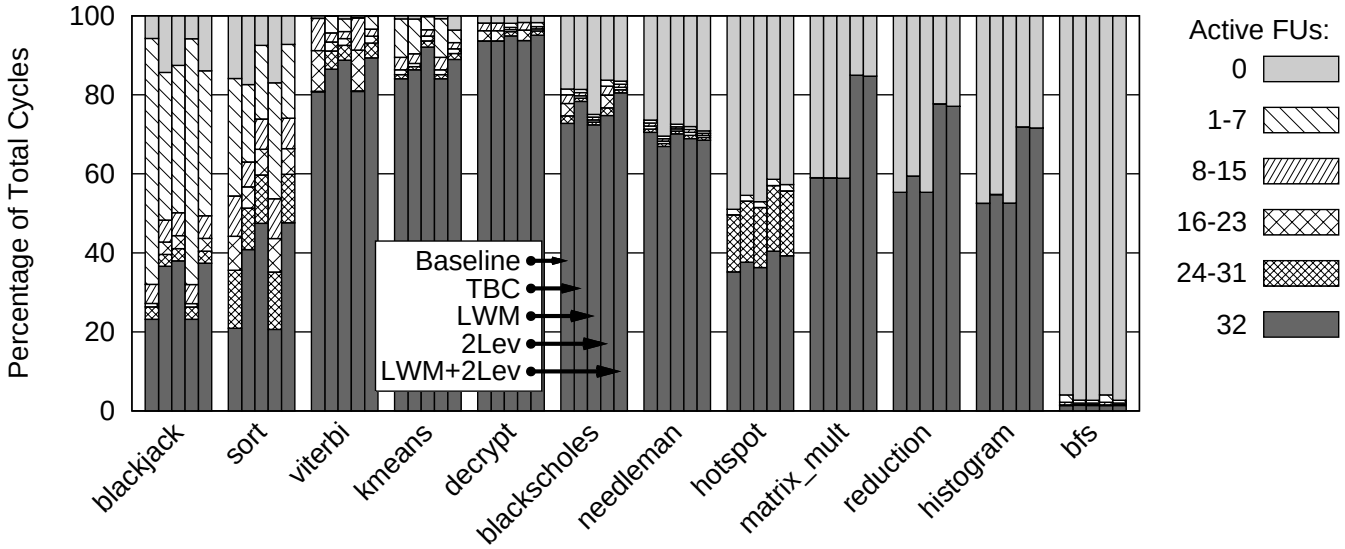


Figure 5.2: Functional unit utilization

threads and sticky round robin (SRR) scheduling, which performed slightly better than round-robin in my evaluation. For the LWM, I created 4 large warps of 256 threads each. For two-level scheduling only, I set the fetch group size to 8 (i.e., 4 fetch groups, each consisting of 8 warps). For the combination of LWM and two-level scheduling, I again formed 4 large warps of 256 threads each and set the fetch group size to 1 (i.e., 4 fetch groups, each consisting of 1 large warp).

As expected, the LWM (third bar) significantly improves performance for branch-intensive applications (the leftmost 4 benchmarks), whereas two-level scheduling (fourth bar) does not provide much benefit compared to the baseline for these applications. The reason for this is that these benchmarks make very good use of the on chip data cache and private memory and therefore do not suffer much from long latencies. However, they do contain frequent divergent branches which is the main reason for performance degradation for these applications. This is justified by looking at the computational resource utilization for these applications in Figure 5.2. There are relatively few idle cycles (0 active FUs) for these benchmarks even in the baseline architecture, however they do have a significant number of cycles where only a small portion of the FUs are active. The LWM improves this by efficiently packing active threads into sub-warps, thereby increasing SIMD utilization and improving performance. TBC also improves performance for these applications but not as much as the LWM does. There are two main reasons for LWM's edge. First, the LWM benefits from the optimization for unconditional branch instructions

discussed in Section 4.2.4. TBC cannot benefit from this optimization since it does not keep a large number of threads together between branch instructions. After compacting threads after a branch, warps are treated as individual scheduling units and therefore this optimization does not apply. Second, when a large warp executes a predicated instruction (e.g., cmov in x86), it creates efficiently packed sub-warps based on the predicated active mask, whereas TBC does not efficiently execute predicated code since it only performs compaction after divergent branch instructions.

The rightmost benchmarks show the opposite behavior. These benchmarks suffer from long latency stalls and therefore the LWM provides no benefit but two-level scheduling effectively reduces idle FU cycles as shown in Figure 5.2. Two-level scheduling results in only a subset of warps (i.e., fetch group) stalling at the same time, allowing warps in different fetch groups to effectively hide the latency. In addition, row buffer locality remains high (hit rate within 1.7% of traditional round-robin on average) since at each level of scheduling a round-robin policy is used. TBC, even with SRR scheduling, does not provide as much benefit due to the synchronization required at each branch instruction.

In summary, LWM alone improves performance by 7.9%, two-level scheduling alone improves performance by 9.9%, and when the two mechanisms are combined, the benefits of each are preserved resulting in 19.1% performance improvement over the baseline and 11.5% over TBC.

5.2.2 Large Warp Microarchitecture Analysis

In this section I provide results relating to only the Large Warp Microarchitecture (LWM) and use round-robin scheduling among the large warps (not two-level) in order to isolate the effects of the LWM.

5.2.2.1 Varying the Large Warp Size

I vary the warp size from the baseline of 32 threads per warp to a maximum of 512.[1] As seen in Figure 5.3, increasing warp size improves performance up to 256 threads. Larger warp sizes give more potential for sub-warping logic to create efficiently packed sub-warps and therefore in general, are beneficial. However, extremely large warp sizes can also be harmful. Having too large a warp size is inefficient for cases where most of the threads have reached the reconvergence point of a divergent branch and are waiting on just a few threads to arrive so that reconvergence can be done and execution can continue. For example, if each thread of a large warp executes a loop for a different number of iterations (i.e., the number of iterations is data dependent), at some point almost all of the threads will have exited the loop but will have to wait and sit idle (on the divergence stack) while the last few threads finally exit. Only then can the reconvergence stack be popped and execution of the reconverged large warp continue. This problem becomes more severe as large warp size increases since a greater number of threads will be sitting idle on the divergence

[1] All 1024 threads in a warp is not evaluated since the branch re-fetch policy would result in inefficient use of the pipeline.

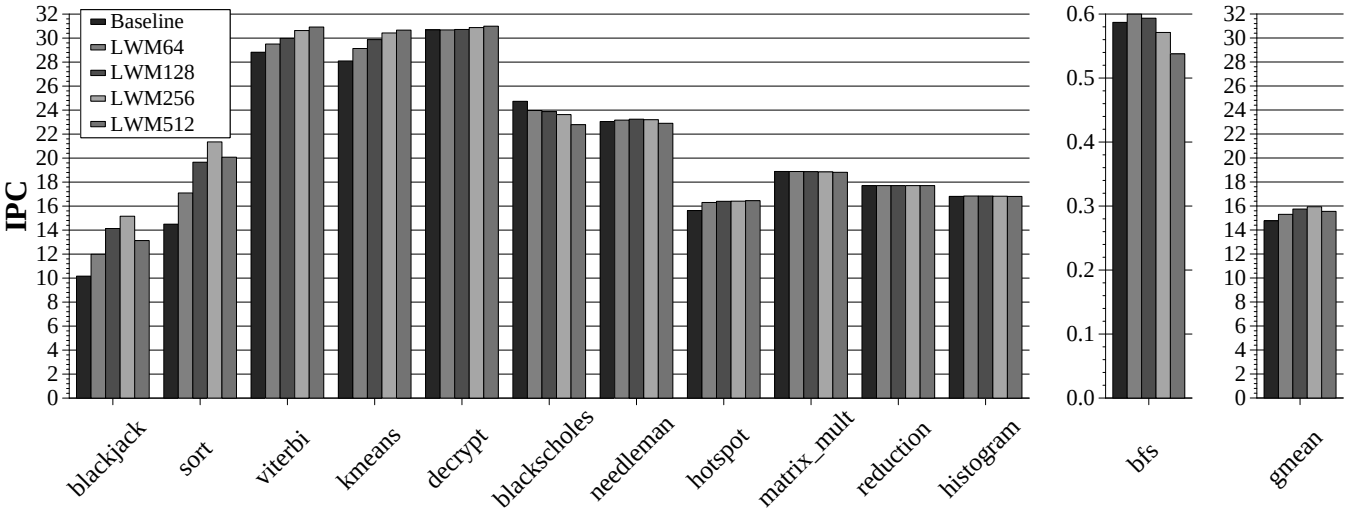


Figure 5.3: Effect of large warp size

stack until the final thread exits the loop. I find this behavior prominent in *blackjack* and *sort*, explaining why 256 threads performs better than 512. On the other hand, a large warp size of 512 slightly outperforms 256 for a few of the benchmarks which don't exhibit the loop divergence discussed above but rather have more regular if-then-else control flow constructs. Having too small a large warp size (64, 128 threads) does not provide enough opportunity for sub-warping logic to efficiently pack threads into sub-warps. Overall, I find a large warp size of 256 balances these trade-offs well and provides the best average performance.

5.2.2.2 Effect of One Thread per Column Sub-warping Restriction

Recall from Chapter 4 that due to register file complexity, I imposed the restriction of selecting only one thread per column of the large warp active

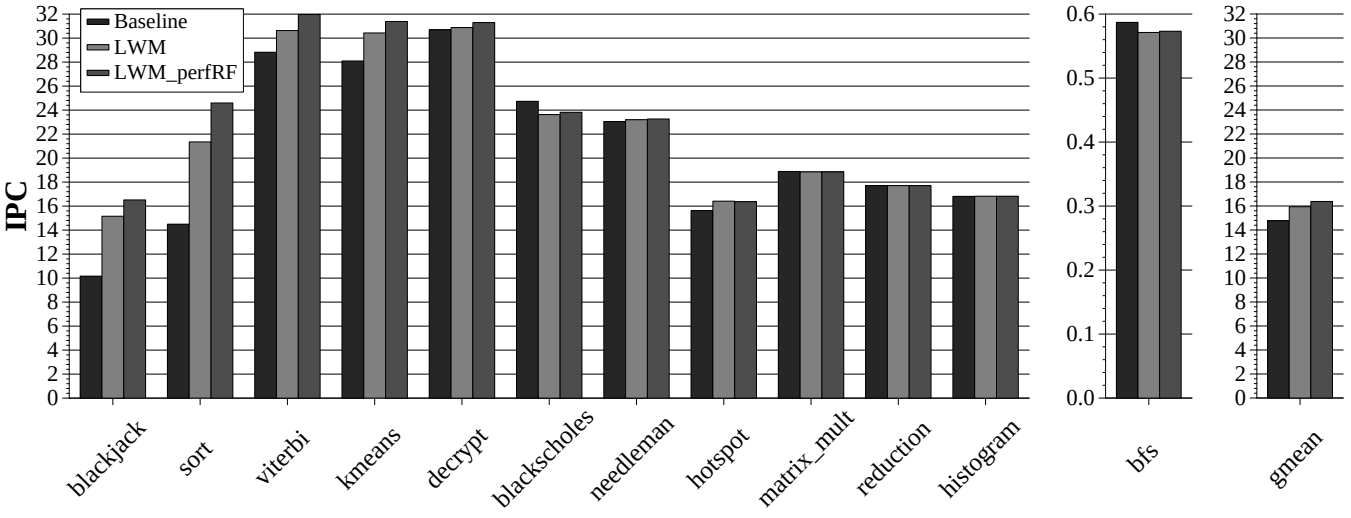


Figure 5.4: Effect of one thread per column sub-warping restriction

mask rather than allowing any arbitrary group of threads to be packed into a sub-warp. Figure 5.4 shows the performance of the baseline GPU core, the LWM with 256 threads per large warp, and also the LWM with 256 threads per warp with a perfect (but very complex) register file design (LWM_perfRF) which allows arbitrary grouping of active threads in a large warp into a sub-warp.

For the leftmost benchmarks (branch intensive applications), even though LWM improves performance significantly over the baseline GPU core, there is a noticeable performance difference between the LWM and the LWM with a perfect register file design. For *sort*, the flexibility of allowing arbitrary sub-warping improves performance by 15.2%. However, the hardware cost associated with arbitary sub-warping does not justify this gain. The LWM with the one thread per column sub-warping restriction comes within at least

90% of the performance of the perfect register file for each benchmark (even for branch intensive applications except for *sort*). I conclude the LWM with the one thread per column sub-warping restriction provides significant performance gains at a reasonable complexity level.

5.2.2.3 Effect of Large Warp Optimizations

Figure 5.5 shows the effect of the memory divergence and unconditional control flow optimizations presented in Chapter 4. The memory divergence optimization does not provide much benefit. The reason for this is that memory instructions for which consecutive threads access consecutive memory locations tend to be on control-independent code. Therefore, when a large warp reaches such an instruction, its active mask is fully populated with all active threads. As such, the dynamic sub-warping mechanism cannot combine threads from different rows in the active mask and therefore no additional memory divergence occurs regardless of whether this optimization is turned on or off. This mechanism would be effective when coalesced (i.e., consecutive) memory instructions appear on control-dependent code paths. However, I find this to be rare in the benchmarks used for this study.

The unconditional control flow optimization does slightly improve performance (by about 1%) by taking advantage of the fact that a large warp only needs a single PC update which is valid for all threads in the large warp.

This concludes the evaluation of the Large Warp Microarchitecture in isolation (i.e., no two-level scheduling). I now move on to evaluating two-level

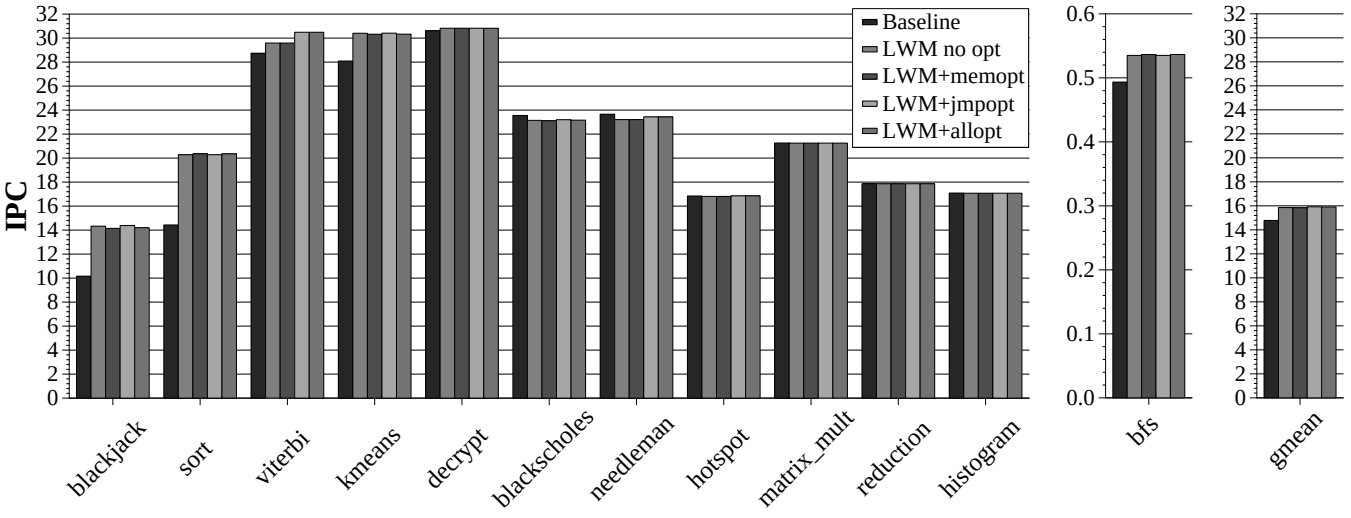


Figure 5.5: Effect of LWM optimizations

scheduling in isolation (i.e., without large warps).

5.2.3 Analysis of Two-level Scheduling

In this section, I apply two-level scheduling on top of the baseline microarchitecture (not LWM) and vary the fetch group size. Since there are 32 total warps concurrently executing on the baseline 7-stage GPU core, I use fetch group sizes of 8, 16 and 32 warps. In my notation, "2Lev8" stands for two-level scheduling with a fetch group size of 8 (i.e., 4 fetch groups each consisting of 8 warps). Figure 5.6 shows the IPC as I vary the fetch group size.

For the benchmarks on the left, there is no variation since these benchmarks have very few idle cycles even with the baseline round-robin policy. However, the rightmost benchmarks do illustrate that a fetch group size of

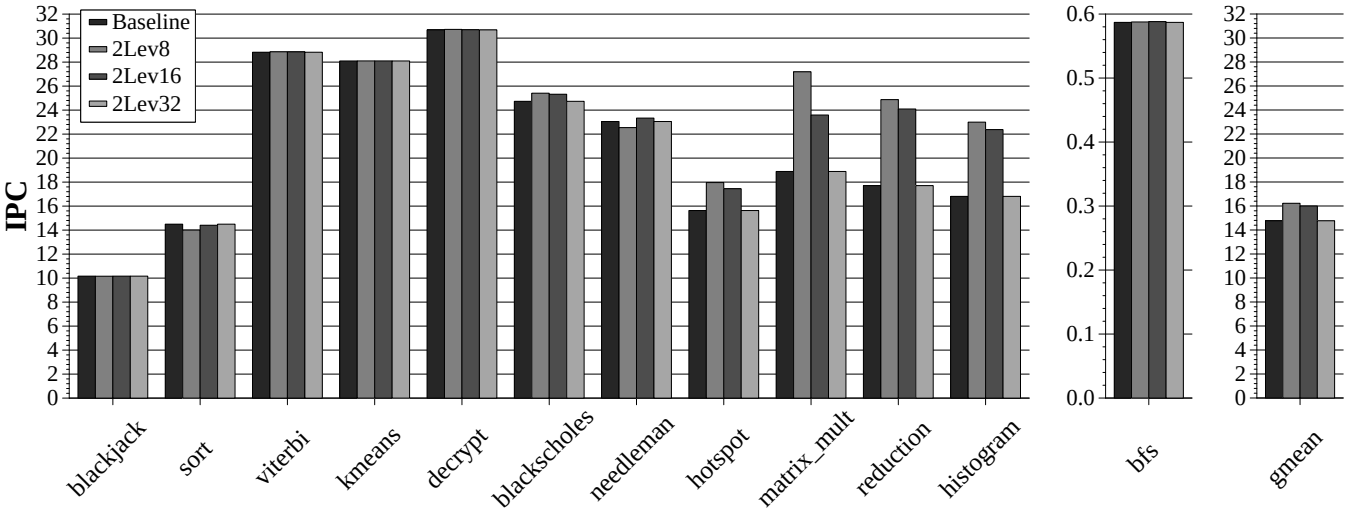


Figure 5.6: Effect of fetch group size on two-level scheduling

8 warps works best. Recall from Chapter 4 that the fetch group size should have just enough warps to keep the pipeline busy, but that too many warps in a fetch group limits the effectiveness of two-level scheduling. A fetch group size of 16 (half of all the warps) still improves performance over the baseline round-robin policy but not as much as 8. 16 warps is more than necessary to keep the pipeline busy and results in a larger subset of warps arriving at the long latency operation together and therefore is unable to hide latencies as well as 8 warps. Note that a fetch group size of 32 (i.e., all 32 warps in a single fetch group) is by definition equivalent to the baseline round-robin policy and therefore performs identically to it. Having only a single fetch group removes one of the levels of scheduling (the prioritization of the fetch groups) and therefore all the benefits of two-level scheduling are lost.

Chapter 6

Background and Related Work on Database Search on GPUs

Since a large part of this thesis focuses on accelerating one particular and very important not-so-regular parallel application, database search, in this chapter I present previous work related to executing database search on graphics processors. I also discuss in detail previously proposed state-of-the-art database search algorithms (for both full table scan and index-based search) designed to run on GPUs. Knowledge of these GPU search algorithms is necessary to understand the proposals presented in Chapter 7.

6.1 Related work on Database Search on GPUs

Rao et al. [43, 44] were the first to propose optimizing the data layout of databases for microarchitectural features such as cache size. Zhou et al. [58] proposed database search algorithms that made use of SIMD instructions on CPUs. Elements of these early proposals are found in the state-of-the-art GPU algorithms analyzed in this dissertation.

There has been plenty of recent work relating to accelerating database operations by porting database operations to GPUs. Bakkum et al. [6] imple-

ment a subset of SQLite commands (simple SELECT statements that translate to full table scans) that can run directly on a GPU. The main contribution of their work is an implementation of the SQLite command processor on a GPU. This enables database programmers to run their SQL queries on a GPU without having to re-write them using GPU specific languages such as CUDA [41] or other non-SQL libraries. Although some of the potential GPU architectural bottlenecks are discussed, there is no evaluation of any proposed changes to the GPU architecture.

He et al. [17, 18] and Fang et al. [12] also port database operations to a GPU and show impressive speedups over CPU execution. These works optimize the algorithms and software so that database operations run well on existing GPUs. Unlike my proposals, no changes to the GPU architecture are proposed but instead the algorithms and software are tailored for existing GPU architectures.

Wu, Haicheng et al. [56] propose Kernel-Weaver, a compiler framework that fuses multiple GPU kernels to 1) reduce redundant data transfers to and from CPU memory and GPU memory, and 2) enlarge the compiler optimization scope. They apply this to database applications and show impressive speedups of database operations running on real GPU hardware. In contrast, I show the potential improvement that can be realized if existing GPU architectures are enhanced with my proposals. Furthermore, their optimizations focus on minimizing overhead created by having multiple kernels for the multiple database queries involved in a single database transaction. On the other

hand, my proposals aim to speed up the individual kernels themselves.

Wu, Lisa et al. [57] propose a special hardware accelerator, called HARP, to speed up partitioning of large data sets. Like my proposal, this is one of the few works that speed up database operations with new hardware rather than optimizing software for existing hardware. However, my proposals directly target search, not partitioning, and therefore my work is orthogonal to theirs. Furthermore, I propose simple extensions to existing GPU architectures rather than designing a completely new specialized processing element.

Netezza [35], an IBM company, designed custom FPGAs called FAST (FPGA Accelerated Streaming Technology) Engines to use in combination with multi-core CPUs to speed up database queries. The custom hardware is designed to un-compress and filter out data as soon as it is read from disk thereby minimizing the amount of data that has to be stored in memory which alleviates the I/O bottleneck. In contrast, my proposals enhance existing GPU architectures rather than designing custom hardware, and assume all data is already stored in memory.

6.2 Database Search on GPUs: Algorithm Details

In this section I describe in detail the GPU search algorithms I use as the baseline source code for my proposals. These algorithms assume the entire data set is resident in the GPU's memory.

6.2.1 Special Instructions

Before describing the state-of-the-art GPU search algorithms, I must briefly describe the functionality of some special instructions used by these algorithms. GPU ISAs such as PTX from NVIDIA [42] have some special instructions that the search algorithms use to maximize performance. One such instruction is called *ballot*, which takes as input a single predicate bit from each thread in the warp, and concatenates them to produce a 32-bit value that is broadcast to all threads in the warp. Another special instruction is *popc* (i.e., population count) which counts the number of bits set (i.e., equal to 1) in a 32-bit value.

6.2.2 Full Table Scan on GPUs

Data in a database is organized in tables consisting of rows and columns. Each row corresponds to a different item or entry in the table, and each column is an attribute. Full table scans are essentially filter operations that examine an attribute (or multiple attributes) of every row in a table, and produce an output which consists of data from only those rows that pass a certain condition (i.e., the WHERE condition of a SQL SELECT statement).

In this section I describe the full table scan GPU algorithm which consists of three steps repeated in a loop: 1) load input data, 2) compute WHERE condition, and 3) conditionally write to output. For this algorithm, thousands of threads work together on a single full table scan by assigning different rows to each thread. The algorithm below is based on techniques

presented by Skjellum et al. [50] and Harris [16].

6.2.2.1 Load Input Data

In this first step, each thread within a warp reads input data from the same column but consecutive rows of the table. In order to make this a SIMD friendly coalesced load, the table is stored in column major order where data in the same column is stored consecutively in an array [1]. With this data organization, all loads are coalesced. Furthermore, row buffer locality is high as the thousands of threads stream through a column, resulting in efficient use of memory bandwidth.

6.2.2.2 Compute WHERE Condition

After each thread has read the input data, this data needs to be processed to see if the row passes the WHERE condition of the query. This step is also handled efficiently by the SIMD hardware on GPU cores since each thread performs the same calculation, but on different input data (which is what SIMD is all about). The result of this step is a single predicate bit per thread indicating whether or not the row passed the WHERE condition.

6.2.2.3 Conditionally Write Output

The third and final step of the loop is the problematic step for GPUs in that this step is not SIMD friendly like the first two. In this step, threads in a warp must conditionally write data to consecutive locations in a per-warp

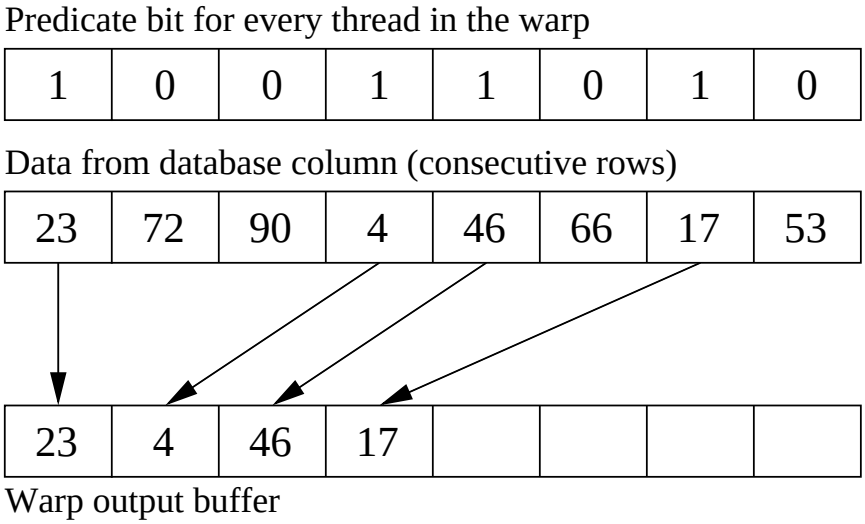


Figure 6.1: Step 3 of full table scan algorithm assuming SIMD width of 8

output buffer based on the predicates computed in the second step. Figure 6.1 graphically illustrates this process. For ease of illustration, the SIMD width (i.e., warp size) is assumed to be 8 in Figure 6.1.

Conditional execution is known to be problematic for GPUs [14, 15, 13, 33] since it results in underutilization of SIMD resources. To make matters worse, each thread that passed the WHERE condition (and therefore needs to write to the output buffer) must know how many threads before it also passed the WHERE condition so that it can write its data to the correct location in the output buffer. On current GPU architectures [38], communicating this information to each thread in a warp requires multiple instructions. The pseudo-code snippet below shows how this is accomplished on current GPUs.

```
bits = __ballot(predicate); // broadcast predicates
total = __popc(bits); // total passed
if(predicate){
  bits &= lane_mask; // cout only threads before me
  idx = __popc(bits); // compute unique output idx
  output[idx] = data; // write data to output
}
output += total; // update output pointer
```

Essentially, each warp executes a complicated sequence of special instructions to perform the relatively simple task illustrated in Figure 6.1. This is the process I speed up with my new *conditional accumulate* instruction presented in Chapter 7.

6.2.2.4 Combining Per-Warp Output Buffers

In the full table scan algorithm, each warp accumulates output data in a private per warp output buffer by continually repeating the three steps described in Sections 6.2.2.1 to 6.2.2.3. The per warp output buffer is stored in scratchpad memory which ensures that the potentially unaligned (but contiguous) writes illustrated in Figure 6.1 are handled efficiently. However, once this buffer is full, it must be flushed to an array in global memory that all warps write to. This is accomplished by each warp executing an atomic add operation to allocate space in the global output array before copying data from its private buffer to global memory. The atomic operation ensures that no two warps write to the same location in the global output array when flushing

data from their private buffer. Therefore no output data is lost or overwritten so the query executes correctly. The copy from scratchpad to global memory consists only of full cache line writes and therefore can be efficiently coalesced by the memory system and is not a major bottleneck.

6.2.3 Index-Based Search

Index-based search is essentially a tree traversal algorithm. To make efficient use of the SIMD hardware on GPUs, Kaldewey et al. propose that each warp performs a P-ary search [22] where P is equal to the SIMD width of the GPU core. To make the memory accesses of a warp during P-ary search SIMD friendly (i.e., threads in a warp access data in the same cache line), Kim et al. [26] propose organizing the index structure as a hierarchical tree with the elements rearranged based on architectural features such as SIMD width. I review both concepts below.

6.2.3.1 P-ary search

In traditional binary search, each node consists of only a single element (i.e., key). Each step of the search loads this key and performs a single comparison to decide which of the two possible keys to load next, thereby narrowing the search space by a factor of two every step. In contrast, in P-ary search, each node consists of P keys, and P simultaneous comparisons are performed to decide which of the P+1 nodes to load next thereby narrowing the search space by a factor of P+1 each step. P-ary search reduces the number

steps in the search to log_{P+1}(data set size) instead of log_2(data set size) steps for binary search. In addition, P-ary search makes efficient use of the SIMD hardware on GPUs since the threads of a warp can work together on a single P-ary search. Since GPU cores simultaneously execute multiple warps, multiple index-based searches can be processed simultaneously where each warp is searching for a different key. However, in order for the memory accesses of a warp during P-ary search to be SIMD friendly, the data in the search tree must be reorganized such that the P elements in a node are stored in an aligned contiguous region of memory (i.e., a single cache line).

6.2.3.2 Index-tree Data Layout

Figure 6.2, adapted from Kim et al. [26] illustrates the SIMD-friendly data layout of the index tree. For ease of illustration, Figure 6.2 represents an index tree with 1 million keys corresponding to 4 levels of a 31-ary search. The smaller shaded triangles represent cache-line sized nodes consisting of P=30 32-bit keys so that each step of the 31-ary search requires a single coalesced memory read from a warp.

In addition to organizing the tree with respect to SIMD width, the data is also rearranged according to other important microarchitectural features such as DRAM page/row buffer size. Although this second level of reorganization was originally proposed to better exploit TLB locality of the memory accesses created during a P-ary traversal [26], the same rationale applies to exploiting row buffer locality. The larger triangles in Figure 6.2, outlined by solid

lines, represent row buffer sized nodes. With this two-level reorganization of the index-tree, once a warp reads the first 30 keys of a row buffer node (shaded triangles at the top of the larger outlined triangles), after processing that data (i.e., performing 30 comparisons), a second memory request will be created a short time later which maps to the same row buffer as the first request. Since subsequent accesses to an already opened page/row (i.e., row buffer hits) are serviced much quicker (about one third the latency) than accesses to a different row (i.e., row conflicts), the search can be significantly sped up if this row buffer locality is exploited. However, I find that this row buffer locality is often destroyed by memory requests from other warps processing different index-based queries. The row buffer locality hint bits I propose in the next chapter specifically address this problem.

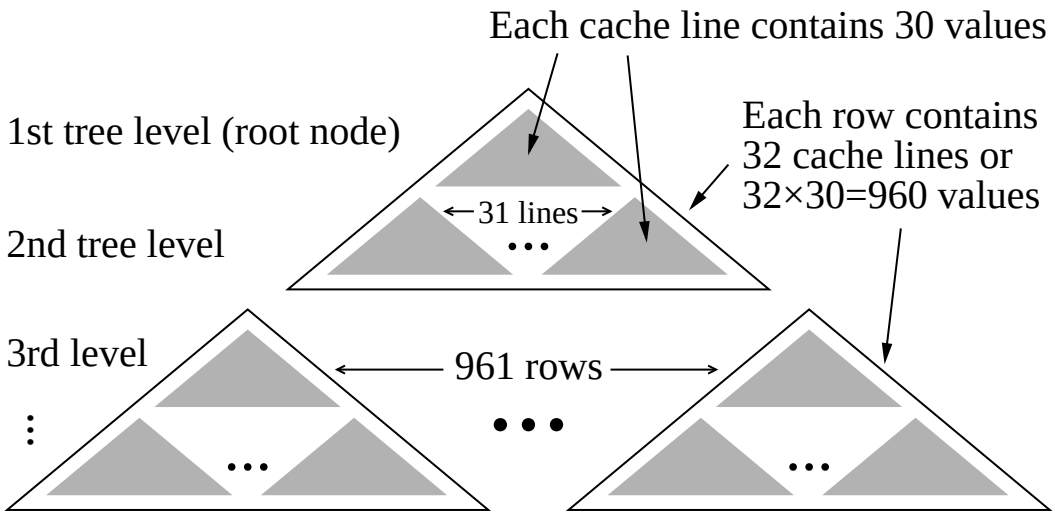


Figure 6.2: Index tree layout reorganized

6.2.3.3 Breadth First Node Organization

Unlike traditional trees that store pointers to child nodes along with the data, the index tree presented here does not. Instead, nodes are statically laid out in a breadth first fashion in memory starting with the root row buffer node. The 961 row buffer nodes in the next row buffer level are stored contiguously in memory directly after the root row buffer node. The ∼1 million row buffer nodes that would be present for a 1 billion key index tree come next. Furthermore, within a single row buffer node, the 32 cache lines are also laid out breadth first. This organization avoids child pointers and therefore saves space making it more likely for the index to fit entirely in memory. Furthermore, the tree can be traversed with simple shift+add instructions based on the 30 comparisons done in each step. The following pseudo-code snippet corresponds to a single step of a 31-ary search and is repeated based on the total depth of the index tree.

```
value = tree[idx]; // read 30 32-bit values
predicate = (search_key > value); // 30 comparisons
bits = __ballot(predicate); // broadcast results
skip = __popc(bits); // which path
idx += ((skip+1)*SIMD_width)); // compute next idx
```

Chapter 7

Improving Database Search on GPUs

In this section I describe my proposals to improve the performance of database search on GPUs which consists of new ISA instructions (and the hardware to execute them efficiently), and a row buffer locality hint bit for load instructions so that row buffer locality can be better exploited when processing multiple index-based queries simultaneously.

7.1 Conditional Accumulate Instruction

Recall from Section 6.2.2.3, that the 3rd step of the full table scan algorithm requires threads in a warp to conditionally write data to contiguous locations in an output buffer based on predicate values computed in the previous step. Current GPU architectures [38] require several complex instructions to accomplish this task (see code snippet in Section 6.2.2.3). To speed up this process, I propose a new *conditional accumulate* instruction that accomplishes this task in one instruction. GPU programmers can make use of this feature by calling the special conditional accumulate intrinsic function just as is already done for instructions like *ballot* and *popc*. This new instruction is defined as follows:

```
COND_ACC DestReg, BaseReg, PredicateReg, SourcReg
```

The new *conditional accumulate* instruction takes as input:

1. A base register which points to where in the output buffer the warp needs to start accumulating data. This pointer is the same for all threads in a warp. It can be a pointer to scratchpad memory (as done in the full table scan algorithm) or global memory.

2. A 1-bit predicate register. This register indicates whether or not the corresponding thread within the warp needs to write its data to the output.

3. A source register, unique to each thread, that contains the data that potentially needs to be written to the output.

The result of executing this instruction is:

1. A *contiguous* write of 0-32 data elements (since the warp size is 32) to memory (either scratchpad or global) starting at the address specified in the base register. Source registers from only those threads with a true input predicate bit will be written to memory. The order in which the data is written does not matter, but the write must be contiguous, and no data can be lost.

2. The number of data elements actually written to memory is counted and written back to a destination register specified by the instruction.

With this new instruction the following code snippet repeated from Section 6.2.2.3:

```
bits = __ballot(predicate); // broadcast predicates
total = __popc(bits); // total passed
if(predicate){
  bits &= lane_mask; // count only threads before me
  idx = __popc(bits); // compute unique output idx
  output[idx] = data; // write data to output
}
output += total; // update output pointer
```

is replaced with the following:

```
total = __cond_acc(output, predicate, data);
output += total; // update output pointer
```

This results in only a single instruction executed instead of six, and also avoids the conditional branch that would be generated by the routine from Section 6.2.2.3. If this instruction can be executed efficiently in the pipeline, there is strong potential for performance improvement for programs that make use of this new instruction. As will be shown in Chapter 8, this instruction results in a 21% increase in performance for full table scans that produce large outputs.

Figure 7.1 illustrates the hardware needed to execute this new instruction efficiently in the pipeline. For ease of illustration, the warp size is again assumed to be 8 threads, but the same combinational logic can easily be extended to 32. The hardware essentially performs a prefix sum or scan operation with multiple trees of adders that grow in size. The top row consists of 7 1-bit adders (31 for a warp size of 32) where each bit is added with the bit directly before it. The next set of adders are 2-bit adders that add each 2-bit value with the 2-bit value produced 2 to its left. Lastly, there is a row of 3-bit adders where each 3-bit value is added with the 3-bit value 4 to its left. This easily scales to larger warp sizes. With a warp size of 32, 5 levels of adders would be needed and the final row would contain 16 5-bit adders. There are algorithms that accomplish this such as the work-efficient algorithm proposed by Blelloch [7] that are more hardware efficient (fewer adders). However such algorithms result in more gate delays (double the levels of adders, but far fewer adders at each level). Since this new instruction is only summing 1-bit predicates, the hardware cost of the adders is not excessive and latency is more important so as to not increase the cycle time of the core.

After the prefix sum operation on the predicates is computed[1], the sums are left shifted by 0, 1, 2, or 3 bits depending on the size of the data that potentially needs to be written to memory (1, 2, 4 or 8 bytes), and then added to the base register to create the correct output addresses. Finally, data

[1]The hardware shown in Figure 7.1 actually calculates an inclusive prefix sum (all threads before it plus itself), but this instruction needs an exclusive prefix sum which is simply the inclusive sum shifted right.

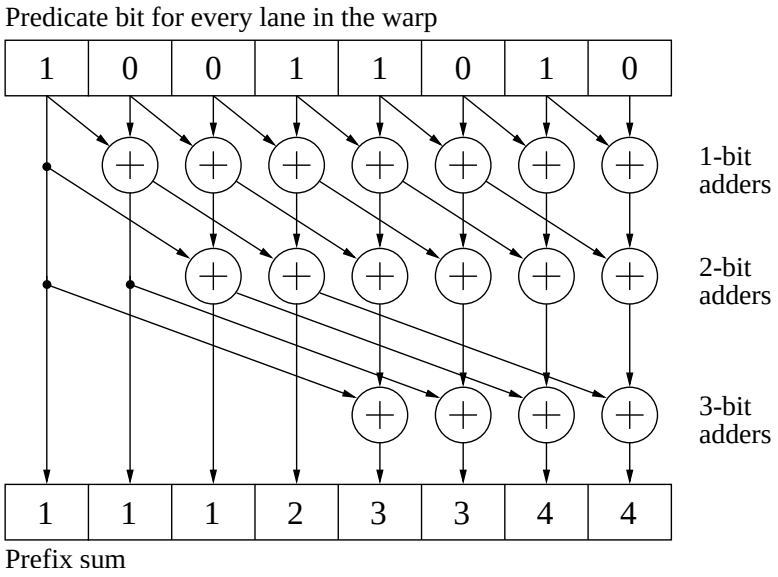


Figure 7.1: Hardware required for conditional accumulate

is stored in memory at the addresses calculated, guarded by the predicate bits so that those threads whose bit is false do not corrupt the data that is written by those threads whose predicate bit is true.

7.2 Row Buffer Locality Hint Bits

To speed up index-based searches by increasing row buffer locality when multiple index-based search queries are executed simultaneously by the GPU, I propose row buffer locality hint bits. Recall that memory is organized such that a single memory channel consists of multiple memory banks that can be accessed in parallel. Each bank contains a row buffer of several thousand bytes. Whenever a particular row buffer is first opened, subsequent accesses to that same row can be serviced quickly. Such accesses are called row hits.

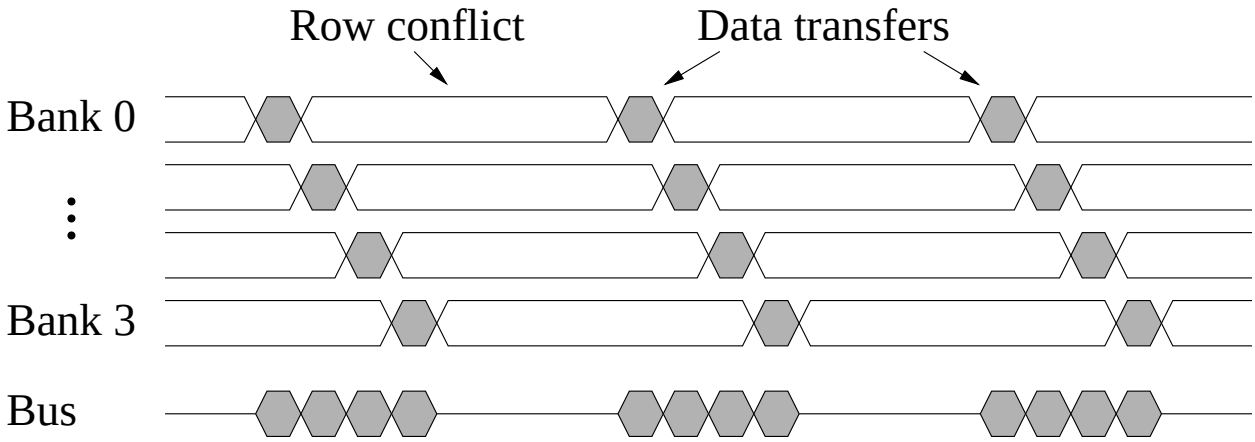
Row hits have a short latency, and can be pipelined with other accesses to the same row, and therefore result in efficient use of the large memory bandwidth available on GPUs. An access to a different row requires a much longer latency, and is known as a row conflict. Row conflict latency is about three times as long as row hit latency, and row conflicts cannot be pipelined. For this reason, existing memory controllers use a first-ready first-come-first-serve (FR-FCFS) scheduling policy [46, 45] which essentially prioritizes row hit requests over row conflicts. When many row hit requests are exposed to the memory system concurrently, FR-FCFS scheduling exploits this locality resulting in efficient use of memory bandwidth. However, if row buffer locality exists, but the requests are not concurrently exposed to the memory system due to a data dependency between them (which is the case for index-based search), even FR-FCFS will not capture this locality.

Recall from Section 6.2.3.2 and Figure 6.2, that due to the page level data blocking proposed by Kim et al. [26], there is some row buffer locality in the requests of a single index-based query. More specifically, the pattern of memory requests from a single tree traversal, if executed in isolation, will be a single row conflict followed by a single row hit. This row conflict then row hit pattern is repeated over and over until the traversal finishes. This means there is a potential for a row buffer hit rate of 50% if this locality can be exploited by the memory system. However, when multiple such queries are simultaneously executed on the GPU, this row buffer locality is destroyed due to conflicting requests from warps executing different queries (i.e., searching

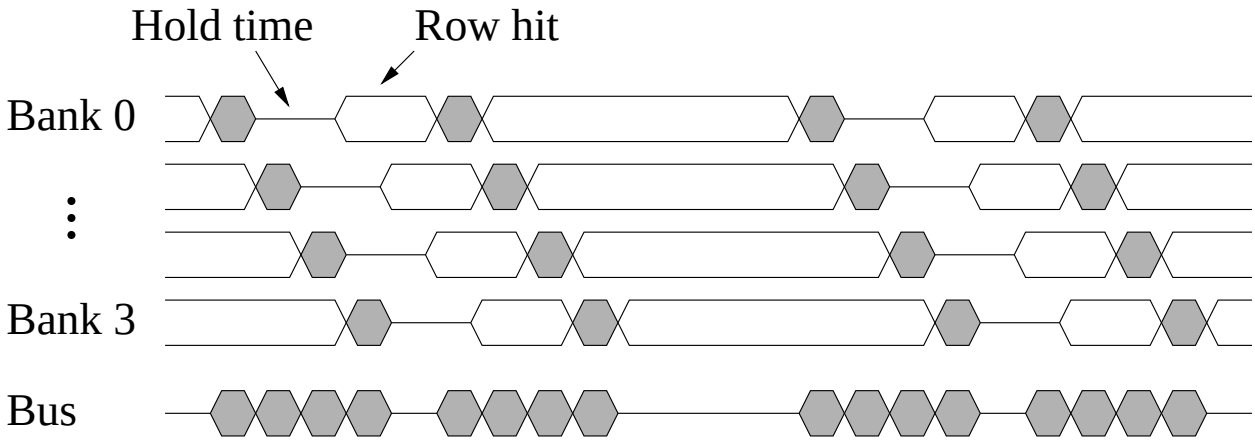
for different keys). The problem is that there is a data dependency between the request that opens up a row, and the next request to that same row which is generated soon after. During the time that data from the first request (which opened the row) is being processed by the GPU core, there exists no other memory request to that row in the memory system. Therefore, even an FR-FCFS scheduler will schedule a conflicting request and therefore close the row. Soon after, the request that would have been a row buffer hit is exposed to the memory system, but the opportunity to exploit row buffer locality has already been lost. With many queries being processed simultaneously, the likelihood of this happening is very high and therefore little to no row buffer locality can be exploited.

This problem can be alleviated by my new proposal which adds a row buffer locality hint bit to existing load instructions. Since the GPU programmer knows the details behind the traversal algorithm and the data access patterns, he/she can insert pragmas before memory accesses in the program indicating that another request to the same row will be generated soon. If this hint bit is communicated with the load instruction to the memory system, the memory controller can use this information to *hold* the row buffer open and not schedule a conflicting request, knowing that another request to this same row will be generated very soon. In this way, the row buffer locality present in index-based tree traversal can be exploited.

Figure 7.2 illustrates the improvement in bus utilization that row buffer hints can provide. For ease of illustration, this figure represents a single channel



(a) No hints - all row conflicts



(b) With hints - some row hits

Figure 7.2: Comparison of bus bandwidth with and without row buffer hint bits

with only 4 banks. Figure 7.2(a) on top shows what happens without any row buffer hints. In this case, all requests are row conflicts since immediately after a request is serviced, the row is closed and another conflicting request (from a different warp) is serviced. Due to the long latency of row conflicts, the bus is underutilized. On the other hand, Figure 7.2(b) on the bottom shows what happens when appropriate hint bits are supplied by the programmer. The memory controller takes advantage of the row buffer locality hint bit and holds rows open (for only those requests whose hint bit is set) instead of immediately scheduling a conflicting request. As such, row buffer locality is exploited as seen by the significantly shorter row hit latencies shown in Figure 7.2(b). This improves the achievable bandwidth and therefore increases bus utilization which leads to significant improvement in index-based query performance.

When using row buffer hint bits to hold rows open, an important question is how long to actually hold the row open before allowing a conflicting request to be scheduled in case the anticipated row buffer hit does not arrive. Holding the row open for too long can degrade performance especially if the programmer is not accurate with his/her hints. Figure 7.3 explains how I set this threshold. If the anticipated subsequent row hit arrives quickly, there is no problem and bus utilization is increased as shown by the "beneficial case" in Figure 7.3. However, to set the threshold, I consider how late the anticipated request must arrive so that I "break even" had the row not been held open and instead a conflicting request was immediately scheduled. The

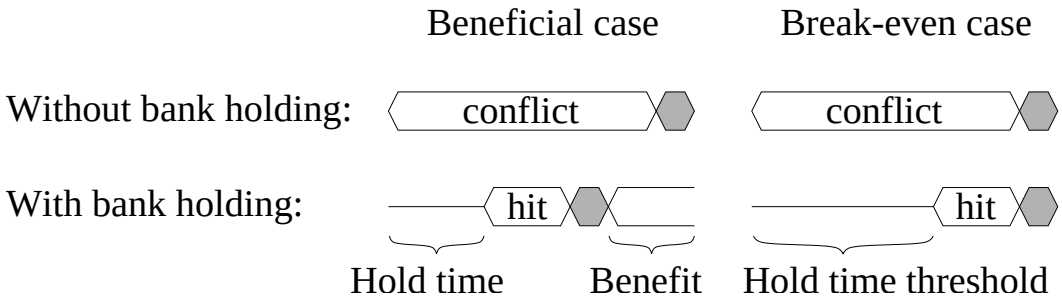


Figure 7.3: How to set the row buffer hold threshold

"break even" case of Figure 7.3 shows that this threshold is equal to the row conflict latency minus the row hit latency, i.e., if the anticipated row buffer hit arrives within (row_conflict_latency - row_hit_latency) cycles, bus utilization increases. If it arrives later, it would have been better to have immediately scheduled a conflicting request. Therefore I set the hold time threshold to (row_conflict_latency - row_hit_latency) cycles.

7.2.1 Tree Traverse Instruction

The benefit of holding a row open for an anticipated subsequent request to that same row depends on how quickly that next request is generated. The sooner the subsequent request is exposed to the memory system, the less time the corresponding bank must be held idle, which further improves bus utilization and therefore increases query throughput. To this end, I propose a new *tree traverse* instruction which results in the anticipated next row buffer hit to be exposed to the memory system sooner.

Recall from Section 6.2.3.3 that the sequence of instructions required to

generate the next request during tree traversal is very similar to the computation required to support the new *conditional accumulate* instruction proposed to speed up full table scans. The difference is that this instruction requires just a sum of the predicate values instead of an entire prefix sum. Just as in *conditional accumulate*, this sum is shifted (but by the node size instead of the data element size) and added to a base register to compute the address of the next node in the traversal. No data has to be written to memory, instead the newly computed address is immediately used to load the values in the next node of the tree traversal. Therefore I can leverage the hardware added to support *conditional accumulate* to also support *tree traverse*.

This new instruction is defined as follows:

```
TREE_TRAVERSE DestReg, BaseReg, PredicateReg, SourceReg
```

The new *tree traverse* instruction takes as input:

1. A base register which contains the address (i.e., array index) of the left-most child of the current node.

2. A 1-bit predicate register. This bit indicates if the search key is greater than the corresponding value in the current node (i.e., the result of the P-1 comparisons in P-ary search). The predicate bits of each thread in the warp are summed to compute the address of the next node.

3. A source register which contains the node size (must be a power of 2). This value indicates how much to left shift the sum of the predicates.

The shifted value is added to the base register to produce the address of the next node.

The result of executing this instruction is:

1. DestReg will contain the newly computed address (i.e., array index) of the next node in the tree traversal.

With this new instruction, the anticipated row buffer hit is generated sooner (since fewer instructions need to be executed to create the next request) which reduces the average hold time. This in turn improves the benefits of row buffer hints applied to index-based tree search.

Chapter 8

Evaluation of Database Search on GPUs

In this chapter I present the results of my proposals to improve database search performance on GPUs. I first describe my evaluation methodology, then present full table scan results, followed by index-based search results.

8.1 Evaluation Methodology

I use a cycle-level simulator that simulates parallel x86 threads programmed in a GPU style where each thread executes the same compute kernel. In the results presented in this chapter, I simulate a single GPU core (i.e., one Streaming Multiprocessor) that concurrently executes 1024 threads (32 warps, each with 32 threads). The core has access to a single DDR memory channel with 8 banks. Table 8.1 presents the system parameters used in the simulations for the baseline GPU core and memory system. It is essentially identical to the parameters presented in Chapter 5 Table 5.1 and is repeated here for convenience.

I augmented the x86 ISA to include instructions needed for GPU style processing and also to support the special intrinsic instructions (e.g., ballot, popc) current GPU ISAs [42] offer. The intrinsic instructions are simulated

Scalar front end	1-wide fetch, decode stages, round-robin warp scheduling 4KB single-ported I-Cache
SIMD back end	In order, 5 stages, 32 parallel SIMD lanes
Register file and on-chip memories	64KB register file 32KB, 4-way, 1-cycle D-Cache, 1 read/write port, 128-byte line size 1MB L2 Cache 32KB, 32-banked scratchpad memory (1KB per warp)
Memory system	Open-row, FR-FCFS scheduling policy, 8 banks, 4KB row buffer 100-cycle row-hit, 300-cycle row-conflict, 32 GB/s memory BW

Table 8.1: Baseline GPU core and memory configuration

with function calls that compile to a single instruction in the binary. I also use the same procedure to simulate the effect of the new instructions I proposed in this work.

Furthermore, since x86 does not have instructions to aid with branch divergence/re-convergence of parallel threads like GPU ISAs do [42], I created instrumentation tools to identify conditional branch instructions and their control flow merge points to correctly handle conditional branches present in full table scan.

For input data to full table scan, I created column data consisting of 4 million random 32-bit integers similar to the methodology used by [58]. The scan performs a search that finds all values less than a given maximum value. The equivalent SQL query is:

```
SELECT value FROM table WHERE value < max_value;
```

By varying max_value, I change the selectivity of the query (i.e., how

many rows pass the WHERE condition). In the full table scan results presented in the next section, I vary the selectivity of queries from 50%, down to less than 0.1% (1 out of 1024 rows pass the WHERE condition).

For index-based search, the index consists of 1 billion randomly generated unique 32-bit keys similar to the methodology in [26]. These 1 billion keys correspond to a unique attribute (i.e., primary key column) of a table with 1 billion rows. The keys to lookup are also randomly generated but I ensure that most of them actually exist in the data set (only 1 out of 1024 lookups are not found). Each warp processes a different search key by traversing the index tree, and returns the row ID of where the key exists in the original 1 billion row table. If the key was not found, an invalid (i.e., out of range) row ID is returned. For my evaluation, I process 128K queries over the 1 billion key data set. To measure performance, I compute query throughput by dividing the 128K queries by the time it took to execute all of them (total cycle count).

8.2 Results

In this section I present the results of my proposals to improve database search performance on GPUs. I first present results for full table scan and then index-based tree search.

8.2.1 Full Table Scan Results

Figure 8.1(a) shows the speedup obtained from the new *conditional accumulate* instruction I proposed in Section 7.1. For this experiment, the

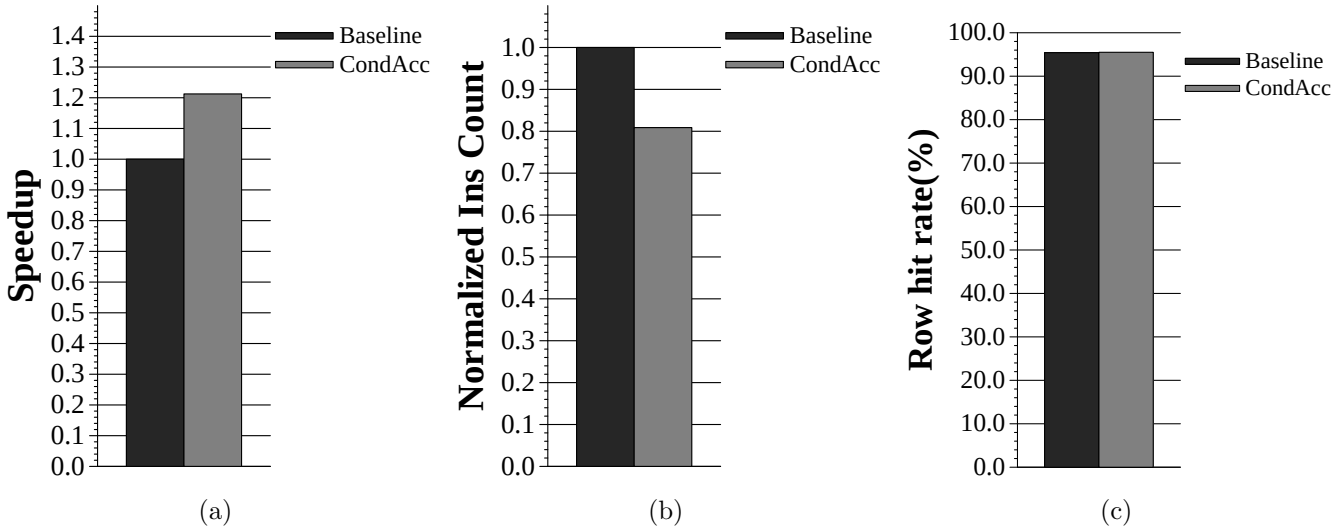


Figure 8.1: Full table scan results, selectivity = 50%

selectivity of the query was 50% so the output size is large. To explain the source of performance gain, Figures 8.1(b) and 8.1(c) compare the *warp* instruction count and row buffer hit rate of full table scan with and without the new *conditional accumulate* instruction. Warp instruction count is slightly different from regular instruction count. When a warp retires an instruction, if all threads in the warp are active, the regular instruction count would be incremented by the warp size (e.g., 32). If some threads are inactive (due to branch divergence), the regular instruction count would be incremented by the number of active threads in the warp (e.g., 1-31). On the other hand, warp instruction count is always incremented by one regardless of whether or not all threads in the warp are active. Warp instruction count is a better indicator of the expected performance improvement of the new conditional accumulate instruction since comparing regular instruction count does not properly take

into account the penalty of executing conditional code. For example, when a warp executes an if statement, it does not matter if just one thread needs to execute it, or 31 threads need to, the performance would be the same regardless of the way the warp diverged. However, regular instruction count treats these situations differently whereas warp instruction count does not. As shown in Figure 8.1(b), the 20% reduction in warp instruction count correlates well with the 21% speedup. Figure 8.1(c) shows that row buffer locality is very high for this workload (95%), which is expected since the memory access patterns are streaming. Therefore, memory is not a major bottleneck, and the reduction in warp instruction count directly translates to speedup.

Figure 8.2 shows full table scan results as the selectivity of the query is varied from 1 out of every 2 rows (50%) to 1 out of every 1024 rows (less than 0.1%) passing the WHERE condition. As the selectivity is decreased, the probability that all 32 threads in the warp fail the WHERE condition increases. When all 32 threads fail, step 3 of the full table scan algorithm (which is what my new instruction speeds up) is skipped since nothing needs to be written to the output. Therefore, I expect the performance improvement for full table scan to decrease as selectivity decreases. Figure 8.2(a) shows exactly this behavior. However, notice that the performance improvement does not change much as selectivity is decreased from 1 out of 2 (50%) to 1 out of 8 (12.5%). This is because at a selectivity of 12.5%, the probability that all 32 threads in the warp fail the WHERE condition is still very low ($\sim$1%). Even if just a few threads in the warp pass the WHERE condition, the

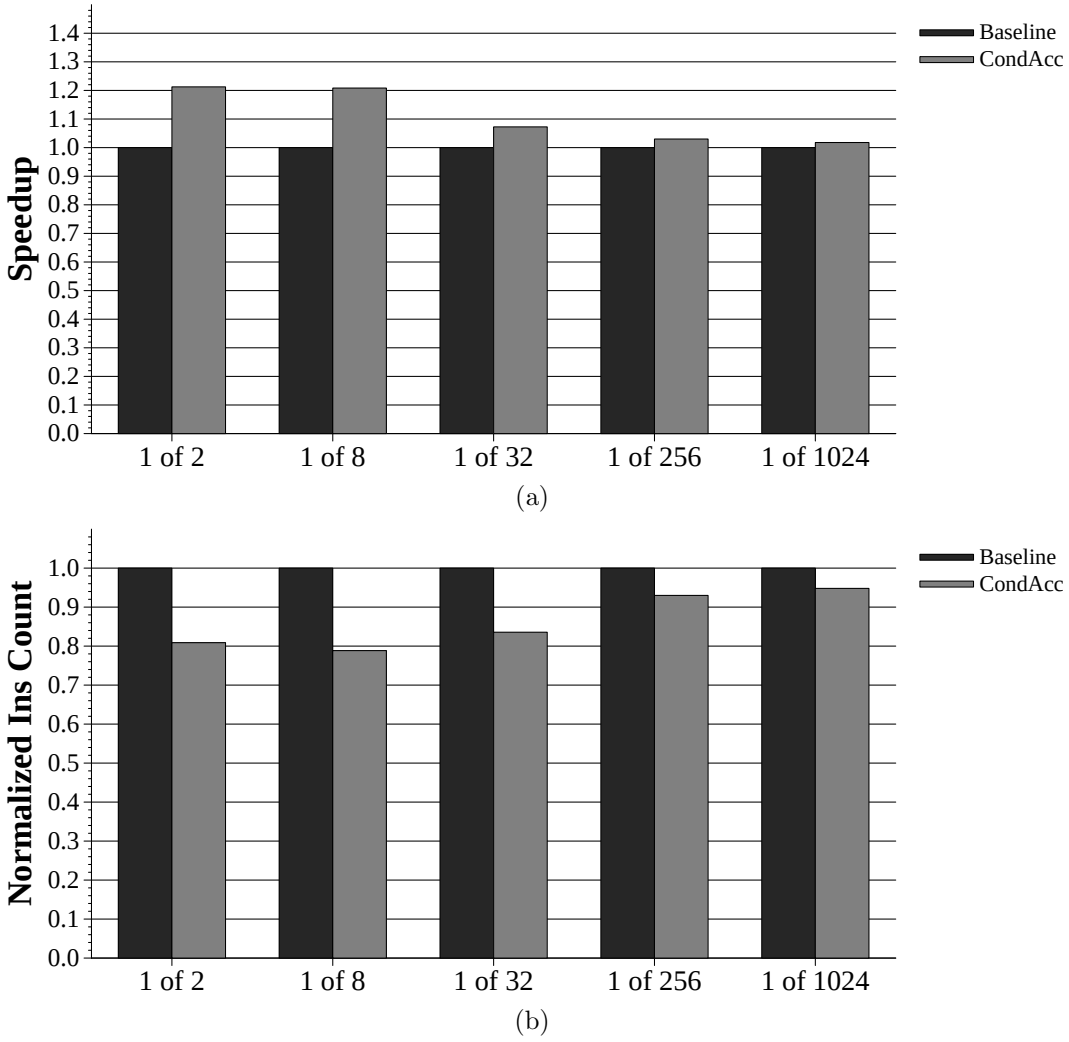


Figure 8.2: Full table scan results - varying the selectivity

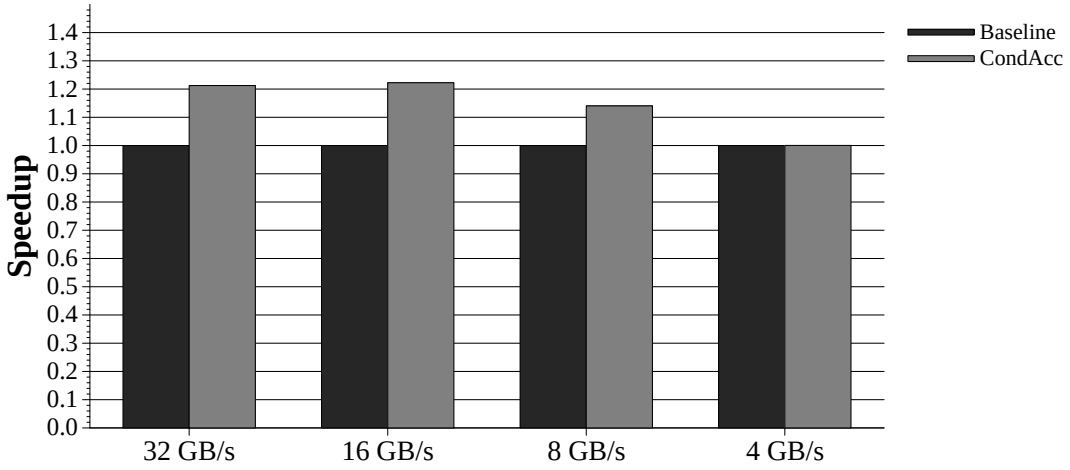


Figure 8.3: Full table scan results - sensitivity to bandwidth, selectivity = 50%

penalty (in warp instructions) is the same as the case where half of the threads pass the WHERE condition. The *conditional accumulate* instruction therefore improves performance for all but the most selective queries. The reduction in warp instruction count shown in Figure 8.2(b) indicates that the difference in warp instruction count converges to zero as selectivity decreases. This explains why performance improves for all except the most selective queries.

Although conditional accumulate reduces the warp instruction count, this only translates to performance improvement if the workload is compute bound. However, in bandwidth limited systems, or for queries with complicated WHERE clauses (involving data from several columns), full table scan may become memory bound. To this end, Figure 8.3 shows full table scan results as the bandwidth of the memory system is decreased from 32 GB/s

(gigabytes per second) to 4 GB/s.[1] Significant performance improvement is still achieved in all except the most bandwidth limited configuration. With only 4 GB/s of memory bandwidth, the full table scan workload becomes memory bound and therefore conditional accumulate becomes inn-effective as expected. I conclude that conditional accumulate significantly improves full table scan performance for all except the most bandwidth constrained environments.

8.2.2 Index-based Search Results

Figure 8.4(a) shows the improvement in index-based query throughput. To explain the source of performance gain, Figures 8.4(b) and 8.4(c) show the improvement in row buffer hit rate and bus utilization. Recall from Section 8.1 that for this experiment, a total of 128K index-based queries are processed in order and that 32 queries are simultaneously executed by the 32 concurrently running warps on a single GPU core. As soon as a warp finishes one lookup, it immediately starts another one until all 128K lookups are done.

The four bars on each graph in Figure 8.4 represent from left to right 1) the baseline tree traversal algorithm described in Section 6.2.3, 2) the baseline algorithm with the new *tree traverse* instruction presented in Section 7.2.1, 3) the baseline algorithm with appropriate row buffer locality hint pragmas

[1]Reducing the memory bandwidth is analogous to increasing the bandwidth demand (number of columns accessed) of full table scan. Therefore, decreasing bandwidth by a factor of 8 (down to 4 GB/s) approximates performance when the number of columns accessed increases from 1 to 8.

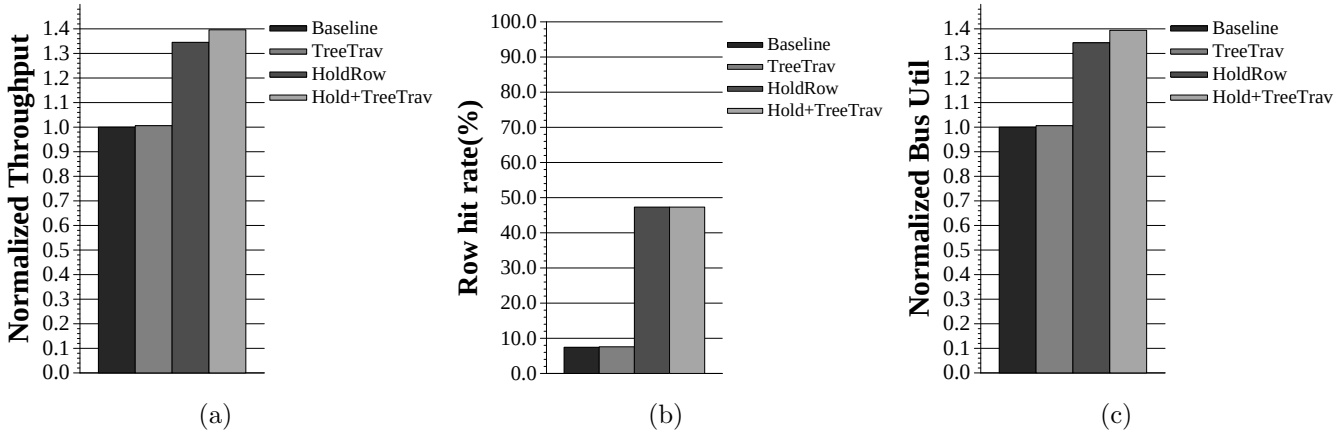


Figure 8.4: Index-based search results, Tree size = 1 billion 32-bit keys

supplied by the programmer, and 4) the combination of row buffer hints and the new *tree traverse* instruction. As can be seen from Figure 8.4(a), the *tree traverse* instruction applied in isolation does not have much effect on performance. This is because index-based search of a 1 billion key data set is more memory bound than compute bound. The row buffer hint bits, on the other hand, improve performance significantly. This can be explained by the dramatic increase in row buffer hit rate as shown in Figure 8.4(b) (from below 10% in the baseline to almost 50%).Recall from Section 7.2 that the multi-level reorganization of the index tree [26] allows for a 50% potential row buffer hit rate when the requests generated from a single query are processed in isolation. This locality is almost completely lost in the baseline algorithm due to DRAM row buffer contention from requests from different warps executing other queries. However, the row buffer hint bits provide the memory controller with the additional information to hold rows open until the anticipated next

row buffer hit arrives. Therefore the 50% potential row buffer hit rate is realized. This leads to more efficient use of the bus and therefore overall query throughput is increased by 34%.

The *tree traverse* instruction, when applied with row buffer hints, improves query throughput by 39% over the baseline. Recall from Section 7.2.1, that the benefit of holding a row open depends on how quickly the next request to that same row can be generated. *Tree traverse* exposes the anticipated row buffer hit to the memory system sooner. Therefore the corresponding bank that was holding that row open can service the row hit even sooner. This reduces hold time, and therefore improves bus utilization resulting in improved index-based query throughput.

Chapter 9

Conclusion and Future Research Directions

9.1 Conclusion

Graphics Processing Units (GPUs) have been shown to be a very promising architecture for executing general purpose (i.e., non-graphics) parallel applications. For an application to experience significant speedup from GPU execution, it must be able to be parallelized into many (e.g., thousands) threads each performing the same fundamental task on different data (i.e., data parallel). GPU programming paradigms such as CUDA (Compute Unified Device Architecture) [41] and OpenCL [25] allow programmers to express such parallelism by creating thousands of parallel threads each of which is set to execute the same piece of static code known as a kernel. Previous work [49, 19] has shown that *some* applications experience an order of magnitude speedup when executed on a GPU instead of a CPU. In addition to having lots of data parallelism, the applications that benefit most from GPU execution tend to have certain characteristics such as high computational intensity, regular control flow behavior and memory access patterns, and little to no fine grain communication among threads. However, *not all parallel applications have these characteristics*.

Parallel applications with a more balanced compute to memory ratio, divergent control flow behavior, irregular memory access patterns, and/or frequent fine grain communication among threads will not make full use of the GPU, resulting in performance far short of what could be delivered. I call such applications *not-so-regular* parallel applications. In this dissertation, I have proposed enhancements to the GPU architecture to improve the performance of not-so-regular parallel applications when executed on a GPU.

This was accomplished in two parts. First, I analyzed a diverse set of data parallel applications that suffer from divergent control flow and/or significant stall time due to memory. I proposed two microarchitectural enhancements to the GPU called the *Large Warp Microarchitecture* and *Two-Level Warp Scheduling* to address these problems respectively. While existing GPU cores concurrently execute multiple SIMD-width sized warps, the Large Warp Microarchitecture forms fewer but correspondingly larger warps and dynamically creates efficiently 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. To improve long latency tolerance for applications with a more balanced or low compute to memory ratio, I proposed a new two-level round-robin warp instruction fetch scheduling policy. This policy prevents all warps from arriving at the same long latency operation at the same time, thereby better overlapping computation with memory latency which reduces idle execution time. My experimental evaluations show that each mechanism significantly improves performance. Combined, both tech-

niques improve performance by 19% on average for a wide variety of general purpose parallel applications.

Second, I examined one of the most important and fundamental applications in computing: *database search*. Database search is an excellent example of an important application that is rich in parallelism, but rife with several of the not-so-regular parallel application characteristics. I proposed enhancements to the GPU architecture including new instructions that improve intra-warp thread communication and decision making, and also a row-buffer locality hint bit to better handle the irregular memory access patterns of index-based tree search. These proposals lead to a 21% performance increase for full table scans, and a 39% improvement in index-based query throughput.

The result of this dissertation is an enhanced GPU architecture that better handles not-so-regular parallel applications. This increases the scope of applications that run efficiently on the GPU, making it a more viable platform not only for current parallel workloads such as databases, but also for future and emerging parallel applications.

9.2 Future Research Directions

9.2.1 Future Research Directions Related to Warp Size

The Large Warp Microarchitecture improves performance for applications that suffer from significant branch divergence by grouping many (e.g., 256) threads into a single large warp, and dynamically forming SIMD-width sized sub-warps (e.g., 32 threads) from the active threads in the large warp.

However, as shown in Chapter 5, not all applications benefit from large warps. Applications with little or no branch divergence do not benefit and can even slightly degrade in performance with large warps. Furthermore, a fixed large warp size is not best for all applications. As shown in Chapter 5, some applications benefit from even larger warp sizes (e.g., 512 threads) whereas others are better off with smaller large warps sizes, or even SIMD-width sized warps.

In addition, the Large Warp Microarchitecture does not apply to GPU programs that use the concept of warp-level programming. Some advanced GPU programmers implement algorithms with the knowledge that threads are grouped into fixed size warps of 32 threads which are executed in lockstep. For example, the state-of-the-art database search kernels presented in Chapter 6 use warp-level programming. The special instructions used in those algorithms (i.e., the ballot instruction) assume fixed sized warps of 32 threads and may not execute correctly on architectures with different warp sizes.

This motivates the need for a microarchitecture that supports variable warp sizes. That is the programmer can specify through pragmas whether or not the microarchitecture is allowed to create warps larger than the default size. A microarchitecture that supports hybrid warp sizes could choose the best warp size per application resulting in overall better performance. It would also support high performing custom applications that use warp-level programming constructs. Therefore, designing such a microarchitecture that supports variable warp sizes is part of my future research directions.

Another extension to large warps I am interested in is scalarization [10,

5]. Recall that the unconditional branch optimization presented in Chapter 4 is a specific instance of a more general optimization that applies to any instruction for which the outcome is identical for all threads in a large warp (or even all threads in a SIMD-width sized warp). Take for example the loop control variable in a loop that all threads iterate the same fixed number of times. Every iteration, each thread reads the value of this variable, increments it, and writes it back to the register file before comparing it with a sentinel value to check whether or not to exit the loop. Existing GPU architectures unnecessarily waste resources by reading 32 values (one for each thread in the warp) from the register file, performing 32 additions to increment it, and then writing 32 values back to the register file even though these values and calculations are identical for all threads. As suggested by previous work [5], if instead each warp had a warp global register file in addition to per-thread register files, such scalar values could be stored only once per warp in the global register file instead of being repeated in each per-thread register file. Similarly, redundant computation would also be eliminated thereby saving energy.

The concept of scalarization applies to existing GPU architectures but has special implications for the Large Warp Microarchitecture. Since scalar values and computations are amortized over all threads in a warp, with large warps, the number of register reads/writes and computations is reduced even more since they are amortized over a larger number of threads. This not only saves additional energy, but can also improve performance since the scalar computations can be done in a single cycle for all threads (e.g., 256) in a large

warp. Furthermore, since there are fewer (but larger) warps in the Large Warp Microarchitecture, the overhead of the per-warp global register files would be reduced. Therefore, combining scalarization with Large Warps is a promising future research direction.

9.2.2 Future Research Directions Related to Warp Scheduling

Warp scheduling will continue to be an important design issue for GPUs. Two-level scheduling is a promising idea that improves performance by better overlapping memory latency and computation for applications with a more balanced or low compute to memory ratio. However, even within two-level scheduling, there is still much to explore. Given a two-level scheduling policy, each level of scheduling is independent. I used round-robin scheduling for both levels, but subsequent work [47] has shown that maintaining two levels of scheduling with different individual scheduling policies at each level (not round-robin) provides more benefit on average. An adaptive scheduling policy that learns the most effective scheduling policy at each level for each application (or different phases of the same application) would be even better.

Another aspect of warp scheduling is load imbalance. When the many warps concurrently executing on a GPU core synchronize at a barrier, those warps which reach the barrier last can be a significant bottleneck since the other warps have to stall and wait until these late arriving warps reach the barrier. The warp scheduling policy employed on the GPU core can significantly affect the amount of imbalance. A programmer driven progress metric such

as a special progress() function call could help alleviate load imbalance. The microarchitecture could be enhanced with a special progress register per warp that is incremented when the progress() function is executed. This progress register could be used to make better warp scheduling decisions to reduce the imbalance in applications that have barrier synchronization by prioritizing warps that have made less progress. This would reduce imbalance by making warps arrive at the barrier at roughly the same time.

9.2.3 Future Research Directions on Databases and GPUs

Database search is a vast and interesting field in and of itself that has been researched for decades. The work related to databases in this dissertation has focused on two types of database search: full table scans and index-based tree search. Although these search algorithms encapsulate a great deal of what databases are all about, there are a number of other database operations whose suitability for execution on GPUs is another future research direction.

First, aggregate queries (i.e., SUM, AVERAGE, MIN, MAX, etc.) are common especially in the field of business analytics. In general, aggregate queries scan a large amount of data (similar to the full table scans studied in this dissertation) but produce only a single output. Therefore, unlike the full table scans I analyzed, these types of queries do not produce large outputs, nor do they require the frequent fine grain thread communication needed to store output data contiguously. Instead, threads are largely independent and calculate their own local aggregate values before synchronizing at a global bar-

rier, after which these per-thread aggregate values are reduced to one value. As such, these queries may have load imbalance when calculating their private aggregate values resulting in inefficiency when executed on a GPU. The progress metric based scheduling idea discussed in the previous section may be particularly useful for speeding up aggregate queries run on GPUs.

Joins are another important relational database operation. Joins involve searching and returning data from more than one table. In fact, a join operation can be conceptually viewed as simply a full table scan on a single very large table created from taking the cross product of the rows from two existing tables. As such, the conditional accumulate instruction I proposed in Chapter 7 could also be useful for speeding up brute force joins on un-indexed attributes. However, typical join operations deal with indexed attributes and therefore consist of multiple index-based searches (i.e, key value lookups). My proposals to speed up index-based tree search should also apply to most join operations. However, other join algorithms, such as hash-based joins, have also been proposed. In the future, I want to study the efficiency of alternative join algorithms, such as hash joins, when executed on a GPU.

In this dissertation, I have only considered read-only database operations. However, updates and inserts are also important database operations. Most updates, however, first involve a search (i.e., lookup) on an indexed attribute (e.g., a unique ID number like a social security number) before an un-indexed attribute (e.g., a person's address) can be updated. Therefore, improving index-based search also improves the performance of typical updates.

However, updates on an indexed attribute or the insertion of a new row in a table (e.g., a new customer) requires the index structure itself to be updated. This can be especially problematic for custom index-tree layouts such as the one designed for GPUs described in Chapter 6. Therefore, designing methods to support fast and efficient updates to index structures designed for GPUs is another important research direction.

9.2.4 Future Research Directions on Memory Divergence

One not-so-regular characteristic of parallel applications that is especially problematic for GPUs that I have not specifically addressed is memory divergence. The memory divergence optimization for the Large Warp Microarchitecture presented in Chapter 4 prevents additional memory divergence that would not exist with regular sized warps from happening. However, it does not prevent memory divergence that exists in the baseline architecture. Furthermore, although the memory access patterns in index-based tree search are highly irregular, have poor row-buffer locality, and are unpredictable, they are not divergent given the index tree layout and search algorithm presented in Chapter 6. However, some parallel applications do suffer from memory divergence when threads in the same warp wish to access different regions of memory (i.e., different cache lines). When this happens, the entire warp stalls until all of its independent memory accesses complete. If multiple warps are stalled on divergent accesses to memory, there is an opportunity to reduce the average stall time caused by divergent memory accesses by changing

the memory scheduling policy. For example, interleaving the individual memory accesses from each warp would cause all warps to stall for a very long time. However, prioritizing memory requests from one of the warps, e.g., the one with the fewest outstanding misses, could significantly reduce the average stall time. Instead of all warps stalling until the last request from each warp is serviced, some of the warps would have all their requests serviced earlier. This would reduce average stall time and therefore potentially improve performance for applications with memory divergence. Such a warp-stalled aware memory scheduling policy is another promising future research direction.

Appendix

Appendix 1

Source Code for Database Search Algorithms

1.1 Full Table Scan Kernel

```
void * full_table_scan(void * thread_id)
{
  int my_max_value = get_max_value();
  unsigned tid = (unsigned long long)thread_id;
  unsigned k = tid;
  unsigned warp_id = (tid >> 5);
  unsigned lane_id = (tid & 0x1F);
  int my_mask = (1<<lane_id) - 1;
  unsigned * warp_buffer = &(smem[warp_id*256]);
  unsigned warp_num_found = 0;

  do{
    bool cond = (input_column[k] < my_max_value);
    int bits = ballot(cond, tid);
    unsigned warp_total = popc(bits);
    if(cond){
      bits = bits & my_mask;
      unsigned within_warp_idx = popc(bits);
      warp_buffer[warp_num_found+within_warp_idx] = k;
    }
    warp_num_found += warp_total;

    if(warp_num_found > 224){
      unsigned warp_global_output_idx = 0;
      if(lane_id == 0){
        warp_global_output_idx = atomicAdd(224);
      }

      warp_global_output_idx = broadcast_source_lane_zero(warp_global_output_idx, tid);
      unsigned * tmp_output_list = &output[warp_global_output_idx];
      unsigned j = lane_id;
```

```
      do{
        tmp_output_list[j] = warp_buffer[j];
        j+=32;
      }while(j<224);
      unsigned leftover = warp_num_found - 224;
      if(lane_id < leftover){
        warp_buffer[lane_id] = warp_buffer[224+lane_id];
      }
      warp_num_found = leftover;
    }
    k += 1024;
  }while(k<NUM_ROWS);

  unsigned last_full_write = (warp_num_found & 0xFFE0);
  unsigned warp_global_output_idx = 0;
  if(lane_id == 0){
    warp_global_output_idx = atomicAdd(last_full_write);
  }
  warp_global_output_idx = broadcast_source_lane_zero(warp_global_output_idx, tid);
  unsigned * tmp_output_list = &output[warp_global_output_idx];
  unsigned j = lane_id;
  while(j<last_full_write){
    tmp_output_list[j] = warp_buffer[j];
    j+=32;
  }
  unsigned leftover = warp_num_found - last_full_write;

  pthread_barrier_wait(&global_barrier); // actual SYNC among all 1024 threads

  if(lane_id == 0){
    warp_global_output_idx = atomicAdd(leftover);
  }
  warp_global_output_idx = broadcast_source_lane_zero(warp_global_output_idx, tid);
  tmp_output_list = &output[warp_global_output_idx];
  if(j < warp_num_found){
    tmp_output_list[lane_id] = warp_buffer[j];
  }

  return NULL;
}
```

1.2 Index Based Search Kernel

```
void * index_based_search(void * thread_id)
{
  int tid = (long long)thread_id;
  int lookup_counter = (tid >> 5);
  int lane_id = (tid & 0x1F);

  do{
    unsigned account_number_to_lookup = get_next_account_number_to_lookup(lookup_counter);
    int current_row_buffer_node = 0;

    for(int row_buffer_level = 0; row_buffer_level<3; row_buffer_level++){
      int simd_idx = (current_row_buffer_node<<10) + lane_id;
      int next_simd_idx = simd_idx + 32;

      // open the row
      unsigned my_compare_value = simd_sorted_account_numbers_2level[simd_idx];

      int count = popc(ballot((account_number_to_lookup > my_compare_value), tid));
      next_simd_idx += (count<<5);

      // subsequent row buffer hit
      unsigned my_next_compare_value = simd_sorted_account_numbers_2level[next_simd_idx];

      if(account_number_to_lookup == my_compare_value){
        write_result(simd_idx, lookup_counter);
      }
      if(account_number_to_lookup == my_next_compare_value){
        write_result(next_simd_idx, lookup_counter);
      }

      current_row_buffer_node = current_row_buffer_node*961 + 1 + count*31;
      current_row_buffer_node += popc(ballot((account_number_to_lookup > my_next_compare_value), tid));
    }

    lookup_counter += 32;
  }while(lookup_counter<NUM_SEARCHES);

  return NULL;
}
```

Bibliography

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proc. VLDB Endow.*, 2(2):1664–1665, Aug. 2009.

[2] A. Agarwal et al. April: a processor architecture for multiprocessing. In *ISCA-17*, 1990.

[3] J. R. Allen et al. Conversion of control dependence to data dependence. In *POPL*, 1983.

[4] B. Amrutur and M. Horowitz. Speed and power scaling of SRAMs. *IEEE JSCC*, 35(2):175–185, Feb. 2000.

[5] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[6] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[7] G. E. Blelloch. *Prefix Sums and Their Applications in John H. Reif (Ed.) Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1990.

[8] W. J. Bouknight et al. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, Apr. 1972.

[9] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[10] S. Collange et al. Dynamic detection of uniform and affine vectors in GPGPU computations. In *HPPC*, 2009.

[11] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. Simd re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 477–488, New York, NY, USA, 2011. ACM.

[12] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: Query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1061–1063, New York, NY, USA, 2007. ACM.

[13] W. W. L. Fung and T. Aamodt. Thread block compaction for efficient simt control flow. In *HPCA-17*, 2011.

[14] W. W. L. Fung et al. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO-40*, 2007.

[15] W. W. L. Fung et al. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM TACO*, 6(2):1–37, June 2009.

[16] M. Harris. *State of the Art in GPU Data-Parallel Algorithm Primitives*, 2010.

[17] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[19] W.-M. Hwu et al. Compute unified device architecture application suitability. *Computing in Science Engineering*, may-jun 2009.

[20] Intel Corporation. *Intel HD Graphics OpenSource Programmer Reference Manual*, 2010.

[21] N. Jayasena et al. Stream register files with indexed access. In *HPCA-10*, 2004.

[22] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[23] U. Kapasi et al. Efficient conditional operations for data-parallel architectures. In *MICRO-33*, 2000.

[24] B. Khailany et al. Vlsi design and verification of the imagine processor. In *ICCD*, 2002.

[25] Khronos Group. *OpenCL*. http://www.khronos.org/opencl.

[26] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[27] D. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2010.

[28] R. Krashinsky et al. The vector-thread architecture. In *ISCA-31*, 2004.

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

[30] A. Levinthal and T. Porter. Chap - a simd graphics processor. In *SIGGRAPH*, 1984.

[31] J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA-37*, 2010.

[32] G. Michelogiannakis, A. Williams, and J. Shalf. Collective memory transfers for multi-core chips. Technical report, Lawrence Berkeley National Laboratory, 2013.

[33] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 308–317, New York, NY, USA, 2011. ACM.

[34] R. Narayanan et al. MineBench: A benchmark suite for data mining workloads. In *IISWC*, 2006.

[35] Netezza, an IBM Company. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*, 2011.

[36] J. Nickolls and W. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56 –69, 2010.

[37] NVIDIA. *CUDA GPU Computing SDK*. http://developer.nvidia.com/gpu-computing-sdk.

[38] NVIDIA. *NVIDIA Kepler GK110 Next-Generation CUDA Compute Architecture*.

[39] NVIDIA. *PTX ISA Version 2.0*, 2010.

[40] NVIDIA. *CUDA C Best Practices Guide v5.5*, 2013. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[41] NVIDIA. *CUDA C Programming Guide Version 5.5*, 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[42] NVIDIA. *Parallel Thread Execution ISA Version 3.2*, 2013. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[43] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[44] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, May 2000.

[45] S. Rixner. Memory controller optimizations for web servers. In *MICRO-37*, 2004.

[46] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA-27*, 2000.

[47] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[48] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[49] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[50] A. Skjellum, D. Whittaker, and P. Bangalore. *Ballot Counting for Optimal Binary Prefix Sum*, 2010.

[51] B. J. Smith. A pipelined shared resource MIMD computer. In *ICPP*, 1978.

[52] J. E. Smith et al. Vector instruction set support for conditional operations. In *ISCA-27*, 2000.

[53] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[54] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS*, 1965.

[55] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO-34*, 2001.

[56] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation.

In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.

[57] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. *SIGARCH Comput. Archit. News*, 41(3):249–260, June 2013.

[58] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.