



PDF Download  
1543753.1543756.pdf  
16 February 2026  
Total Citations: 55  
Total Downloads: 4109

Latest updates: <https://dl.acm.org/doi/10.1145/1543753.1543756>

RESEARCH-ARTICLE

## Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware

**WILSON WAI LUN FUNG**, The University of British Columbia, Vancouver, BC, Canada

**IVAN SHAM**, The University of British Columbia, Vancouver, BC, Canada

**GEORGE YUAN**, The University of British Columbia, Vancouver, BC, Canada

**TOR AAMODT**, The University of British Columbia, Vancouver, BC, Canada

**Open Access Support** provided by:

**The University of British Columbia**

Published: 06 July 2009  
Accepted: 01 December 2008  
Revised: 01 November 2008  
Received: 01 April 2008

[Citation in BibTeX format](#)

# Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware

WILSON W. L. FUNG, IVAN SHAM, GEORGE YUAN, and TOR M. AAMODT  
University of British Columbia

Recent advances in graphics processing units (GPUs) have resulted in massively parallel hardware that is easily programmable and widely available in today's desktop and notebook computer systems. GPUs typically use single-instruction, multiple-data (SIMD) pipelines to achieve high performance with minimal overhead for control hardware. Scalar threads running the same computing kernel are grouped together into SIMD batches, sometimes referred to as warps. While SIMD is ideally suited for simple programs, recent GPUs include control flow instructions in the GPU instruction set architecture and programs using these instructions may experience reduced performance due to the way branch execution is supported in hardware. One solution is to add a stack to allow different SIMD processing elements to execute distinct program paths after a branch instruction. The occurrence of diverging branch outcomes for different processing elements significantly degrades performance using this approach. In this article, we propose dynamic warp formation and scheduling, a mechanism for more efficient SIMD branch execution on GPUs. It dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. We show that a realistic hardware implementation of this mechanism improves performance by 13%, on average, with 256 threads per core, 24% with 512 threads, and 47% with 768 threads for an estimated area increase of 8%.

---

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

An earlier version of this article appeared in the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07) [Fung et al. 2007]. The new material in this article consists of: (1) a reevaluation of our proposed mechanism with a baseline configuration providing a better match to existing GPUs to provide a more accurate performance-area analysis; (2) a more detailed description of an area-efficient implementation of our best performing warp scheduling policy (Majority); (3) an extended analysis of the sources of performance loss of our Majority scheduling policy; (4) a data cache bank conflict model is now incorporated into our baseline configuration; (5) sensitivity analysis of our proposed mechanism against various SIMD warp sizes and thread pool sizes; (6) a proposal to further reduce the area consumed by the warp pool while maintaining the ability to handle excessively diverging code via spilling to memory; (7) a discussion on the limitations of dynamic warp formation.

Author's address: Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, B.C., V6T 1Z4, Canada; email: {wwlfung, aamodt}@ece.ubc.ca.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.  
© 2009 ACM 1544-3566/2009/06-ART7 \$10.00

DOI 10.1145/1543753.1543756 <http://doi.acm.org/10.1145/1543753.1543756>

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms: Design, Performance

Additional Key Words and Phrases: SIMD, fine-grained multithreading, control flow, GPU

**ACM Reference Format:**

Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. 2009. Dynamic warp formation: Efficient SIMD control flow on SIMD graphics hardware. ACM Trans. Architec. Code Optim. 6, 2, Article 7 (June 2009), 37 pages.

DOI = 10.1145/1543753.1543756 <http://doi.acm.org/10.1145/1543753.1543756>.

---

## 1. INTRODUCTION

As semiconductor process technology advances continue and transistor density increases, computation potential continues to grow. However, finding effective ways to leverage process technology scaling for improving the performance of real-world applications has become increasingly challenging. To improve performance, hardware must exploit parallelism. Until recently, the dominant approach has been to extract more instruction level parallelism from a single thread through increasingly complex scheduling logic, larger caches, and sophisticated prediction mechanisms. As diminishing returns to ILP scaling begin to limit performance of single-threaded applications [Agarwal et al. 2000], attention has shifted toward employing additional resources to increase throughput by exploiting explicit thread-level parallelism in software (forcing software developers to share the responsibility for improving performance).

The modern graphics processing unit (GPU) can be viewed as an example of the latter approach [Purcell et al. 2002; Buck et al. 2004]. Earlier generations of GPUs consisted of fixed function 3D-rendering pipelines. New real-time rendering techniques required new hardware, which impeded the adoption of new graphics algorithms and thus motivated the introduction of programmability [Lindholm et al. 2001], long available in traditional offline computer animation [Upstill 1990], into GPU hardware for real-time computer graphics. In modern GPUs, much of the formerly hardwired pipeline is replaced with programmable hardware processors that run a relatively small program, called a shader, on each input vertex or pixel [Lindholm et al. 2001]. Shaders are either written by the application developer or substituted by the graphics driver to implement traditional fixed-function graphics pipeline operations. The compute model provided by modern graphics processors for running nongraphics workloads is closely related to stream processing [Rixner et al. 1998; Dally et al. 2003].

The programmability of shader processors has greatly improved over the past several years, and the shaders of the latest generation GPUs are turing-complete, opening up exciting new opportunities to speedup “general purpose” (i.e., nongraphics) applications. Based on experience gained from pioneering efforts to generalize the usage of GPU hardware [Purcell et al. 2002; Buck et al. 2004], GPU vendors have introduced new programming models and

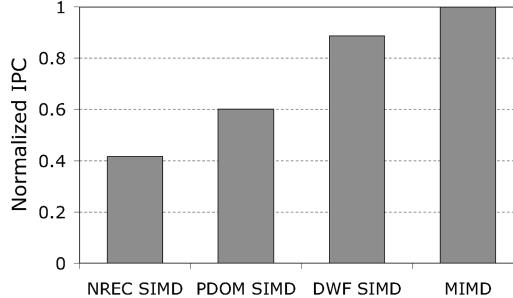


Fig. 1. Performance loss due to branch divergence when executing scalar SPMD threads using SIMD hardware. NREC is a naïve branch handling mechanism; PDOM is our baseline representation of modern GPU branch handling capability; DWF is our proposed approach; MIMD represents an upper bound.<sup>2</sup>

associated hardware support to further broaden the class of applications that may efficiently use GPU hardware [AMD, Inc. 2006; NVIDIA Corp. 2007a].

Even with a general-purpose programming interface, mapping existing applications to the parallel architecture of a GPU is a nontrivial task. Although some applications can achieve speedups of up to 431 times over a modern CPU [Hwu et al. 2007], other applications show less improvement when mapped to a GPU [Buatois et al. 2008]. One major challenge for contemporary GPU architectures is efficiently handling control flow in shaders [Shebanow 2007]. The reason is that, in an effort to improve computation density, modern GPUs typically batch together groups of individual threads running the same shader, and execute them together in lock step on a single-instruction, multiple-data (SIMD) pipeline [Lorie and Strong 1984; Montrym and Moreton 2005; Moy and Lindholm 2005; Luebke and Humphreys 2007; NVIDIA Corp. 2007a]. Such thread batches are referred to as warps<sup>1</sup> by NVIDIA [Lindholm et al. 2008; NVIDIA Corp. 2007a; Shebanow 2007]. This approach has worked well [Levinthal and Porter 1984] for graphics operations such as shadow volume generation [Crow 1977] and bump mapping [Blinn 1978], which historically have not required branch instructions. However, when shaders do include branches, the execution of different threads grouped into a warp to run on the SIMD pipeline may no longer be uniform across SIMD elements. This causes a hazard in the SIMD pipeline [Moy and Lindholm 2005; Woop et al. 2005] known as branch divergence [Lorie and Strong 1984; Shebanow 2007]. We found that naïve handling of *branch divergence* incurs a significant performance penalty on the GPU for control-flow intensive applications relative to an ideal multiple-instruction, multiple-data (MIMD) architecture with the same peak IPC capability (see Figure 1).

This article makes the following contributions:

- It establishes that, for the set of nongraphics applications we studied, reconverging control flow of processing elements at the immediate postdominator

---

<sup>1</sup>The term “warp” originates from weaving, the first parallel-thread technology in the textile industry [Lindholm et al. 2008]. In its original context, the term “warp” refers to “the threads stretched lengthwise in a loom to be crossed by the weft” [Fowler et al. 1995].

of the divergent branch is nearly optimal with respect to oracle information about the future control flow of each individual processing element for our baseline intrawarp branch reconvergence mechanism.

- It quantifies the performance gap between the immediate postdominator branch reconvergence mechanism and the performance that would be obtained on a MIMD architecture with support for the same peak number of operations per cycle. This performance gap highlights the importance of finding better branch handling mechanisms.
- It proposes and evaluates a novel hardware mechanism, dynamic warp formation, for regrouping processing elements of individual SIMD warps in hardware on a cycle-by-cycle basis to improve the efficiency of branch handling.
- It highlights quantitatively that warp scheduling policy (the order in which the warps are issued from the scheduler) is an integral part to both the performance and area overhead of dynamic warp formation, and proposes an area efficient implementation of a well-performing scheduling policy.

In particular, for a set of data parallel, nongraphics applications ported to our modern GPU-like multithreaded SIMD architecture, we find the speedup obtained by reconverging the diverging threads within a SIMD warp at the immediate postdominator of the diverging branch obtains a speedup of 45% over not reconverging. Dynamically regrouping scalar threads into SIMD warps on a cycle-by-cycle basis obtains an additional speedup of 13% with 256 threads per shader core. This additional speedup increases to 22%, 24%, 37%, and 47% with 384, 512, 640, and 768 threads per shader core, respectively (the 47% speedup with 768 threads per shader core corresponds to 114% speedup versus not reconverging). Using CACTI 4.2, we estimate the hardware required by this regrouping mechanism would add 8% to the total chip area if it were implemented on a contemporary GPU, such as the NVIDIA GeForce 8800GTX (470mm<sup>2</sup>) [Lindholm et al. 2008].

The rest of this article is organized as follows: Section 2 gives an overview of the baseline architecture used in this article. Section 3 describes the immediate postdominator control-flow reconvergence mechanism and evaluates the potential of intrawarp stack-based reconvergence mechanisms with a limit study. Section 4 describes our proposed dynamic warp formation mechanism. Section 5 describes the simulation methodology of the proposed GPU microarchitecture. Section 6 describes our experimental results. Section 7 estimates the area overhead to implement our proposed mechanism. Section 8 discusses some limitations of dynamic warp formation. Section 9 describes related work. Section 10 summarizes this article and suggests future work.

## 2. BASELINE ARCHITECTURE

This section describes the compute model and the baseline SIMD GPU microarchitecture used for the rest of this article.

---

<sup>2</sup>*NREC SIMD* and *PDOM SIMD* are described in Section 3, while *DWF SIMD* is described in Section 4. Benchmarks and microarchitecture used here are described in Section 5.

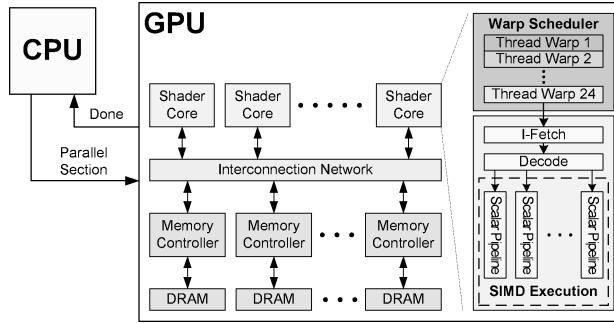


Fig. 2. Baseline GPU microarchitecture. Blocks labeled “scalar pipeline” include register read, execute, memory, and writeback stages.

## 2.1 Compute Model

In this article, we have adopted a compute model that is similar to NVIDIA’s CUDA programming model [NVIDIA Corp. 2007a]. In this compute model, the application starts off as a single thread running on the CPU. At some point during execution, the CPU reaches a kernel call and spawns a parallel section to the GPU to exploit data-level parallelism. At this point, the CPU will then stop its execution and wait for the GPU to finish the parallel section.<sup>3</sup> This sequence can repeat multiple times until the program completes.

Each parallel section consists of a collection of threads executing the same code, which we call a thread program. Similar to many thread programming APIs, a thread program is encapsulated as a function call. In our implementation, at least one of the arguments is dedicated to passing in the thread ID, which each thread uses to help determine its behavior during the parallel section. For example, each thread may use its ID to select which data to operate upon. In this sense, the programming model is essentially the *Single-Program, Multiple-Data* (SPMD) model commonly used in large scale shared memory multiprocessors.

All threads within a parallel section execute in parallel and share the same memory space. Cache coherence and memory consistency are not enforced in our model, as threads running on different shader cores are independent of each other (similar to CUDA applications that do not make use of the “atomic” global memory operations added in CUDA 1.1 [NVIDIA Corp. 2007a]). In the present study, to ensure that threads of the next parallel section will have access to the correct data, data caches are flushed and a memory fence operation is performed at the end of each parallel section.

## 2.2 SIMD GPU Microarchitecture

Figure 2 illustrates the baseline microarchitecture used in the rest of this article. In this figure, the GPU is composed of several shader cores connected via a interconnection network to several independent DRAM controllers. Each

<sup>3</sup>CUDA version 1.1 and higher allows the CPU to continue execution in parallel with the GPU [NVIDIA Corp. 2007a].

shader core executes multiple parallel threads from the same parallel section, with each thread’s instructions executed in order by the hardware.<sup>4</sup> The multiple threads on a given shader core are grouped into SIMD warps by the scheduler. Each warp of threads executes the same instruction simultaneously on different data values in parallel scalar pipelines (i.e., vector lanes). Operands are read from and written to a wide register file in parallel. Memory requests access a highly banked data cache and cache misses are forwarded to memory controllers and/or higher level caches via an interconnection network. Each memory controller processes memory requests by accessing its associated DRAM, possibly in a different order than the requests are received so as to reduce row-activate and precharge overheads [Rixner et al. 2000]. The interconnection network we simulated is a crossbar with a parallel iterative matching allocator [Dally and Towles 2004].

Since our focus in this article is nongraphics applications, graphic-centric details are omitted from Figure 2. However, traditional graphics processing still heavily influences this design: The use of what is essentially SIMD hardware to execute SPMD software (with possible interthread communication) is heavily motivated by the need to balance efficient “general purpose” computing kernel execution with a large quantity of existing and important (to GPU vendors) graphics software that has few, if any, control-flow operations in its shaders [Shebanow 2007]. However, it is important to recognize that thread programs for graphics (i.e., shaders) may very well make increasing use of control flow operations in the future, for example, to achieve more realistic lighting effects.

### 2.3 Latency Hiding

Because texture workloads in graphics rendering normally exceed the capacity of on-chip caches, cache miss rates can be high even though significant locality exists [Hakura and Gupta 1997], which may severely lower performance if the pipeline had to stall for every cache miss. This is especially true when the latency of memory requests can take several hundreds of cycles due to the combined effects of contention in the interconnection network and row-activate and precharge overheads at the DRAM. While traditional microprocessors can mitigate the effects of cache misses using out-of-order execution, a more compelling approach when software provides the parallelism is to interleave instruction execution from different threads.

With a large number of shader threads multiplexed on the same execution resources, our architecture employs fine-grained multithreading (FGMT), also known as barrel processing, where individual threads are interleaved by the fetch unit to proactively hide the potential latency of stalls before they occur [Thornton 1964; Thistle and Smith 1988]. As illustrated in Figure 3(a), instructions from multiple shader threads are issued fairly in a round-robin queue. When a shader thread is blocked by a memory request, the corresponding shader core simply removes that thread’s warp from the pool of “ready” warps and thereby allows other shader threads to proceed while the memory

---

<sup>4</sup>A shader core in our study is akin to CUDA’s notion of a Streaming Multiprocessor [Lindholm et al. 2008].

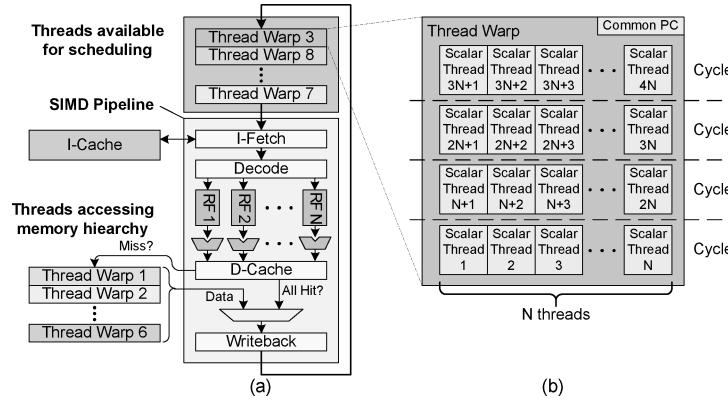


Fig. 3. Detail of a shader core. (a) Using fine-grained multithreading to hide data memory access latency.  $N$  is the SIMD width of the pipeline. (b) Grouping  $4N$  scalar threads into a SIMD warp executed over 4 cycles.

system processes its request. With a large number of threads (768 per shader core, 12,288 in total in this article) interleaved on the same pipeline, FGMT effectively hides the latency of most memory operations, since the pipeline is occupied with instructions from other threads while memory operations complete. Barrel processing also hides the pipeline latency so that data bypassing logic can potentially be omitted to save area with minimal impact on performance. In this article, we also simplify the dependency check logic design by restricting each thread to have at most one instruction running in the pipeline at any time.

An alternative to FGMT with a large number of threads is to interleave fewer threads, but provide many registers to each thread [Intel Corporation 2008]. Each thread executes on the pipeline until it encounters a data dependency upon an outstanding long latency memory request, at which time the pipeline will switch execution to another thread. Latency hiding is achieved via software loop unrolling, which generates independent memory access operations, which can be overlapped to achieve the memory level parallelism required for effective use of the graphics memory subsystem.

#### 2.4 SIMD Execution of Scalar Threads

While FGMT can hide memory latency with relatively simple hardware, a modern GPU must also exploit the explicit parallelism provided by the data-parallel programming model [Buck et al. 2004; NVIDIA Corp. 2007a] associated with programmable shader hardware to achieve maximum performance at minimum cost. SIMD hardware [Bouknight et al. 1972] can efficiently support SPMD program execution provided that individual threads follow similar control flow paths. Figure 3(b) illustrates how instructions from multiple ( $M = 4N$ ) shader threads are grouped into a single SIMD warp and scheduled together on multiple ( $N$ ) scalar pipelines over several ( $M/N = 4$ ) consecutive clock cycles. The multiple scalar pipelines execute in “lock-step” and all control logic may be shared to greatly reduce area relative to a MIMD architecture. A significant

source of area savings for such a SIMD pipeline is the simpler instruction cache support required for a given number of scalar threads.

SIMD can also be used to relax the latency requirement of the scheduler and simplify the scheduler’s hardware. With a SIMD warp size wider than the actual SIMD hardware pipeline, the scheduler only needs to issue a single warp every  $M/N$  cycles ( $M=\text{warp size}$ ,  $N=\text{pipeline width}$ ) [Lindholm et al. 2008]. The scheduler’s hardware is also simplified, as it has fewer warps to manage. Similar to the NVIDIA’s GeForce 8 series GPU [Lindholm et al. 2008], the performance evaluations presented in this article assume the use of this technique (see Section 4.4).

### 3. SIMD CONTROL FLOW SUPPORT

To ensure the hardware can be easily programmed for a wide variety of applications, some recent GPU architectures provide branch instructions that allow individual scalar threads to follow distinct program paths [NVIDIA Corp. 2007a; AMD, Inc. 2006] while executing on SIMD hardware. We note that where it applies, predication [Allen et al. 1983] is a natural way to support fine-grained control flow on a SIMD pipeline. With branches on superword condition codes (BOSCCs) [Shin et al. 2007], a predicated instruction is skipped entirely if all threads in a warp evaluate to false. BOSCCs work effectively for strongly biased branches, such as loops with no early exits and if-branches for error handling where all threads are likely to follow the same control flow. With weakly biased branches, false predicates introduce significant overhead.

To support distinct control flow operation outcomes on distinct processing elements with loops and function calls, several approaches have been proposed: Lorie and Strong describe a mechanism using mask bits along with special compiler-generated priority encoding “else” and “join” instructions [Lorie and Strong 1984]. Lindholm and Moy [2005] describe a mechanism for supporting branching using a serialization mode. Woop et al. [2005] describe the use of a hardware stack and masked execution. Kapasi et al. [2000] propose conditional streams, a technique for transforming a single kernel with conditional code to multiple kernels connected with interkernel buffers. These and other approaches are discussed in Section 9.

The effectiveness of a SIMD pipeline is based on the assumption that all threads running the same thread program expose identical control-flow behavior. While this assumption holds for most existing graphics shaders [Shebanow 2007], many parallel nongraphics applications (and potentially, future graphics shaders) tend to have more diverse control-flow behavior. When a branch leads to different control flow paths for different threads in a warp, branch divergence occurs because a SIMD pipeline cannot execute different instructions in the same cycle. The following sections describe two baseline techniques for handling branch divergence, both of which were implemented in our simulator.

#### 3.1 SIMD Serialization

A naïve solution to handle branch divergence in a SIMD pipeline is to serialize the threads within a warp after their program counters diverge. While this

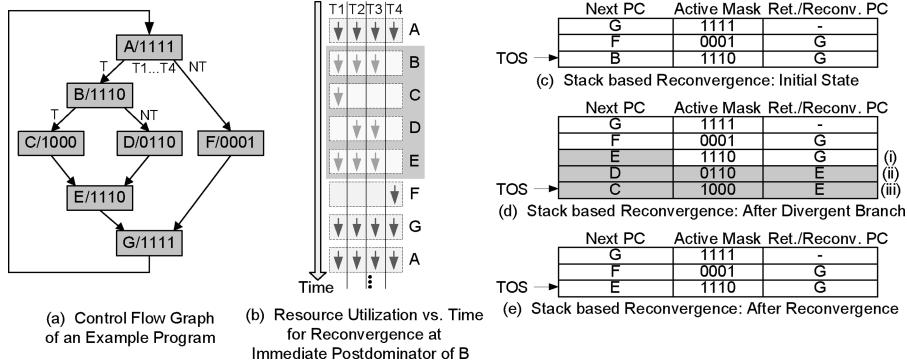


Fig. 4. Implementation of immediate postdominator-based reconvergence.

approach is simple, it achieves poor performance. A single warp with branch divergence is effectively separated into multiple warps, each containing threads taking the same execution path. These warps are then scheduled and executed independently of each other, and they never reconverge back into a single warp. Without branch reconvergence, threads within a warp will continue diverging until each thread is executed in isolation from other threads in the original warp, leading to very low utilization of the parallel functional units, as shown by the first bar in Figure 1.

### 3.2 SIMD Reconvergence (Baseline Branch Handling Mechanism)

Given the drawback of serialization it is desirable to use a mechanism for reconverging control flow. The opportunity for such reconvergence is illustrated in Figure 4(a). In this example, threads in a warp diverge after reaching the branch at A. The first three threads encounter a taken branch and go to basic block<sup>5</sup> B (denoted by the bit mask 1110 in Figure 4(a)), while the last thread goes to basic block F (denoted by the bit mask 0001 in Figure 4(a)). The three threads executing basic block B further diverge to basic blocks C and D. However, at basic block E the control-flow paths reach a join point [Muchnick 1997]. If the threads that diverged from basic block B to C wait before executing E for the threads that go from basic block B to basic block D, then all three threads can continue execution simultaneously at block E. Similarly, if these three threads wait after executing E for the thread that diverged from A to F, then all four threads can execute basic block G, simultaneously. Figure 4(b) illustrates how this sequence of events would be executed by the SIMD function units. In this part of the figure, solid arrows indicate SIMD units that are active.

The behavior described earlier can be achieved using a stack-based reconvergence mechanism [Woop et al. 2005]. In this article, we use the mechanism shown in Figure 4(c,d,e) as our baseline. Here, we show how the stack is updated as the group of three threads in Figure 4(a) that execute B diverge and

<sup>5</sup>A basic block is a sequence of consecutive statements in which flow of control enters only at the beginning and leaves only at the end without the possibility of branching except at the end [Aho et al. 1986].

then reconverge at E. Before the threads execute the diverging branch at B, the state of the stack is as shown in Figure 4(c). When the branch divergence is detected after executing B, the stack is modified to the state shown in Figure 4(d). The changes that occur are the following:

- (1) The original top of stack (TOS) in Figure 4(c), also at (i) in Figure 4(d), has its next PC field modified to the instruction address of the reconvergence point E (this address could be specified through an extra field in the branch instruction);
- (2) A new entry (ii) is allocated onto the stack and initialized with the next PC value of the fall through of the branch (D) and a mask encoding, which processing elements evaluated the branch as “not-taken” (0110) along with the reconvergence point address (E);
- (3) A new entry (iii) is allocated onto the stack with the target address (C) of the branch, the mask (1000) encoding the processing element that evaluated the branch as “taken”, and the reconvergence point address (E).

Whenever the new next PC of the top entry on the stack equals the reconvergence PC of the TOS entry, the top entry is immediately popped (before being used to fetch an instruction), and the value of the next PC field from the next entry in the stack is used to fetch the next instruction. Note that this mechanism easily supports complex “nested” branch hammocks, irreducible control flow [Muchnick 1997], as well as data-dependent loops.

In this article we use the immediate postdominator [Muchnick 1997] of the diverging branch instruction as the reconvergence point.<sup>6</sup> A *postdominator* is defined as follows: A basic block X postdominates basic block Y (written as “X pdom Y”) if and only if all paths from Y to the exit node (of a function) go through X. A basic block X, distinct from Y, immediately postdominates basic block Y if and only if X pdom Y and there is no basic block Z distinct from Y and X such that X pdom Z and Z pdom Y. Immediate postdominators are typically found at compile time as part of the control flow analysis necessary for code optimization.

The performance impact of the immediate postdominator reconvergence technique (labeled PDOM throughout this article) depends upon the SIMD warp size. Figure 5 shows the harmonic mean IPC of the benchmarks studied in Section 6 compared to the performance of MIMD hardware for 8, 16, and 32 wide SIMD execution assuming 16 shader cores. Execution unit utilization decreases from 26.3% for MIMD to 21.1% for 8-thread, to 18.7% for 16-thread, and down to 15.8% for 32-thread SIMD warps.<sup>7</sup> This increasing performance loss is due to the higher impact of branch divergence when SIMD warps are wider and each warp contains more threads. Thus, using 32-thread SIMD warps results in a 40% slowdown, so performance can potentially improve by 66% with a better branch handling mechanism.

---

<sup>6</sup>While Rotenberg et al. [1999] also identified immediate postdominators as control reconvergence points in the context of superscalar processor, to our knowledge, we are the first to propose this scheme for SIMD control flow.

<sup>7</sup>MIMD is < 100% due to the memory bandwidth requirements of our benchmarks.

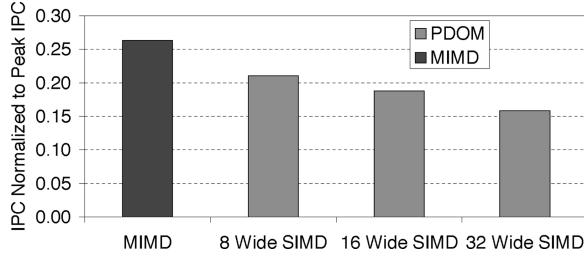


Fig. 5. Performance loss for PDOM versus SIMD warp size (realistic memory system). An 8-wide SIMD pipeline is used for all configurations.

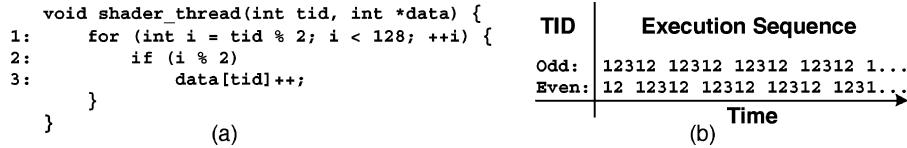


Fig. 6. (a) Contrived example for which reconvergence at points beyond the immediate postdominator yields a significant improvement in performance like that shown in Figure 7. The parameter `tid` is the thread ID. (b) Execution sequence for threads with odd or even `tid` (numbers refer to lines in part(a)).

In the following section, we explore whether immediate postdominators are the “best” reconvergence points, or whether there might be a benefit to dynamically predicting a reconvergence point past the immediate postdominator (we will show a contrived code example where this is beneficial).

### 3.3 Reconvergence Point Limit Study

While a mechanism for reconverging at the immediate postdominator is able to recover much of the performance lost due to branch divergence compared to not reconverging at all, Figure 6(a) shows an example where this reconvergence mechanism is suboptimal. In this example, threads with even value of `tid` diverge from those with odd values of `tid` each iteration of the loop. If even threads allow the odd threads to “get ahead” by one iteration, all threads can execute in lock-step until individual threads reach the end of the loop. This suggests that reconverging at points beyond the immediate postdominator may yield better performance. To explore this possibility, we conducted a limit study, assessing the impact of always predicting the best reconvergence point assuming oracle knowledge of each thread’s future control flow.

For this limit study, dynamic instruction traces are captured from only the first 128 threads. SIMD warps are formed by grouping threads by increasing thread ID, and an optimal alignment for the instruction traces of each thread in a warp is found via repeated applications of the Needleman-Wunsch algorithm [Needleman and Wunsch 1970]. With four threads per warp, the optimal alignment is found by exhaustively searching all possible pair-wise alignments between threads within a warp. The best reconvergence points are then identified from the optimal alignment.

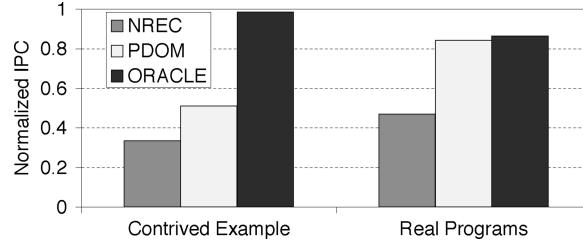


Fig. 7. Impact of predicting optimal SIMD branch reconvergence points (SIMD width=4). NREC=No Reconvergence. PDOM=reconvergence at immediate postdominators. ORACLE=reconvergence at optimal postdominators.

Figure 7 compares performance of not reconverging (NREC), immediate postdominator reconvergence (PDOM), and reconverging to the points found using the previously described method (ORACLE), assuming a warp size of 4. In this figure, we assume an idealized memory system (all cache accesses hit) and examine both a contrived program with the behavior abstracted in Figure 6 and the seven benchmarks described in Section 5 (depicted by the bars labeled Real Programs). While the contrived example experiences a 92% speedup with ORACLE versus PDOM, the improvement on the real programs we studied is much less (2.6%). Interestingly, one of the benchmarks (bitonic sort) also has similar even/odd thread dependence as our contrived example, but it also has barrier synchronizations forcing loop iterations to run in lock-step. While this limit study indicates that reconverging at the immediate postdominator is a good heuristic, in the next section, we present a mechanism for improving performance significantly relative to PDOM.

It is important to recognize the limitations of this limit study: We explored a limited set of benchmarks and used short SIMD width. It may be valuable to repeat this study on a larger set of applications with wider SIMD width.

#### 4. DYNAMIC WARP FORMATION AND SCHEDULING

While the postdominator reconvergence mechanism can recover performance loss due to diverging branches, it does not fully utilize the SIMD pipeline relative to a MIMD architecture with the same peak IPC capability (resulting in a 40% slowdown relative to MIMD for a warp size of 32). In this section, we describe our proposed hardware mechanism for recovering the lost performance potential of the hardware.

The performance penalty due to branch divergence is hard to avoid with only one thread warp, since the diverged parts of the warp cannot execute simultaneously on the SIMD hardware in a single cycle. Dynamic warp formation attempts to improve upon this by exploiting the FGMT aspect of the GPU microarchitecture: With FGMT employed to hide memory-access latency, there is usually more than one thread warp ready for scheduling in a shader core. Every cycle, the thread scheduler tries to form new warps from a pool of active threads by combining scalar threads whose next program counter (PC) values are the same. As the thread program executes, diverged warps are broken up into scalar threads to be regrouped into new warps according to their branch

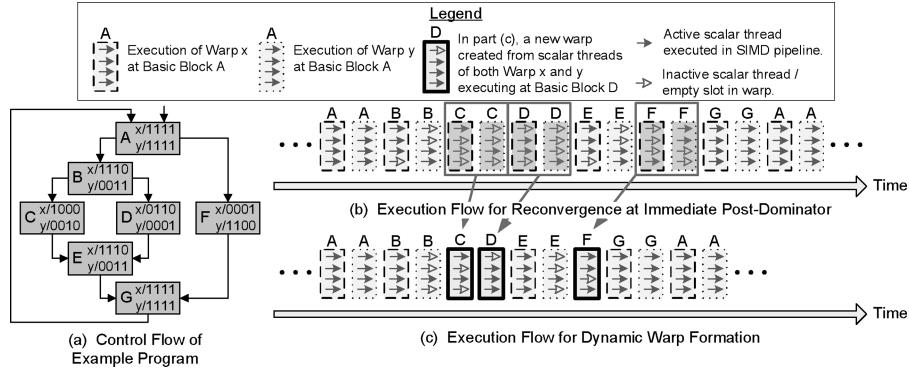


Fig. 8. Dynamic warp formation example.

targets (indicated by the next PC value of each scalar thread). In this way, the SIMD pipeline is fully utilized even if a thread program consists of diverging branches.

Figure 8 illustrates this idea. In this figure, two warps, Warp x and Warp y, are executing the example program shown in Figure 8(a) on the same shader core and both suffer from branch divergence. Figure 8(b) shows the interleaved execution of both warps using the stack-based reconvergence technique discussed in Section 3.2, which results in a SIMD pipeline utilization below 50% when basic blocks C, D, and F are executed. As shown in Figure 8(c), using dynamic warp formation to regroup scalar threads from both warps in Figure 8(b) into a single warp in Figure 8(c) with more active threads for these blocks can significantly increase average pipeline utilization (from 66% to 81%).

Implementing dynamic warp formation requires careful attention to the details of the register file, a consideration we explore in Section 4.1. In addition to forming warps, the thread scheduler also selects one warp to issue to the SIMD pipeline every scheduler cycle depending upon a scheduling policy. We explore the design space of this scheduling policy in Section 4.3. We show that thread scheduler policy is critical to the performance impact of dynamic warp formation in Section 6.1.

#### 4.1 Register File Access

To reduce area and support high-bandwidth access to data in a SIMD pipeline, a well-known approach is to implement a register file with registers for different processing elements (lanes of a SIMD pipeline) packed into a single-wide register. Registers for different elements are accessed by a single decoder, as shown in Figure 9(a). This hardware is a natural fit when threads are grouped into warps “statically” before they begin executing instructions and each thread stays in the same lane until it is completed. For our baseline architecture, each wide register contains a scalar register for each lane. Each scalar thread in a warp is statically assigned to a unique lane and always accesses the corresponding portion of the wide registers associated with that lane. The registers used by each scalar thread within a given lane are then assigned statically at a given offset based on the warp ID.

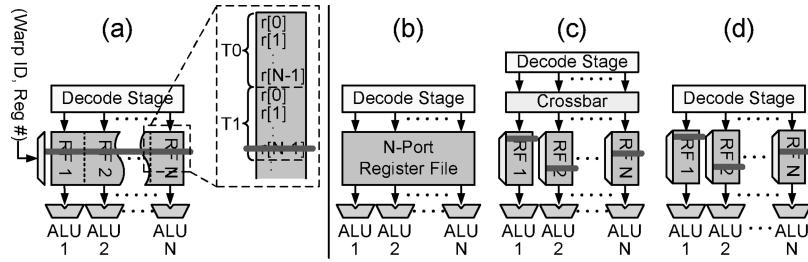


Fig. 9. Register file configuration for (a) static warp formation, (b) ideal dynamic warp formation and MIMD, (c) unconstrained dynamic warp formation with potential for bank conflicts, (d) lane aware dynamic warp formation. The line running across each lane represents whether registers in different lanes are all addressed by a common decoder (continuous line in part (a)) or each lane has its own decoder for independent addressing ((c) and (d)). In part (a), the blown up section illustrates the layout of scalar registers ( $r[0] \dots r[N-1]$ ) for individual threads ( $T_0, T_1$ ) in a lane.

However, we have not explicitly considered the impact of such static register assignment on dynamic warp formation. As described so far, dynamic warp formation would require each register to be equally accessible from all lanes, as illustrated in Figure 9(b). While grouping threads into warps dynamically, it is preferable to avoid the need to migrate register values with threads as they are regrouped into different warps. To accomplish this, organizations, such as those shown in Figure 9(b,c,d), allow the registers used by each scalar thread to be assigned statically to the lanes in the same way as described previously. If we dynamically form new warps from scalar threads with identical next PC values without consideration of the “home” lane of a scalar thread’s registers but wish to avoid a heavily multported register file organization, such as in Figure 9(b), we must design a multibanked register file, where each bank contains registers of threads with the same “home” lane, with a crossbar as in Figure 9(c). Warps formed dynamically may then have two or more threads with the same “home” lane, resulting in bank conflicts. These bank conflicts introduce stalls into all lanes of the pipeline and significantly reduce performance, as shown in Section 6.4.

A better solution, which we call *lane aware* dynamic warp formation, ensures that each thread remains within its “home” lane. In particular, lane aware dynamic warp formation assigns a thread to a warp only if that warp does not already contain another thread in the same lane. While the crossbar in Figure 9(c) is unnecessary for lane aware dynamic warp formation, the traditional hardware in Figure 9(a) is insufficient. When a particular set of threads are grouped into a warp “statically,” each thread’s registers are at the same offset within the corresponding lane, thus requiring only a single decoder. With lane aware dynamic warp formation, the offsets to access a register in a warp will not be the same in each lane.<sup>8</sup> This yields the register file configuration shown in Figure 9(d), which is used for our performance and area estimations in Section 6 and Section 7.

<sup>8</sup>Note that each lane is still executing the same instruction in any given cycle—the varying offsets are a byproduct of supporting fine-grained multithreading to hide memory access latency combined with dynamic warp formation.

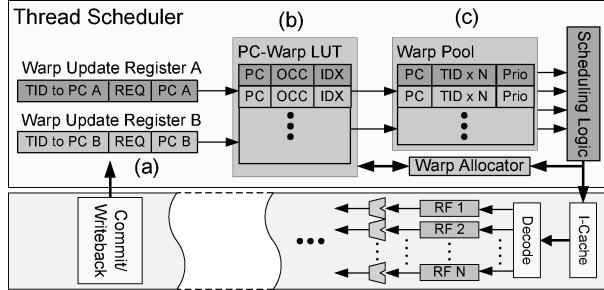


Fig. 10. Implementation of dynamic warp formation and scheduling. In this figure,  $N$  is the width of the SIMD warp. See text for a detailed description.

One subtle performance issue affecting the impact of lane aware scheduling for one of our benchmarks (Bitonic) is related to the type of pathological even/odd thread identifier control dependence described in Section 3.3. For example, if all warps have threads in even lanes that evaluate a branch as taken, while threads in odd lanes evaluate the same branch as not-taken, then it is impossible for dynamic warp formation to create warps with more active threads. A simple solution we employ for all applications is to alternately swap the position of even and odd thread home lanes every other warp when threads are first created (an approach we call *thread swizzling*).

#### 4.2 Hardware Implementation

Figure 10 shows a high-level block diagram illustrating how dynamic warp formation can be implemented in hardware. The two warp update registers, labeled (a) in Figure 10, store information for different target PCs of an incoming, possibly diverged warp; the PC-warp LUT (b) provides a level of indirection to guide diverged threads to an existing or newly created warp in the warp pool (c). The warp pool is a staging area holding all the warps ready to be issued to the SIMD pipeline. A detailed implementation of the scheduling logic, assuming the Majority scheduling policy introduced in Section 4.3, is proposed in Section 4.4.

When a warp arrives at the last stage of the SIMD pipeline, its threads' identifiers (TIDs) and next PC(s) are passed to the thread scheduler (Figure 10(a)). For conditional branches, there are at most two different next PC values.<sup>9</sup> For each unique next PC sent to the scheduler from writeback, the scheduler looks for an existing entry in the PC-warp LUT already mapped to the PC and allocates a new entry if none exists<sup>10</sup> (Figure 10(b)).

The PC-warp LUT (Figure 10(b)) provides a level of indirection to reduce the complexity of locating warps in the warp pool (Figure 10(c)). It does this by using the IDX field to point to a warp being formed in the warp pool. This warp is updated with the thread identifiers of committing threads having this

<sup>9</sup>Indirect branches that diverge to more than two PCs can be handled by stalling the pipeline and sending up to two PCs to the thread scheduler every cycle.

<sup>10</sup>In our detailed model, we assume the PC-warp LUT is organized as a small dual-ported 4-way set associative structure. Sets in the PC-warp LUT are indexed by the lower bits of the PC.

next PC value. Each entry in the warp pool contains the next PC value of the warp, N TID entries for hardware supporting N-wide warps, and some policy-specific data (labelled “Prio”) for scheduling logic. A design to handle the worst case where each thread diverges to a different execution path would require the warp pool to have enough entries for each thread in a shader core to have its own entry. However, we observe that for the benchmarks we simulated only a small portion of the warp pool is used, and we can shrink the warp pool significantly to reduce area overhead without causing a performance penalty (see Section 6.8). This much smaller warp pool can still support the worst-case where each thread has diverged, by employing the spilling mechanism described in Section 4.5.

To implement the lane aware scheduler mentioned in Section 4.1, each entry in the PC-warp LUT has an occupancy vector (OCC) tracking, which lanes of the current warp are free. This is compared against the request vector (REQ) of the warp update register that indicates which lanes are required by the threads assigned to one portion of a newly diverged warp. If OCC indicates that a required lane is already occupied by a thread in the warp pointed to by IDX, a new warp will be allocated and the TIDs of the threads causing the conflict will be assigned into this new warp. The TIDs of the threads that do not cause any conflict will be assigned to the original warp pointed to by IDX. In the case of such conflicts, the PC-warp LUT IDX field is updated to point to the new warp in the warp pool. The warp with the older PC still resides in the warp pool but will no longer be updated.

Each scheduler cycle a single warp in the warp pool may be issued to the SIMD pipeline according to one of the scheduling policies described in the next section. A forwarding path is added to allow updating of the warp that is issued in the same scheduler cycle to allow concurrently merging warps (after a decision has been made to issue the warp). This is possible because instruction cache access does not require TID information, so the TID can potentially be updated as the warp traverses the fetch stage, reducing the effective pipeline depth. Once issued, the associated warp pool entry is returned to the warp allocator.

#### 4.3 Scheduling Policies

Even though dynamic warp formation has the potential to fully utilize the SIMD pipeline, this will only happen when the set of PC values currently being executed is small relative to the number of scalar threads. If each scalar thread progresses at a substantially different rate, then all threads will eventually map to entirely different PCs. To avoid this, all threads should have a similar rate of progress. We have found that the warp scheduling policy, namely, the order in which warps are issued from the warp pool, has a critical effect on performance due to this effect (as shown in Section 6.1). We evaluated the following policies:

*Time Stamp* (DTime). Warps are issued in the order they arrive at the scheduler. When a warp triggers a data cache miss, it is taken out of the scheduler until its requested data arrives from memory. This may change the order that warps are issued.

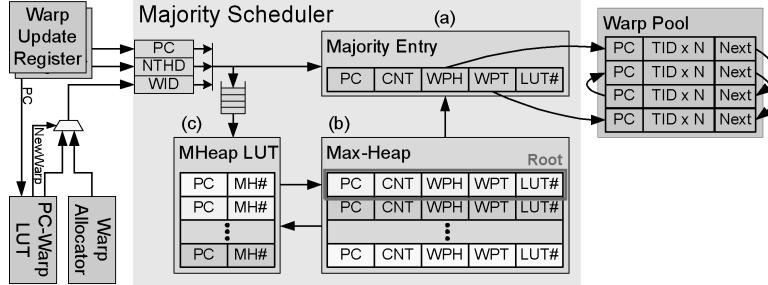


Fig. 11. Implementation of majority scheduling policy with Max-Heap. See text for details.

*Program Counter* (DPC). In a program sequentially laid out in instruction memory, the program counter value itself may be a good indicator of a thread’s progress. By giving higher issue priority to warps with smaller PCs, threads lagging behind are given the opportunity to catch up.

*Majority* (DMaj). As long as a majority of the threads are progressing at the same rate, the scheduling logic will have a large pool of threads with similar PC values from which it can create new warps every cycle. The majority policy attempts to encourage this behavior by choosing the most common PC among all the existing warps and issuing all warps at this PC before choosing a new PC.

*Minority* (DMin). If a small minority of threads diverges away from the rest, the majority policy tends to leave these threads behind. In the minority policy, warps with the least frequent PCs are given priority with the hope that, by doing so, the threads in these warps may eventually catch up and converge with other threads.

*Postdominator Priority* (DPdPri). Threads falling behind after a divergence need to catch up with other threads after the immediate postdominator. If the issue priority is set lower for warps that have gone beyond more postdominators, then the threads that have not yet gone past the postdominator tend to catch up.

In Section 6, we show that Majority scheduling policy achieves the best result over the benchmarks we evaluated. In Section 6.2, we show that DPdPri’s remedies certain performance deficiencies of DMaj identified for one of our benchmarks with simple control-flow behavior. While DPdPri is better in some cases, the number of postdominators encountered by a thread does not always accurately represent a thread’s progress through a program with nested control flow, resulting in lost opportunities to group threads into warps. The simulation results in Section 6.4 that take into account a realistic implementation with a Max-Heap indicate that, overall, the DMaj scheduling policy offers better performance.

#### 4.4 A Majority Scheduling Policy Implementation

Majority scheduling, the policy we found to work best (as shown in Section 6.1), can be implemented in hardware with a Max-Heap and a look-up table, as shown in Figure 11. The performance and area impact of this hardware implementation is carefully evaluated in Sections 6 and 7, respectively. In Figure 11, the Majority Entry (a) keeps track of the current majority PC value

and a group of warps with this PC value<sup>11</sup>; the Max-Heap (b) is a full binary tree (including a root entry) of PC values (and its group of warps) sorted by number of threads with this PC using the algorithm described in Cormen et al. [2001], and the MHeap LUT (c) provides a level of indirection for incoming warps to update their corresponding entries in the Max-Heap (similar to the function of the PC-warp LUT but for the Max-Heap rather than the Warp Pool). While a simple Max-Heap performing one swap per cycle per updated warp is sufficient for our usage, using a pipelined max-heap structure, such as those explored by Ioannou and Katevenis [2007] and Basu et al. [2007], may provide an even better hardware implementation by further reducing the bandwidth (and hence area) requirements of the Max-Heap.

Each entry in the Max-Heap represents a group of warps with the same PC value, and this group of warps has CNT threads. This group of warps forms a “linked list” in the warp pool with a “Next” entry referring to the next warp in the list. WPH is the index of the head of this list in the Warp Pool and WPT is the index of the tail of the list (warps are inserted at the end of the “list”). LUT# is the index of the corresponding entry in the MHeap LUT and is used to update the corresponding MH# entry in the LUT after a swap (see details later in the text).

**4.4.1 Warp Insertion.** In the implementation of the majority scheduler that we consider, a 32-thread warp is issued over 4 cycles. Once a (possibly diverged) warp leaves the writeback stage of the pipeline and enters the scheduler, it is processed as follows. When a warp arrives at the thread scheduler, its threads are divided among the two warp update registers, as before. Each warp update register then sends a warp update consisting of the following information to the Majority scheduler: the warp’s next PC value (labeled PC in Figure 11), the number of threads of this warp (NTHD), and the ID of the warp in the warp pool (WID) (acquired either via the PC-warp LUT or warp allocator, as in Section 4.2). Inside the Majority scheduler, the next PC value of each incoming warp update request is compared against the one stored in the Majority Entry. If they match, the Majority Entry is updated directly; otherwise, the warp update request is pushed into an update queue to the Max-Heap.

Every cycle up to two warp updates are processed in parallel from the update queue. For each update, the MHeap LUT is searched to provide the index (MH#) to an existing Max-Heap entry with a matching next PC value. If such an entry is not found in the MHeap LUT, an index of a new entry in the Max-Heap is obtained. The Max-Heap entry, at the location identified by MH#, is then updated (toward the end of the same cycle): CNT is incremented by NTHD, and if WID is different from WPT, WPT will be updated with WID, and WID will be assigned to the “Next” entry in the warp referred to by the original WPT. (This is essentially attaching the warp referred to by WID to the end of the linked list of this Max-Heap entry). If this is a newly inserted entry in the Max-Heap, the value of WID will be assigned to both WPH and WPT.

---

<sup>11</sup>This entry is separated because we finish executing all warps at a given PC value before selecting a new PC value.

In subsequent cycles after the entries in the Max-Heap are updated or inserted, each of these modified entries is compared with its parent entry and swapped up the binary tree until it encounters a parent with a larger thread count (CNT) or it becomes the root of the binary tree (which may be popped from the Max-Heap and becomes the next Majority Entry). Whenever a swap occurs, the corresponding entry in MHeap LUT (denoted by LUT# in the swapping Max-Heap entries) is updated. During this operation, no entries in the Max-Heap can be updated. New warp updates generated by incoming warps in the meantime are buffered in the update queue.

**4.4.2 Warp Issue.** In every scheduler cycle,<sup>12</sup> a warp from the warp pool will be issued to the SIMD pipeline. If the Majority Entry has any warps remaining ( $CNT > 0$ ), the warp denoted by WPH will be issued to the pipeline. WPH will then be updated with the value in the “Next” entry of the issued warp, and CNT will be decremented by the number of threads in the issued warp. If the Majority Entry runs out of warps ( $CNT = 0$ ), the root entry of the Max-Heap will be popped and will become the Majority Entry. If the Max-Heap is in the process of rebalancing itself after a warp insertion, no warp will be issued until the Max-Heap is eventually balanced. The Max-Heap balances itself over multiple cycles according to the algorithm described by Cormen et al. [2001] (bandwidth constrained as in Table V). The MHeap LUT is updated in parallel to any swap operation, and again, before the Max-Heap is rebalanced, none of its entries can be updated by warps arriving at the scheduler.

**4.4.3 Complexity.** As the Max-Heap is a tree structure, every warp insertion only requires up to  $\log_2(N)$  swaps to rebalance for a N-entry max-heap. Experimentally, we find that usually fewer swaps are required. We find in Section 6.3 that a design with both the Max-Heap and the MHeap LUT each having two read ports and two write ports is sufficient to keep the Max-Heap balanced in the common case (handling two warp insertions from each part of the incoming warp every scheduler cycle<sup>12</sup>). This is assuming that both structures are clocked as fast as the SIMD pipeline, so that a total of eight reads and eight writes can be performed in every scheduler cycle.<sup>12</sup>

When the Max-Heap is rebalancing, updates from incoming warps are buffered in a queue and will be performed in subsequent cycles. This does not disrupt the scheduler as long as the updates are done before a new Majority PC is needed. If this queue is full, the scheduler stops accepting warps, which stalls the pipeline for a finite duration until the Max-Heap is rebalanced again and is free to process the next update from the queue. We find that a four-entry queue is enough to avoid stalling the pipeline for all of our benchmarks. If the number of PCs in flight exceeds the Max-Heap capacity, a spilling mechanism as described in Section 4.5 is used. However, we observe in Section 7 that with a 64-entry Max-Heap, the need for spilling never arose for our benchmarks.

---

<sup>12</sup>A scheduler cycle is equivalent to several pipeline cycles, because each warp can take several cycles (e.g., 4 for a warp size of 32) to execute in the pipeline (see Section 2.4), so the scheduler may operate at a slower frequency.

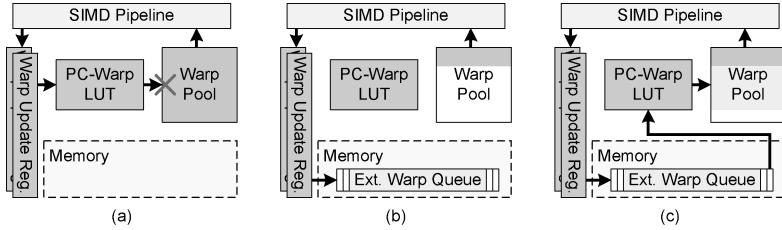


Fig. 12. Warp spilling mechanism to handle excessively diverging code.

**4.4.4 Applicability to Other Scheduling Policies.** The previously described hardware implementation of the Majority scheduling policy can be modified to implement other policies. DMin can be implemented by using a Min-Heap in place of the Max-Heap. The priority criterion can be changed to the PC of the group to implement DPC. On the other hand, with DPdPri, the priority of a warp changes when incoming threads with higher priority are assigned to the warp. This volatility of warp priorities renders the use of the Max-Heap impractical as it will cause the Max-Heap to rebalance frequently. The wide range of priorities also makes it hard to group the warps into a small number of entries in the Max-Heap. Note that DTime can be implemented with a circular FIFO.

#### 4.5 Handling Excessively Diverging Code

While our programming model does allow each thread to diverge to its unique execution path, sizing the warp pool for this unlikely case is wasteful, as most entries would be unused in the common case (as shown in Section 6.8). With a smaller warp pool, dynamic warp formation can still correctly execute code that results in excessively diverging code by spilling entries to memory when the warp pool overflows.

When the warp pool overflows, the scheduler starts writing incoming warps (with threads grouped according to their next PC) directly to an external warp queue in memory, bypassing the PC-warp LUT and scheduling logic (as shown in Figure 12(a) and (b)). Warps already in the warp pool are issued until the warp pool drains. As these warps are drained from the warp pool, it is repopulated by groups of warps loaded from the external queue (as shown in Figure 12(c)). Warps from the external queue arrive at the scheduler and can be merged into existing warps inside the warp pool via the PC-warp LUT, using the same mechanism described in Section 4.2. Late arrival of warps due to memory latency does not stall the pipeline, but the warp pool may be drained to empty in the meantime, taking away opportunities for warps to be merged. This creates a FIFO warp update queue with virtually unlimited capacity. The external warp queue can be managed as a circular buffer with three hardware registers: an offset pointer to the circular buffer and two pointers to head and tail of the queue tracking where to spill and fill warps.

We leave a more-detailed exploration and performance evaluation of this spilling mechanism as future work.

Table I. Hardware Configuration

Shader Core Pipeline Frequency	650 MHz
# Shader Cores	16
SIMD Warp Size	32 (issued over 4 clocks)
SIMD Pipeline Width	8
# Threads per Shader Core	768
# Memory Modules	8
DRAM Control Bus Frequency	650 MHz
GDDR3 Memory Timing	$t_{CL}=9, t_{RP}=13, t_{RC}=34$ $t_{RAS}=21, t_{RCD}=12, t_{RRD}=8$
Bandwidth per Memory Module	8Byte/Cycle (5.2GB/s)
Memory Controller	out of order
Data Cache Size (per core)	512KB 8-way set assoc.
# Data Cache Banks (per core)	16
Data Cache Hit Latency	10 cycle latency (pipelined 1 access/thread/cycle)
Default Warp Scheduling Policy	Majority
PC-Warp LUT	64 entries, 4-way set assoc.
MHeap LUT	128 entries, 8-way set assoc.
Max-Heap	64 entries

## 5. METHODOLOGY

While simulators for contemporary GPU architectures exist [Sheaffer et al. 2004; del Barrio et al. 2006], none of them we are aware of currently model the general-purpose GPU computing architecture described in this article. Therefore, we developed a novel simulator, *GPGPU-Sim*, to model various aspects of the massively parallel architecture used in modern GPUs with highly programmable pipelines. GPGPU-Sim was constructed “from the ground up” starting from the instruction set simulator (sim-safe) of SimpleScalar version 3.0d [Burger and Austin 1997]. We developed our cycle-accurate GPU performance simulator, modeling the microarchitecture described in Section 2.2, around the SimpleScalar PISA instruction set architecture and then interfaced it with sim-outorder, which only provides timing for code running on the CPU in Figure 2. For the purpose of this article, the PISA instruction set architecture is a good representation of PTX, the virtual ISA used in the CUDA programming model [NVIDIA Corp. 2007b]. While lacking multiply-add, transcendental, and texture instructions, PISA features control flow instructions similar to PTX. PISA lacks support for predication; however, on our hardware model, we believe PDOM captures much of the benefits of predication. Table I shows our baseline configuration.

The SimpleScalar out-of-order core waits for the GPU when it reaches a parallel section. After GPU simulation of the compute kernel is completed, program control is returned to the out-of-order core. This repeats until the benchmark finishes.

The benchmark applications used for this study were selected from SPEC CPU2006 [Standard Performance Evaluation Corporation], SPLASH2 [Woo et al. 1995], and CUDA [NVIDIA Corp.]. Each benchmark was manually modified to extract and annotate the computing kernels, which is a time-consuming task, limiting the number of benchmarks we could consider. The

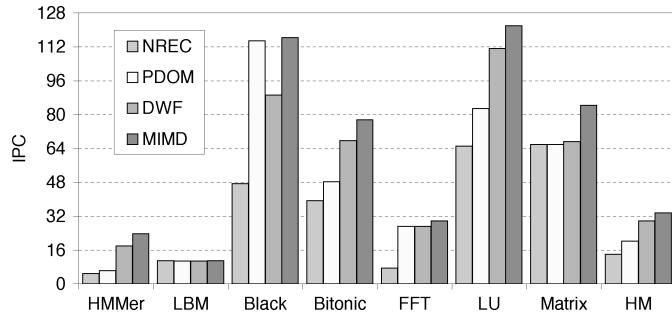


Fig. 13. Performance comparison of NREC, PDOM, and DWF versus MIMD.

programming model we assume is similar to that of CUDA [NVIDIA Corp. 2007a]. In our detailed simulation, a computing kernel is invoked by a spawn instruction, which signals the out-of-order core to launch a predetermined number of threads for parallel execution on the GPU simulator. If the number of threads to be executed exceeds the capacity of the hardware configuration, the software layer is responsible for organizing threads into *blocks* in our model. Threads within a block are assigned to a single shader core, and all of them have to finish before the shader core can begin with a new block.

## 6. EXPERIMENTAL RESULTS

First, we consider dynamic warp formation and scheduling modeling the implementation described in Section 4.2 using the Majority scheduling policy implemented with the Max-Heap modeled, as described in Section 4.4. The hardware is sized as in Table V and employs the thread swizzling mechanism described in Section 4.1. This implementation uses the lane aware scheduling described in Sections 4.1 and 4.2. We also model bank conflicts at the data cache.

Figure 13 shows the performance of the different branch-handling mechanisms discussed in this article and compares them to a MIMD pipeline with the same peak IPC capability. Here, we use the detailed simulation model described in Section 5 including simulation of memory access latencies. On average (HM in Figure 13), PDOM (reconvergence at the immediate postdominator described in Section 3.2) achieves a speedup of 45% versus not reconverging (NREC). Dynamic warp formation (DWF) achieves a further speedup of 47% using the Majority scheduling policy. The average DWF and MIMD performance differs by only 11%. We believe the 47% speedup of DWF versus PDOM justifies the estimated 8% area cost (Section 7) of incorporating dynamic warp formation and scheduling into future GPUs. We note that this magnitude of speedup could not be obtained by simply spending this additional area on extra shader cores.

For benchmarks with little diverging control flow, such as FFT and Matrix, DWF performs as well as PDOM, while MIMD outperforms both of them with its “free running” nature. The significant slowdown of Black-Scholes (Black) with DWF is a phenomenon exposing a weakness of our default Majority scheduling policy. This will be examined in detail in Section 6.2.

Table II. Memory Bandwidth Utilization

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	57.47%	94.71%	8.18%	50.94%	75.43%	0.84%	84.02%
DWF	63.61%	95.04%	6.47%	71.02%	74.89%	1.47%	63.71%
MIMD	64.42%	95.07%	8.30%	81.21%	81.49%	1.58%	99.23%

Table III. Cache Miss Rates Pending Hits<sup>13</sup> Classified as a Miss

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.37%	14.15%	1.52%	21.25%	15.52%	0.05%	7.88%
DWF	5.05%	14.56%	1.50%	30.67%	14.68%	0.06%	9.71%
MIMD	3.74%	15.34%	1.17%	30.67%	13.79%	0.05%	7.30%

Table IV. Cache Miss Rates Without Pending Hits<sup>13</sup>

	HMMer	LBM	Black	Bitonic	FFT	LU	Matrix
PDOM	13.21%	13.55%	0.21%	3.86%	11.77%	0.03%	5.72%
DWF	5.03%	13.57%	0.21%	3.84%	12.53%	0.04%	4.23%
MIMD	3.73%	13.36%	0.21%	3.83%	12.06%	0.04%	5.26%

For most of the benchmarks with significant diverging control flow (HMMer, LBM, Bitonic, LU), MIMD performs the best, and DWF achieves a significant speedup over PDOM. Among them, DWF achieves a speedup for Bitonic and LU purely from better branch-divergence handling, while for HMMer, DWF achieves a speedup in part due to better cache locality as well, as observed from Table III. While LBM also has significant diverging control flow, it is memory bandwidth limited, as shown in Table II and hence sees little gain from DWF. Although the cache miss rate of Bitonic is higher than that of LBM, its much higher pending hit<sup>13</sup> rate significantly lowers the memory bandwidth requirement of this benchmark (see Table IV).

### 6.1 Effects of Scheduling Policies

In this section and in Section 6.2, we evaluate scheduling policies assuming (a) an idealized scheduler able to sort warps in a single cycle to account for changing priorities (i.e., not using the Max-Heap implementation in Figure 11), (b) no restrictions on the home lane of threads grouped into a warp, (c) the multiported register file shown in Figure 9(b) with an unlimited number of ports (i.e., ignoring the effects of lane conflicts), and (d) a realistic memory subsystem.

Figure 14 compares all the warp scheduling policies described in Section 4.3 to show the potential of each policy. Overall, the default Majority (DMaj) policy performs the best, achieving an average speedup of 66.0%, but in some cases, its performance is not as good as the PC policy (DPC) or PDOM Priority (DPdPri) policy.

To provide additional insight into the differences between the scheduling policies, Figure 15 shows the active thread count distribution of warp issued

<sup>13</sup>A pending hit occurs when a memory access misses the cache, but can be merged with one of the in-flight memory requests already sent to the memory subsystem.

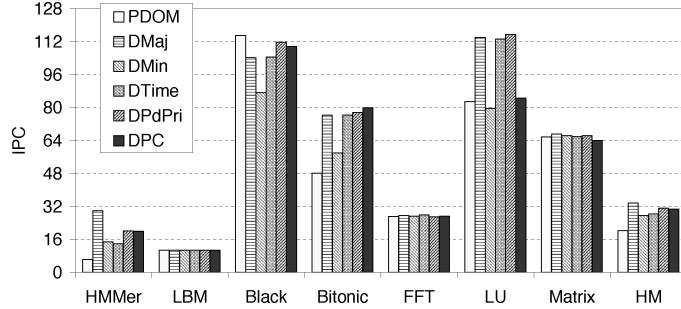


Fig. 14. Comparison of warp scheduling policies. The impact of lane conflicts is ignored.

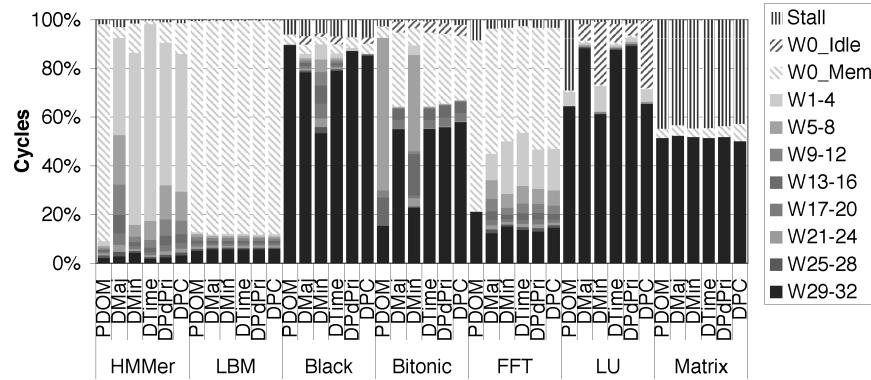


Fig. 15. Active thread count distribution.

each cycle for each policy. Each bar is divided into segments labeled W0, W1-4, ... W29-32, which indicate the fraction of total execution cycles the scheduler issued a warp with 0, (1 to 4), ... (29 to 32) scalar threads. “Stall” indicates a stall due to writeback contention with the memory system (see Figure 3(a)). For policies that do well (DMaj, DPdPri, DPC), we see a decrease in the number of low-occupancy warps relative to those policies, which do poorly (DMin, DTime). Cycles with no scalar threads executed (W0) are classified into “Mem” (W0\_Mem) and “Idle” (W0\_Idle). W0\_Mem denotes the cycles when all threads are waiting for data from global memory. W0\_Idle denotes that all warps within a shader core have already been issued to the pipeline and are not yet ready for issue. These “idle” cycles occur because shader cores execute threads in blocks. In our current simulation, the shader cores have to wait for all threads in a block to complete before moving onto a new block. The active thread count distribution reveals the reasons for DMaj’s poor performance on Black-Scholes to be “Idle” and less than full warps cycles, which can be mostly eliminated by DPdPri. The data also suggest that it may be possible to further improve dynamic warp formation by exploring scheduling policies beyond those proposed in this article.

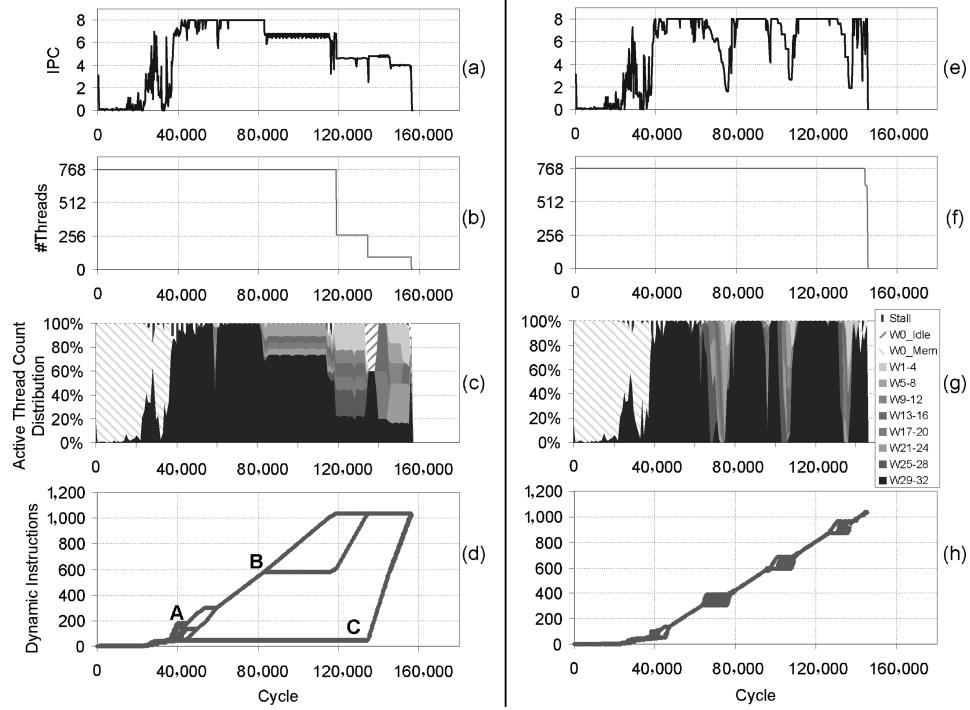


Fig. 16. Dynamic behavior of Black-Scholes using DWF with Majority scheduling policy (a–d) and PDOM Priority scheduling policy (e–h) in a single shader core (warp size of 32 issued over four cycles resulting in max IPC of 8). The active thread count distribution time series in (c) and (g) uses the same classification as in Figure 15.

## 6.2 Detailed Analysis of Majority Scheduling Policy Performance

The significant slowdown of Black-Scholes (Black) results from several phenomenon. First, in this work, we restrict a shader core to execute a single block. Second, we use software subroutines for transcendentals. Third, a quirk in our default Majority scheduling policy (DMaj) can lead some of the threads in a shader core to suffer from starvation. We explore this latter phenomenon in this section. Figure 16(a–d) shows the runtime behavior (IPC and incomplete thread count (#Thread)<sup>14</sup>) of a single shader core using dynamic warp formation with the Majority scheduling policy.

With DMaj, threads having different control-flow behavior from the rest of the threads can be starved during execution. Black-Scholes has several rarely executed, short basic blocks that suffer from this effect, leaving behind several groups of minority threads.<sup>15</sup> When these minority threads finally execute after the majority of threads have finished, they form incomplete warps and the number of warps formed are insufficient to fill up the pipeline (each thread

<sup>14</sup>A thread is counted as incomplete if it has not reached the end of the kernel call on the cycle in question.

<sup>15</sup>We say a thread is a minority thread if its PC value is consistently given a low priority by the Majority scheduling policy throughout its execution.

is only allowed to have one instruction executing in the pipeline, as described in Section 2.3). This behavior is illustrated in Figure 16(d), which shows the dynamic instruction count of each thread versus time. After several branches diverge at A and B, groups of threads are starved (indicated by the stagnant dynamic instruction count in Figure 16(d)), and only resume after C when the majority groups of threads have finished their execution (indicated by the lower thread count after cycle 120,000 in Figure 16(b)). This is responsible for the low IPC of DWF after cycle 120,000 in Figure 16(a).

Meanwhile, although the majority of threads are proceeding ahead after branch divergences at A and B, the pipeline is not completely utilized (indicated by the IPC < 8 from cycle 80,000 to 120,000 in Figure 16(a)) due to the existence of incomplete warps (see the warp size distribution time series in Figure 16(c)). These incomplete warps are formed because the number of threads taking the same execution path is not a multiple of the SIMD width. They could have been combined with the minority threads after the divergence to minimize this performance penalty, but this does not happen when the minority threads are starved by DMaj.

Figure 16(e–h) shows how both of these problems can be mitigated by having a different scheduling policy—PDOM Priority (DPdPri). The dynamic instruction count in Figure 16(h) shows that any thread starvation due to divergence is quickly corrected by the policy, and all the threads have a similar rate of progress. The IPC stays at 8 for most of the execution (see Figure 16(e)), except when DPdPri is giving higher priorities to the incomplete warps inside the diverged execution paths (visible in Figure 16(g,h)) so that they can pass through the diverged execution paths quickly to form complete warps at the end of the diverged paths. Overall, threads in a block finish uniformly as shown in Figure 16(f), indicating that starvation does not happen with DPdPri.

To slowdown the threads that have executed further down a program (so that they can wait for threads left behind in hope of possible reconvergence), DPdPri gives higher priority to threads with lower counts of encountered post-dominators. This is effective on benchmarks with simple control flow, such as Black-Scholes. With nested control flow, however, threads that have executed through more divergent branches will have encountered more postdominators and appear to the scheduler, as if they have progressed further down the program than they actually have. A simple example in which DPdPri is ineffective is a loop with an if-branch inside. Some threads may have exited the loop early and are first given lower priority as they hit the postdominator of the loop’s backward-branch. Threads remaining in the loop would continually encounter the postdominator of the if-branch and eventually their postdominator count would exceed the early-exit threads. This gives the early-exit threads higher priority for execution, leaving behind the threads inside the loop.

This deficiency of DPdPri with complex control flow is exposed by HMMer. In Section 6.4, we evaluate the performance of DPdPri with lane aware scheduling (LAS) and show that the performance of HMMer is significantly lower than that of DMaj’s (12 IPC versus 24 IPC for DMaj). This contributes to a lower average speedup of DPdPri over PDOM (31% versus DMaj’s 47%). Despite this deficiency, DPdPri’s effect on BlackScholes demonstrates the potential of using

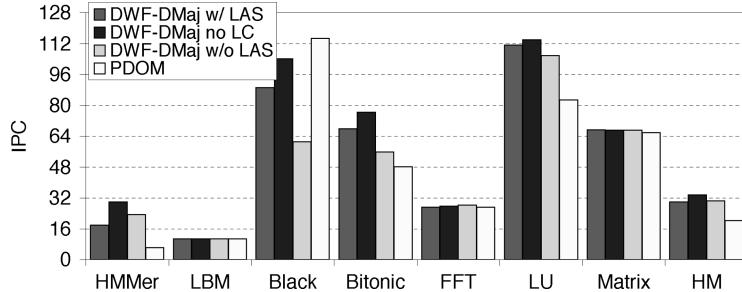


Fig. 17. Performance of dynamic warp formation evaluating the impact of lane aware scheduling and accounting for lane conflicts and scheduler implementation details using Majority scheduling policy. LAS=lane aware scheduling; LC=lane conflict.

static analysis to generate heuristics for improved scheduling. Further investigation of the use of static analysis to aid scheduling for dynamic warp formation is left as future work.

### 6.3 Effect of Limited Resources in Max-Heap

We have evaluated the performance increase achievable by dynamic warp formation with an infinite amount of hardware resources (infinitely ported and unlimited entries for MHeap LUT and Max-Heap) given to the DMaj scheduler. This unbounded version of DMaj has a speedup of 6.1% over its resource-limit counterpart, and is 56.3% faster than PDOM, on average. This speedup with no resource limit comes completely from HMMer (24 IPC versus 18 IPC with limited resources), which is both a control-flow and data-flow intensive benchmark. These properties of HMMer result in a large number of in-flight PC values among threads as a memory access within a warp following a branch divergence can introduce a variable slowdown among the threads in a warp ranging from tens to hundreds of cycles. The large number of in-flight PC values this causes in turn requires a large Max-Heap, which takes more swaps to rebalance every scheduler cycle and introduces stalls when the limited bandwidth resources are exhausted. However, with other benchmarks the resource-limited version of DMaj achieves similar performance.

### 6.4 Effect of Lane Aware Scheduling

To reduce the register file design complexity of DWF, we have chosen to use the organization in Figure 9(d), which necessitated the use of lane aware scheduling (LAS) discussed in Sections 4.1 and 4.2. The data in Figure 17 compares the detailed scheduler hardware model we have considered so far (DWF-DMaj w/ LAS) with an idealized version of dynamic warp formation ignoring the impact of lane conflict and hardware resources (DWF-DMaj no LC). This figure shows the impact on the performance of LAS and, for comparison, also shows the impact when not using LAS but assuming the register file organization in Figure 9(c) and modeling register bank conflicts when multiple threads from the same “home” lane are grouped into a single warp (DWF-DMaj w/o LAS). While the idealized version of DWF unconstrained by lane conflict (DWF-DMaj

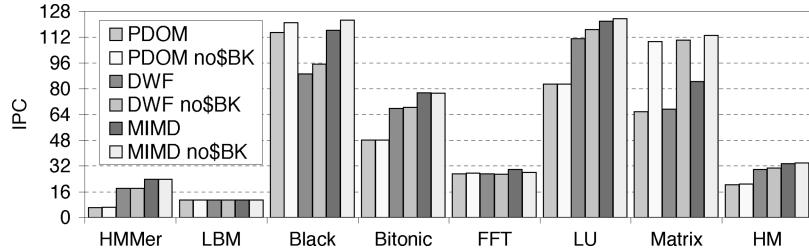


Fig. 18. Performance of PDOM, DWF, and MIMD with cache bank conflict. no\$Bk=ignoring cache bank conflict.

no LC), which assumes hardware similar to Figure 9(b), is, on average, 66% faster than PDOM, the realistic implementation of DWF with LAS we have considered so far (DWF-DMaj w/ LAS) is able to achieve 89% of the idealized DWF's performance (DWF-DMaj no LC).

We also evaluated the performance of the PDOM Priority policy (DPdPri) with LAS and found performance for Black-Scholes improves (99 IPC versus 89 IPC for DMaj), while that of HMMer is reduced (12 IPC versus 24 IPC for DMaj) with an average speedup of 31% over PDOM. Overall, DMaj is the best scheduling policy for the realistic implementation by a significant margin (average of 47% improved with DMaj versus 31% with DPdPri).

### 6.5 Effect of Data Cache Bank Conflicts

Our baseline architecture model assumes a multiported data cache that is implemented using multiple banks of smaller caches. Parallel accesses from a warp to different lines in the same bank creates cache bank conflicts, which can only be resolved by serializing the accesses and stalling the SIMD pipeline.

With a multibanked data cache, a potential performance penalty for dynamic warp formation is that the process of grouping threads into new warps dynamically may introduce extra cache bank conflicts. We have already taken this performance penalty into account by modeling cache bank conflicts in our earlier simulations. However, to evaluate the effect of cache bank conflicts on different branch handling mechanisms, we rerun the simulations with an idealized cache allowing threads within a warp to access any arbitrary lines within the cache in parallel.

The data in Figure 18 compares the performance of PDOM, DWF, and MIMD ignoring cache bank conflicts. Cache bank conflicts have greater effect on benchmarks that are less constrained by the memory subsystem (Black-Scholes, LU, and Matrix). On average, performance of these benchmarks increases by a similar amount with an ideal multiported cache regardless of the branch handling mechanism in place. Overall, the data shows the benefit of DWF does not diminish due to additional cache bank conflicts introduced by regrouping threads into new warps.

### 6.6 Sensitivity to SIMD Warp Size

While DWF with the DMaj scheduling policy provides a significant speedup over PDOM with our default hardware configuration, we can gain further

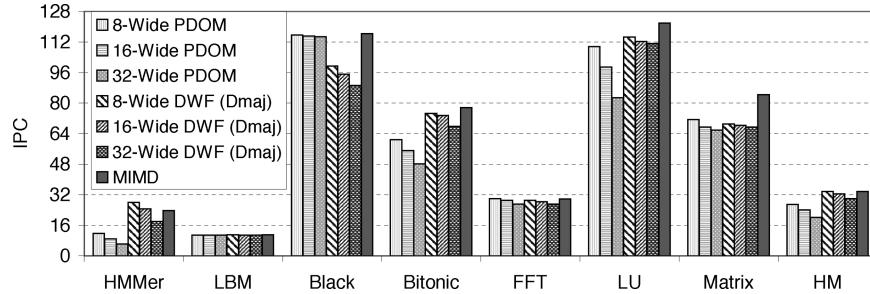


Fig. 19. Performance comparison of PDOM, DWF, and MIMD with a realistic memory subsystem as SIMD warp size increases. The theoretical peak IPC remains constant for all configurations.

intuition into its effectiveness in handling branch divergence as SIMD warp size increases. Figure 19 shows the performance of each mechanism for three configurations with increasing warp size from 8 to 32. Notice that the width of the SIMD pipeline is not changed, a decreased warp size only translates to a shorter execution latency for each warp. As described in Section 2.4, a warp with 32 threads takes 4 cycles to execute on a 8-wide SIMD pipeline, whereas a warp with 8 threads can be executed in a single cycle on the same pipeline.

None of the benchmarks benefit from SIMD warp size increases. This is expected as increasing warp size in an area constrained fashion, as described earlier, does not increase the peak throughput of the architecture while it constraints control-flow and thread-scheduling flexibility. The benefit of increasing SIMD warp size is to improve computational density by relaxing the scheduler's latency requirement, reducing the number of warps to schedule (simplifying scheduler hardware) and lowering instruction cache bandwidth requirements.

Overall, as SIMD warp size increases from 8 to 32, the average performance of PDOM decreases by 24.9%, while the overall performance of DWF decreases by 11.2%. Most of the slowdowns experienced by PDOM as SIMD warp size is increased are due to the control flow intensive benchmarks (HMMer, Bitonic, and LU), while performance is better with DWF. This trend shows that branch divergence becomes a more serious performance bottleneck in control-flow intensive applications as SIMD warp size is increased to improve computational density, but a significant portion of this performance loss can be regained using dynamic warp formation and scheduling.

Notice that the performance of 8-wide PDOM provides an upper bound on the potential gain of a variable rate scheduler for PDOM with a warp size that is a multiple of the SIMD pipeline width. Such a variable rate scheduler might try to skip cycles when no active threads are issued for faster reissue rate. However, this is still 10% slower than DWF with DMaj.

### 6.7 Sensitivity to Thread Pool Size

Our baseline architecture model assumes that each shader core can accommodate up to 768 threads (24 warps), and that the DWF scheduler is free to combine any of these threads into a new warps. However, resource limits, such as limited register capacity per shader core, may restrict the thread capacity

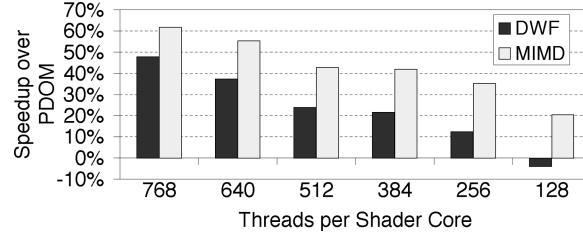


Fig. 20. speedup of DWF and MIMD over PDOM as number of threads per shader core decreases.

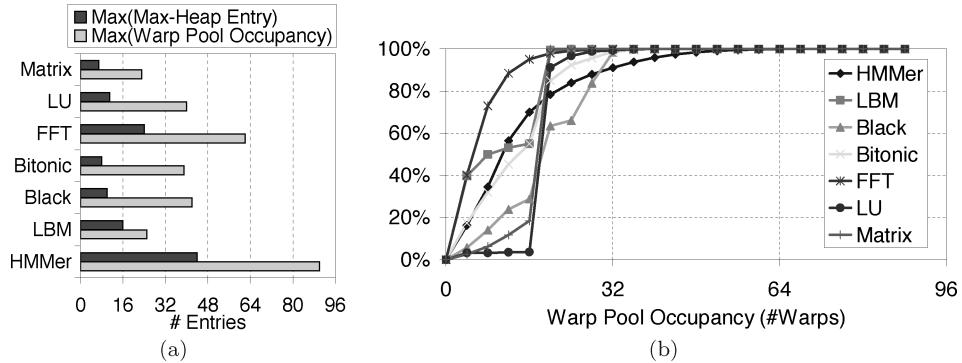


Fig. 21. Warp pool occupancy and Max-Heap size. (a) Maximum #entries. (b) Cumulative distribution of warp pool occupancy.

of each shader core to a fraction of its design limit. Under these circumstances, the DWF scheduler has fewer threads to recombine for new warps, which may lower the performance benefit of DWF.

Figure 20 shows the speedup of DWF (with DMaj and LAS) and MIMD over PDOM with decreasing thread pool size per shader core. While DWF is slower than PDOM by 4% with 128 threads, it shows a speedup of 13% at 256 threads. The speedup increases as the thread pool grows, eventually to 47% with 768 threads. Most of the degraded performance of DWF with fewer threads comes from HMMer (not shown). It is 2.95× faster than PDOM with 768 threads, while with 128 threads, it is 25% slower. The performance gap between DWF and PDOM for Black-Scholes widens with fewer threads (from 23% slower for DWF with 768 threads to 41% slower for DWF with 128 threads—not shown).

### 6.8 Warp Pool Occupancy and Max Heap Size

Figure 21(a) shows the maximum warp pool occupancy (the number of warps inside the warp pool) and the maximum dynamic size of the Max-Heap for each benchmark. As shown in the table, all of the benchmarks, including the control flow and memory intensive HMMer, use < 1/6 of a warp pool sized to the worst-case requirement. This suggests the possibility of using a warp pool with 128 entries (1/6 of the 768-entry warp pool designed for the absolute worst case) without causing any performance penalty. The same argument can be used to

argue that 64 entries is sufficient for the Max-Heap used in implementing the Majority scheduling policy.

Furthermore, Figure 21(b) shows the cumulative distribution of warp pool occupancy for each benchmark in this article. This distribution suggests that it may be feasible to shrink the warp pool to 64 entries, and handle <0.15% warp pool overflow (or 10% if we reduce the warp pool to 32 entries) by spilling warp pool entries to the global memory (as described in Section 4.5). As our simulator does not currently model this spilling mechanism, we assume structures having enough capacity to avoid spilling in the area estimation in Section 7.

## 7. AREA ESTIMATION

The total area cost for dynamic warp formation and scheduling is the amount of hardware added for the logic in Figure 10 plus the overhead needed for having an independent decoder for each register file bank. We have estimated the area of the five major parts of the hardware implementation of dynamic warp formation and scheduling with CACTI 4.2 [Tarjan et al. 2006]: warp update registers, PC-warp LUT, warp pool, warp allocator, and scheduling logic. Table V lists the storage sizing and implementation details of these structures used in our performance simulation in Section 6 along with their area estimates. We use our baseline configuration (32-wide SIMD with 768 threads) listed in Table I to estimate the size of the PC-warp LUT and MHeap LUT. Both are set-associative cache-like structures (4- and 8-way, respectively) with two read ports and two write ports capable of two warp lookups in parallel to handle requests from the two diverged parts of a single incoming warp. Based on the maximum occupancy data in Section 6.8 and Figure 21, we use a 128-entry warp pool and a 64-entry Max-Heap. A memory array with two read ports and two write ports is sufficient for the Max-Heap. Instead of being implemented with a multiported memory array, the warp pool can be separated into banks with only one write port, with each bank storing the TIDs of threads executing in the same SIMD lane. This works because every incoming warp (no matter how warps have diverged) has at most one thread would be written to each bank per cycle. The PC value and any scheduler specific data of each warp are stored in a separate bank from the TIDs with two write ports to support two warp creations in parallel. Overall, we have estimated the area of the dynamic warp scheduler in 90nm process technology to be 1.635mm<sup>2</sup> per core (last row, column on right in Table V).

To evaluate the overhead of having individual decoders per register file bank for dynamic warp formation and scheduling, as described in Section 4.1, we first need to estimate the size of the register file. The SRAM model of CACTI 4.2 [Tarjan et al. 2006] estimates a register file with 8,192 32-bit registers and a single decoder reading a row of eight 32-bit registers to be 3.8628mm<sup>2</sup> (we have adjusted the line size to 512 bytes for the minimum area). On the other hand, since the SRAM model of CACTI 4.2 does not directly estimate the area impact of banking, we first estimate the area of a single register file bank with 1024 32-bit registers. The overall register file is then estimated to require 0.573mm<sup>2</sup> × 8 = 4.585mm<sup>2</sup>. Note that for the banked register file, we have divided the

Table V. Area Estimation for Dynamic Warp Formation and Scheduling. RP=read port,  
WP=write port

Structure	# Entries	Entry Content	Struct. Size (bits)	Implementation	Area ( $mm^2$ )
Warp Update Register	2	TID (10-bit) $\times$ 32 PC (24-bit), REQ (32-bit)	752	Registers (No Decoder)	0.0080
PC-Warp LUT	64	PC (24-bit) OCC (32-bit) IDX (7-bit)	4032	4-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.2633
Warp Pool	128	TID (10-bit) $\times$ 32 PC (24-bit) Scheduler Data (10-bit)	44928	Memory Array (33 Banks) (1 RP, 1 WP)	0.6194
Warp Allocator	128	IDX (7-bit)	896	Memory Array	0.0342
Mheap LUT	128	PC (24-bit) IDX (7-bit)	4992	8-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.4945
Max-Heap	64	PC (24-bit), CNT (10-bit) WPH (8-bit), WPT (8-bit) LUT-IDX (8-bit)	3712	Memory Array (2 RP, 2 WP)	0.2159
<b>Total</b>			59312		<b>1.6353</b>

512-byte line of the original single decoder design into eight 64-byte lines, one in each bank. Notice that both register file configurations have two read ports and one write port. This method may under-estimate the area requirement for adding a decoder to every register file bank, as it may not entirely capture all wiring complexity. However, the overhead we estimate in this way is already larger than the 11% overhead estimate by Jayasena et al. [2004] with CACTI and a custom floorplan for implementing a stream register file with indexed access. (Note: 11% of  $3.863mm^2$  is only  $0.425mm^2$ , which is smaller than our  $4.585mm^2 - 3.863mm^2 = 0.722mm^2$  estimate.) Without an detailed layout of an actual GPU register file, the  $0.722mm^2$  overhead is our best estimate.

Combining the two estimates described earlier, the overall area consumption of dynamic warp formation with DMaj scheduling policy for each core is  $2.357mm^2$ . With 16 cores per chip, as per our initial configuration, this becomes  $37.72mm^2$ , which is 8% of the total area of the GeForce 8800GTX ( $470mm^2$ ) [Lindholm et al. 2008].

## 8. LIMITATIONS OF DYNAMIC WARP FORMATION

This section discusses some challenges with adopting dynamic warp formation (DWF) in existing GPU architectures.

*Synchronous Execution of Warps.* While NVIDIA’s CUDA programming model, which our compute model closely resembles, provides a MIMD thread model, there exist a small number of CUDA programs optimized with the assumption that the threads in a statically formed warp are executed synchronously. A example of such an optimization is shown in Figure 22, which illustrates the final steps of a parallel reduction algorithm [Harris 2007]. Each step needs to be executed synchronously, but the placement of barriers between each step can be avoided as the synchronization is done implicitly within a statically formed warp in current GPU microarchitectures.

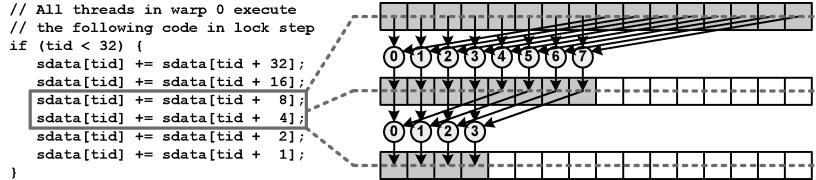


Fig. 22. Parallel reduction assuming synchronous execution of warps. Numbers inside the circles denotes the threads performing the reduction. Only two steps are illustrated here.

With DWF, threads in a statically formed warp may be dynamically reassigned to a different warps. This behavior breaks the implicit synchronization assumed by the optimization and this may render it functionally incorrect. One solution would be to require such behavior be expressed explicitly at the language level by expressing it using partial barriers between each line of code in the example (they are partial in the sense that only those threads in a warp participate in the barrier, rather than all threads in a block as in the `_syncthreads()` primitive in CUDA). Such partial barriers could potentially be optimized out by a compiler, depending upon the target architecture.

*Memory Coalescing.* A related optimization in CUDA programs is to arrange data access from threads within a warp so that they access contiguous data in memory. Memory coalescing is then achieved by combining these simultaneous memory accesses from multiple threads into fewer accesses [NVIDIA Corp. 2007a]. DWF may disrupt this coalesced access pattern by combining threads from different warps. This generates extra memory accesses, which in turn increases memory traffic. The performance impact from such extra memory traffic is modeled in the simulations we presented earlier (though we did not specifically optimize our benchmarks for memory coalescing).

## 9. RELATED WORK

Most approaches to supporting branches in a traditional SIMD machine have centered around the notion of guarded instructions [Bouknight et al. 1972]. Also known as a predicated or vector masked instructions, a guarded instruction's execution is dependent on a conditional mask controlled by another instruction. A predicated instruction with a conditional mask set to false will not update architecture state. A SIMD instruction with a vector of conditional masks, each controlled by an element in a stream, provides the functionality of a data dependent branch. This approach has been employed in existing GPU architectures to eliminate short branches and potential branch divergences [AMD, Inc. 2006; NVIDIA Corp. 2007a].

Guarded instructions and their variants put constraints on input-dependent loops. Branch divergence may be inevitable, but the period of divergence can be kept short with reconvergence to minimize performance lost due to unfilled SIMD pipelines. A patent filed by Lindholm et al. [2005] describes in detail how threads executing in a SIMD pipeline are serialized to avoid hazards, but does not indicate the use of reconvergence points to recover from such divergence. Levinthal and Porter [1984] described a SIMD branch handling mechanism using a stack of execution mask that is similar to our immediate post-dominator

branch reconvergence mechanism. Unlike our approach that uses control-flow analysis, they mapped high-level language control-flow constructs directly into actions to the stack. The notion of reconvergence based on control-flow analysis in SIMD branch handling was described in a patent by Lorie and Strong [1984]. However, they propose to insert the reconvergence point at the beginning of a branch and not at the immediate postdominator, as proposed in this article.

The notion of dynamically regrouping the scalar SPMD threads comprising a single SIMD task after control-flow divergence of the SPMD threads was described by Cervini [2005] in the context of simultaneous multithreading (SMT) on a general-purpose microprocessor that provides SIMD function units for exploiting subword parallelism. However, his proposal lacks a specific mechanism for grouping the diverged SPMD threads, whereas such an implementation is a main contribution of this article. Clark et al. [2007] introduce *Liquid SIMD* to improve SIMD binary compatibility by translating annotated scalar instructions into SIMD instructions at runtime with specialized hardware. The translated SIMD instructions still suffer performance loss from branch divergence when they execute on a traditional SIMD unit.

Kapasi et al. [2000] introduce *conditional streams*, a code transformation that separates a single kernel with conditional code into multiple kernels and connects them via interkernel buffers to increase the utilization of a SIMD pipeline. A filter and a merger kernel are created for each diverging conditional branch to pack and unpack filtered data using an interprocessor switch. While more efficient than predication, this technique may be limited to architectures with software managed interkernel buffers, such as the stream register file in Imagine [Rixner et al. 1998] and Merrimac [Dally et al. 2003]. In comparison, dynamic warp formation is a hardware mechanism applicable to any multi-threaded SIMD architecture, and its implementation does not require data movement between register lanes.

Krashinsky et al. [2004] propose the vector-thread architecture exposing two instruction fetch mechanism in the ISA: vector-fetch and thread-fetch. Vector-fetch is issued by a control processor to command all virtual processors (VPs) to fetch the same atomic instruction block (AIB) for execution, whereas thread-fetch can be issued by a VP to fetch an AIB to itself alone when execution diverges. This eliminates branch divergence, but instruction bandwidth can become a bottleneck if each VP is fetching a different AIB.

## 10. SUMMARY

In this article, we explore the impact of branch divergence on GPU performance for nongraphics applications. Without any mechanism to handle branch divergence, performance of a GPU's SIMD pipeline degrades significantly. While existing approaches to reconverging control flow at join points such as the immediate postdominator improve performance, we found significant performance improvements can be achieved with our proposed dynamic warp formation and scheduling mechanism. We described and evaluated an implementation of the hardware required for dynamic warp formation and tackled the challenge of enabling correct access to register data as thread warps are dynamically regrouped. We found that with 256 threads per shader core, dynamic warp

formation improves performance by 13%, on average, over a mechanism comparable to existing approaches—reconverging threads at the immediate postdominator. The improvement increases to 47% with 768 threads per shader core. Furthermore, we estimated the area of our proposed hardware changes to be around 8%.

Our experimental results highlight the importance of careful prioritization of warps for scheduling in such massively parallel hardware, even when individual scalar threads are executing the same code in the same program phase. Thus, we believe there is room for future work in this area.

#### ACKNOWLEDGMENTS

We thank the associate editor in charge and the anonymous referees for their helpful comments. We also thank Henry Tran for contributions to our simulation infrastructure.

#### REFERENCES

- AGARWAL, V., HIRSHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, 248–259.
- AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Upper Saddle River, NJ.
- ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th Symposium on Principles of Programming Languages (POPL '83)*. ACM, 177–189.
- AMD, INC. 2006. *ATI CTM Guide*, 1.01 ed. AMD, Inc.
- BASU, A., KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. 2007. Scavenger: A new last level cache architecture with global block priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'07)*. ACM, 421–432.
- BLINN, J. F. 1978. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.* 12, 3, 286–292.
- BOUKNIGHT, W., DENENBERG, S., McINTYRE, D., RANDALL, J., SAMEH, A., AND SLOTNICK, D. 1972. The Illiac IV System. *Proc. IEEE* 60, 4, 369–388.
- BUATOIS, L., CAUMON, G., AND LÉVY, B. 2008. Concurrent number cruncher: A GPU implementation of a general sparse linear solver. *Int. J. Parall. Emerge. Distrib. Syst.*
- BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: Stream computing on graphics hardware. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'04)*. ACM, 777–786.
- BURGER, D. AND AUSTIN, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. <http://www.simplescalar.com>.
- CERVINI, S. 2005. European Patent EP 1531391 A2: System and method for efficiently executing single program multiple data (SPMD) programs.
- CLARK, N., HORMATI, A., YEHIA, S., MAHLKE, S., AND FLAUTNER, K. 2007. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 216–227.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. MIT Press, Cambridge, MA.
- CROW, F. C. 1977. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.* 11, 2, 242–248.
- DALLY, W. J., LABONTE, F., DAS, A., HANRAHAN, P., AHN, J.-H., GUMMARAJU, J., EREZ, M., JAYASENA, N., BUCK, I., KNIGHT, T. J., AND KAPASI, U. J. 2003. Merrimac: Supercomputing with streams. In *Proceedings of Supercomputing*. IEEE.
- DALLY, W. J. AND TOWLES, B. 2004. *Interconnection Networks*. Morgan Kaufmann, San Francisco, CA.

- DEL BARRO, V., GONZALEZ, C., ROCA, J., FERNANDEZ, A., AND ESPASA, R. 2006. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 231–241.
- FOWLER, H., FOWLER, F., AND THOMPSON, D., EDS. 1995. *The Concise Oxford Dictionary* 9th Ed. Oxford University Press.
- FUNG, W. W. L., SHAM, I., YUAN, G., AND AAMODT, T. M. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'07)*. ACM, 407–420.
- HAKURA, Z. S. AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. ACM, 25, 2, 108–120.
- HARRIS, M. 2007. Optimizing parallel reduction in cuda. <http://developer.download.nvidia.com/compute/cuda/1.1/Website/projects/reduction/doc/reduction.pdf>.
- HWU, W.-M., KIRK, D., RYOO, S., RODRIGUES, C., STRATTON, J., AND HUANG, K. 2007. Performance insights on executing non-graphics applications on CUDA on the NVIDIA GeForce 8800 GTX. [http://www.hotchips.org/archives/hc19/2\\_Mon/HC19.02/HC19.02.03.pdf](http://www.hotchips.org/archives/hc19/2_Mon/HC19.02/HC19.02.03.pdf).
- INTEL CORP. 2008. Intel 965 Express Chipset Family and Intel G35 Express Chipset Graphics Controller Programmer's Reference Manual. Intel Corporation.
- IOANNOU, A. AND KATEVENIS, M. G. H. 2007. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.* 15, 2, 450–461.
- JAYASENA, N., EREZ, M., AHN, J. H., AND DALLY, W. J. 2004. Stream register files with indexed access. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA'04)*. IEEE, 60–71.
- KAPASI, U. J., DALLY, W. J., RIXNER, S., MATTSON, P. R., OWENS, J. D., AND KHAILANY, B. 2000. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'03)*. ACM, 159–170.
- KRASHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J., AND ASANOVIC, K. 2004. The vector-thread architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. ACM, 52–63.
- LEVINTHAL, A. AND PORTER, T. 1984. Chap: A SIMD graphics processor. In *Proceedings of SIGGRAPH*. 77–82.
- LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. P. 2001. A user-programmable vertex engine. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'01)*. ACM, 149–158.
- LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *Micro IEEE* 28, 2, 39–55.
- LORIE, R. A. AND STRONG, H. R. 1984. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors.
- LUEBKE, D. AND HUMPHREYS, G. 2007. How GPUs work. *Computer* 40, 2, 96–100.
- MONTRYM, J. AND MORETON, H. 2005. The GeForce 6800. *IEEE Micro* 25, 2, 41–51.
- MOY, S. AND LINDHOLM, E. 2005. US Patent 6,947,047: Method and system for programmable pipelined graphics processing with branching instructions.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmanns.
- NEEDLEMAN, S. B. AND WUNSCH, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequences of tow proteins. *Mol. Biol.* 48, 443–453.
- NVIDIA CORP. CUDA SDK code samples. [http://www.nvidia.com/object/cuda\\_get\\_samples.html](http://www.nvidia.com/object/cuda_get_samples.html).
- NVIDIA CORP. 2007a. *NVIDIA CUDA Programming Guide*, 1.1 ed. NVIDIA Corp.
- NVIDIA CORP. 2007b. *PTX: Parallel Thread Execution ISA*, 1.1 ed. NVIDIA Corp.
- PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *Proceeding of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM, 703–712.
- RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LÓPEZ-LAGUNAS, A., MATTSON, P. R., AND OWENS, J. D. 1998. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st International Symposium on Micro-architecture (MICRO'98)*. ACM, 3–13.

- RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, 128–138.
- ROTBENBERG, E., JACOBSON, Q., AND SMITH, J. E. 1999. A study of control independence in superscalar processors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA'99)*. IEEE, 115–124.
- SHEAFFER, J. W., LUEBKE, D., AND SKADRON, K. 2004. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*. ACM, 85–94.
- SHEBANOW, M. 2007. ECE 498 AL: Programming massively parallel processors (lecture 12). <http://courses.ece.uiuc.edu/ece498/al1/Archive/Spring2007>.
- SHIN, J., HALL, M., AND CHAME, J. 2007. Introducing control flow into vectorized code. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 280–291.
- STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2006 benchmarks. <http://www.spec.org/cpu2006/>.
- TARJAN, D., THOZIYOR, S., AND JOUPPI, N. P. 2006. CACTI 4.0. Tech. rep. HPL-2006-86, Hewlett Packard Laboratories, Palo Alto, CA.
- THISTLE, M. R. AND SMITH, B. J. 1988. A processor architecture for Horizon. In *Proceedings of Super-computing*. IEEE, 35–41.
- THORNTON, J. E. 1964. Parallel operation in the control data 6600. In *AFIPS Proceedings of FJCC*. Vol. 26. 33–40.
- UPSTILL, S. 1990. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, Reading, MA.
- WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, 24–36.
- WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for real-time ray tracing. *ACM Trans. Graph.* 24, 3, 434–444.

Received April 2008; revised November 2008; accepted December 2008