Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow

Wilson W. L. Fung Ivan Sham George Yuan Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
{wwlfung,isham,gyuan,aamodt}@ece.ubc.ca

Abstract

Recent advances in graphics processing units (GPUs) have resulted in massively parallel hardware that is easily programmable and widely available in commodity desktop computer systems. GPUs typically use single-instruction, multiple-data *(SIMD) pipelines to achieve high performance with minimal overhead incurred by control hardware. Scalar threads 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 by hardware. One approach 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. In this paper, we explore mechanisms for more efficient SIMD branch execution on GPUs. We show that a realistic hardware implementation that dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes improves performance by an average of 20.7% for an estimated area increase of 4.7%.*

1. Introduction

As semiconductor process technology continues to scale 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 as power limitations restrict clock frequency scaling [13]. 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 and larger caches. Now attention has (again) shifted towards using 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 [18, 28, 6]. Earlier generations of GPUs consisted of fixed function 3D rendering pipelines. This required new hardware to enable new real-time rendering techniques, which impeded the adoption of new graphics algorithms and thus motivated the introduction of programmability, long available in traditional offline computer animation [35], 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 *shader program* on each input vertex or pixel [18]. Shader programs 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 non-graphics workloads is closely related to that of *stream processors* [29, 11].

The programmability of shader hardware has greatly improved over the past decade, and the shaders of the latest generation GPUs are Turing-complete, opening up exciting new opportunities to speed up "general purpose" (i.e., non-graphics) applications. Based upon experience gained from pioneering efforts to generalize the usage of GPU hardware [28, 6], GPU vendors have introduced new programming models and associated hardware support to further broaden the class of applications that may efficiently use GPU hardware [1, 27].

Even with a general-purpose programming interface, mapping existing applications to the parallel architecture of a GPU is a non-trivial task. Although some applications can achieve speedups of 20 to 50 times over their CPU equivalent [15], other applications, while successfully parallelized on different hardware platforms, show little improvement when mapped to a GPU [4]. One major challenge for contemporary GPU architectures is efficiently handling con-

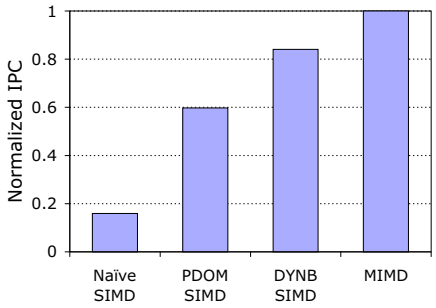

Figure 1. Performance loss due to branching when executing scalar SPMD threads using SIMD hardware (idealized memory system).

trol flow in shader programs [30]. The reason is that, in an effort to improve computation density, modern GPUs typically batch together groups of individual threads running the same shader program, and execute them together in lock step on a *single-instruction, multiple-data* (SIMD) pipeline [19, 21, 22, 20, 27]. Such thread batches are referred to as warps[1] by NVIDIA [27, 30]. This approach has worked well [17] for graphics operations such as texturing [8, 3], which historically have not required branch instructions. However, when shader programs *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 [22, 37] known as branch divergence [19, 30]. 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[2]).

This paper makes the following contributions:

- It establishes that, for the set of non-graphics applications we studied, reconverging control flow of processing elements at the immediate post-dominator [23] of the divergent branch is nearly optimal with respect to oracle information about the future control flow associated with each individual processing element.
- It quantifies the performance gap between the immediate post-dominator 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. Thus, highlighting the importance of finding better branch handling mechanisms.
- It proposes and evaluates a novel mechanism for regrouping processing elements of individual SIMD warps on a cycle-by-cycle basis to greatly improve the efficiency of branch handling.

In particular, for a set of data parallel, non-graphics applications ported to our modern GPU-like SIMD streaming processor architecture, we find the speedup obtained by reconverging the diverging threads within an SIMD warp at the immediate post-dominator of the control flow path obtains a speedup of 93.4% over not reconverging, and dynamically regrouping scalar threads into SIMD warps on a cycle by cycle basis obtains an additional speedup of 20.7% (136.9% speedup versus not reconverging). We estimate the hardware required by this regrouping mechanism adds 4.7% to the total chip area.

The rest of the paper is organized as follows: Section 2 gives an overview of the baseline GPU architecture used in this paper. Section 3 describes the immediate post-dominator control-flow reconvergence mechanism. Section 4 describes our proposed dynamic regrouping mechanism. Section 5 describes the simulation methodology of the proposed GPU architecture. Section 6 describes our experimental results. Section 7 describes related work. Section 8 summarizes this paper and suggests future work.

2. SIMD Stream Processor Architecture

Figure 2 illustrates the baseline architecture used in the rest of this paper. In this figure, each shader core executes multiple parallel threads running the same shader program, with each thread's instructions executed in-order by the hardware. The multiple threads on a given 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. Instructions read their operands in parallel from a highly banked register file. Memory requests access a highly banked data cache and cache misses are forwarded to memory 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. The interconnection network we simulated is a crossbar with a parallel iterative matching allocator [12].

Since our focus in this paper is non-graphics 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 *single-program, multiple-data* (SPMD) software (with possible inter-thread communication) is heavily motivated by the need to balance efficient "general

[1] In the textile industry, the term "warp" refers to "the threads stretched lengthwise in a loom to be crossed by the weft" [14].

[2] *Naïve* and *PDOM SIMD* are described in Section 3 while *DYNB SIMD* is described in Section 4. Benchmarks and microarchitecture are described in Section 5.

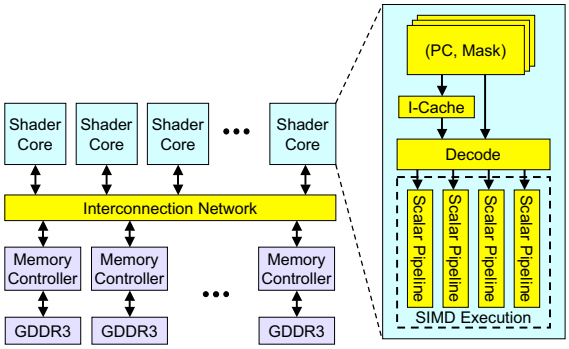

Figure 2. Baseline streaming processor architecture. Blocks labeled 'scalar pipeline' include register read, execute, memory and writeback stages.

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 [30] (shader programs for graphics may very well make increasing use of control flow operations in the future, for example to achieve more realistic lighting effects—increasing the importance of this work).

2.1. Latency Hiding

Since cache hit rates tend to be low for streaming applications, performance would be severely penalized if the pipeline had to stall for every memory request that missed. This is especially true when the latency of memory requests can be several hundred cycles due to the combined effects of contention in the interconnection network and row-activate-precharge overheads at the DRAM data pins. 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 may employ fine-grained multi-threading (i.e., "barrel processing"), where individual threads are interleaved by the fetch unit [34] to proactively hide the potential latency of stalls before they occur. As illustrated by Figure 3, 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 from the pool of "ready" threads and thereby allows other shader threads to proceed while the memory system processes its request. Barrel processing effectively hides the latency of most memory operations since the pipeline is occupied with instructions from other threads while memory operations complete.

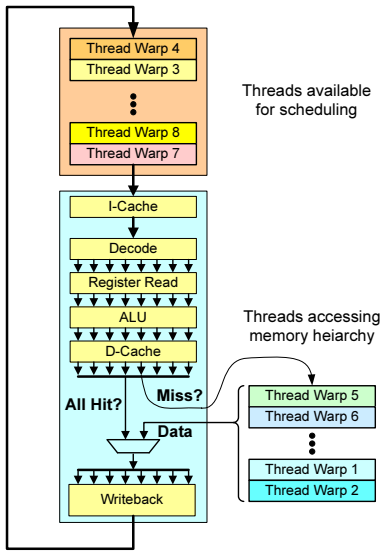

Figure 3. Using barrel processing to hide data memory access latency.

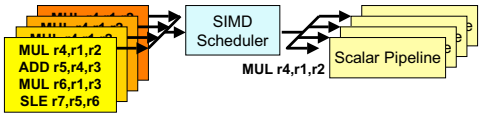

Figure 4. Grouping scalar threads into an SIMD warp.

2.2. SIMD Execution of Scalar Threads

While barrel processing can hide memory latency with relatively simple hardware, a modern GPU must also exploit the explicit parallelism provided by the stream programming model [6, 27] associated with programmable shader hardware to achieve maximum performance at minimum cost. SIMD hardware [5] can efficiently support SPMD program execution provided that individual threads follow similar control flow. Figure 4 illustrates how instructions from multiple shader threads are grouped into a single SIMD warp and scheduled together into multiple scalar pipelines. The multiple scalar pipelines execute in "lock-step" and all data-independent logic may be shared to greatly reduce area relative to a MIMD architecture. A significant source of area savings for such an SIMD pipeline is the simpler instruction cache support required for a given number of scalar threads.

3. SIMD Control Flow Support

To ensure the hardware can be easily programmed for a wide variety of applications, some recent GPU architec-

tures allow individual threads to follow distinct program paths [26, 27]. We note that where it applies, predication [2] is a natural way to support such fine-grained control flow on an SIMD pipeline. However, predication does not eliminate branches due to loops and introduces overhead due to instructions with false predicates.

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 [19]. Lindholm and Moy describe a mechanism for supporting branching using a serialization mode [22]. Finally, Woop et al. describe the use of a hardware stack and masked execution [37].

The effectiveness of an SIMD pipeline is based on the assumption that all threads running the same shader program expose identical control-flow behaviour. While this assumption is true for most existing graphics rendering routines [30], most existing general-purpose applications (and potentially, future graphics rendering routines) tend to have much more diverse control-flow behaviour. When an input-dependent branch leads to different control flow paths for different threads in a warp, a hazard, known as *branch divergence* occurs because an SIMD pipeline cannot execute different instructions in the same cycle. The following sections describe two 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 an SIMD pipeline is to serialize the threads within a warp as soon as the program counters diverge. While this method is easy to understand and implement, it is unacceptable in terms of performance. 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 in Figure 1.

3.2. SIMD Reconvergence

Given the drawback of serialization there must be some mechanism for *reconverging* control flow. The opportunity for such reconvergence is illustrated in Figure 5(a). In this example, a warp of threads diverges after reaching the branch at A. The first three threads encounter a taken branch and go to basic block B (indicated by the bit mask 1110 in Figure 5(a)), while the last thread goes to basic block F (indicated by the bit mask 0001 in Figure 5(a)). The three threads executing basic block B further diverge to basic blocks C and D. However, at basic block E the control

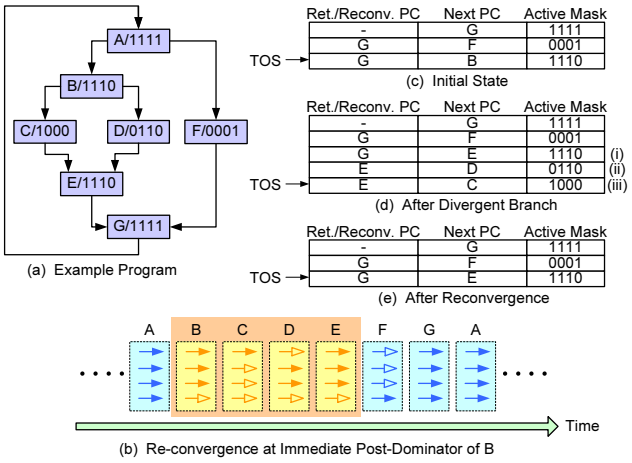

Figure 5. Implementation of immediate post-dominator based reconvergence.

flow paths reach a join point [23]. If the threads that diverged from basic block B to C waited before executing E for the threads that went from basic block B to basic block D, then all three threads can continue execution in-sync 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 in-sync. Figure 5(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 behaviour described above can be achieved using a stack based reconvergence mechanism [37]. In this paper, we use the mechanism shown in Figure 5(c,d,e). Here we show how the stack is updated as the group of three threads in Figure 5(a) that execute B diverge and then reconverge at E. Before the threads execute the diverging branch at B, the state of the stack is as shown in Figure 5(c). When the branch divergence is detected, the stack is modified to the state shown in Figure 5(d). The changes that occur are the following: First, the original top of stack (TOS) at (i) in Figure 5(d) has it's next PC field modified to the instruction address of the reconvergence point E (this could be acquired through an extra target field in the branch instruction). Then, a new entry (ii) is allocated onto the stack and initialized with the reconvergence point address (E) along with a next PC value (D) of the fall through of the branch, and a mask (0110) encoding which processing elements evaluated the branch as "not-taken". Finally, a new entry (iii) is allocated onto the stack with the same reconvergence point address (E), the target address (C) of the branch and the mask (1000) encoding the processing element that evaluated the branch as taken. Note that this mechanism supports "nested" branch hammocks. As pointed out by Woop [37], the same stack can be used to store return ad-

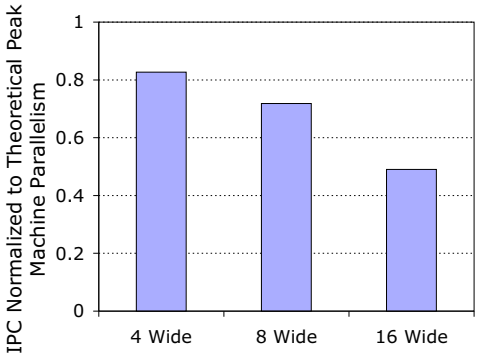

Figure 6. Performance loss for PDOM versus SIMD warp size (idealized memory system).

dresses for function calls.

In this paper we use the *immediate post-dominator* [23] of the diverging branch instruction as the reconvergence point. A *post-dominator* is defined as follows: A basic block X post-dominates basic block Y (written as "X pdom Y") if and only if all paths from Y to the exit node go through X, where a basic block is a piece of code with a single entry and exit point. A basic block X, distinct from Y, *immediately post-dominates* basic block Y if and only if X pdom Y and there is no basic block Z such that X pdom Z and Z pdom Y. Immediate post-dominators are typically found at compile time as part of the control flow analysis necessary for code optimization.

The performance impact of the immediate post-dominator reconvergence technique (labeled PDOM) depends upon the number of processing elements (i.e., "width") of an SIMD warp. Figure 6 shows the harmonic mean IPC of the benchmarks studied in Section 6 normalized to the maximum parallelism possible for 4, 8, and 16 wide SIMD execution assuming 8 shader cores and an ideal memory system in which cache misses have the same latency as cache hits. Execution resource utilization decreased from 82.7% for 4-wide, to 71.8% for 8-wide down to 49% for 16-wide.

In the following section we explore whether immediate post-dominators are the "best" reconvergence points, or whether there might be a benefit to dynamically predicting a reconvergence point past the immediate post-dominator (We show an example where this may be beneficial).

3.3. Reconvergence Point Limit Study

While reconverging at the immediate post-dominator is able to recover much of the performance lost due to branch divergence compared with not reconverging at all, Figure 7 shows a simple example where this reconvergence mechanism is sub-optimal. In this example, threads with even value of `tid` diverge from those with odd values of `tid`

```
void shader_thread(int tid, int *data) {
    for(int i = tid % 2; i < 128; ++i) {
        if(i % 2) {
            data[tid]++;
        }
    }
}
```

Figure 7. Pathological example for which reconvergence at points beyond the immediate post-dominator yields the significant improvement in performance shown on Figure 8. The parameter `tid` is the thread ID.

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 post-dominator 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 about future control flow.

For this limit study, the 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 are determined via repeated applications of the Needleman-Wunsch algorithm [24]. With more than two threads per warp, the optimal alignment is determined by exhaustively searching all possible pair-wise alignments between the threads within a warp. The best reconvergence points are then identified from the optimal alignment.

Figure 8 compares performance of immediate post-dominator reconvergence versus the predicted reconvergence points derived using this method. 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 7 and the benchmarks described in Section 5 (represented by the bar labeled "Real Programs"). While the pathological example experiences a 92% speedup with oracle reconvergence point prediction, the improvement on the real programs we studied is much less (2.6%). Interestingly, one of the benchmarks (bitonic sort) does have similar even/odd thread dependence as our pathological example. However, it also contains frequent barrier synchronizations that ensure loop iterations execute in lock-step.

4. Dynamic Warp Formation & Scheduling

While the post-dominator reconvergence mechanism is able to mitigate performance loss resulting from diverging branches, it does not fully utilize the SIMD pipeline relative to a MIMD architecture with the same peak IPC capability.

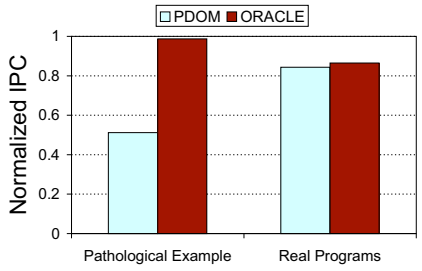

Figure 8. Impact of predicting optimal SIMD branch reconvergence points.

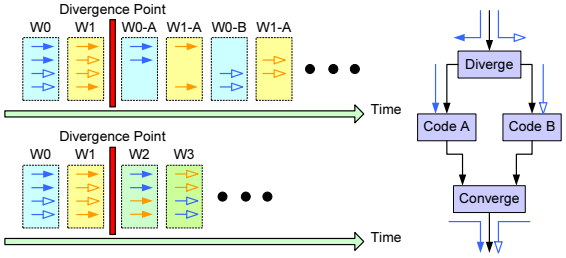

Figure 9. Dynamic warp formation example.

In this section, we describe our proposed hardware mechanism for recovering the lost performance potential of the hardware.

If there is only one thread warp available for scheduling at a time, the performance penalty due to branch divergence is unavoidable since the diverged parts of the warp cannot execute simultaneously on the SIMD hardware in a single cycle. However, with the "barrel processing" technique employed to hide memory access latency, there ought to be more than one thread warp available for scheduling in a shader core on any given cycle. After reaching a diverging branch, each of these warps would be split into two diverged parts: taken and not taken. Ideally, the diverged parts of multiple thread warps branching to the same target can be grouped into complete thread warps and issued to the SIMD pipeline. In this way, the SIMD pipeline is fully utilized even when a shader program executes diverging branches. Figure 9 illustrates this idea. In this figure, eight scalar threads organized into two warps (W0 and W1) execute the "if-then-else" branch illustrated on the right hand side of the figure. Each execution cycle is represented by a rectangular box on the left hand side of the figure. The arrows within each rectangle represent the scalar threads within a warp. Threads with solid arrows diverge to code block A while threads with blank arrows diverge to code block B. In the top part of the figure the SIMD pipeline is not fully utilized after the branch divergence. In the bottom part of the figure new warps (W2 and W3) are created after the branch divergence. In this way dynamic warp formation allows full utilization of the pipeline.

Dynamic warp formation improves performance by creating new thread warps out of diverged warps as the shader program executes. Every cycle, the thread scheduler tries to form new warps from a pool of ready threads by combining scalar threads whose next PC values are the same. 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 cycle depending upon a scheduling policy. We explore the design space of this scheduling policy in detail in Section 4.3. We show that the thread scheduler policy is critical to the performance impact of dynamic warp formation in Section 6.

4.1. Register File Access

So far, we have described dynamic warp formation under the assumption that each thread can be executed in any of the scalar pipelines (or lanes). This requires the registers to be equally accessible from all lanes, as illustrated in Figure 10(a). To reduce area and support a large number of ports in an SIMD pipeline, a well known approach is to implement the register file in multiple banks, each accessible from a single lane as shown in Figure 10(c). The latter hardware is a natural fit when threads are grouped into warps "statically" before they begin executing instructions and stay in the same lane until they complete.

While grouping threads into warps dynamically, it is preferable to avoid the need to migrate register values with threads as they get regrouped into different warps. To do this, the registers used by each scalar thread are assigned statically at a given offset based upon the thread identifier within a given lane just as when threads are grouped into warps "statically". If we form a warp without consideration to the "home" lane of a scalar thread's registers, we must design the register file with a crossbar as in Figure 10(b). 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.

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 10(b) is unnecessary for lane aware dynamic warp formation, the traditional hardware in Figure 10(c) is insufficient. When threads are grouped into

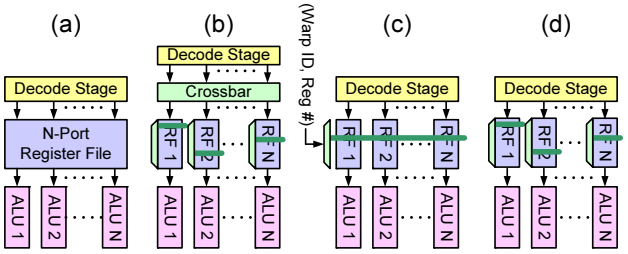

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.

warps "statically", each thread's registers are at the same "offset" within the 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[3]. This yields the register file configuration shown in Figure 10(d), which is accounted for in the area estimation in Section 4.4.

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 threads in all even lanes see a branch as taken, while threads in all odd lanes see the same branch as not-taken, then it is impossible for dynamic warp formation to create larger warps. A simple solution we employ for our simulations is to alternately swap the position of even and odd thread's home lanes every other warp when threads are first created (an approach we call *thread swizzling*).

4.2. Hardware Implementation

Figure 11 shows how dynamic warp formation and scheduling can be implemented in hardware. 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 11(a)). For conditional branches, there are at most two different next PC values[4]. 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[5]

[3]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.

[4]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.

[5]In our detailed model we assume the PC-warp LUT is organized as a small dual-ported set associative structure.

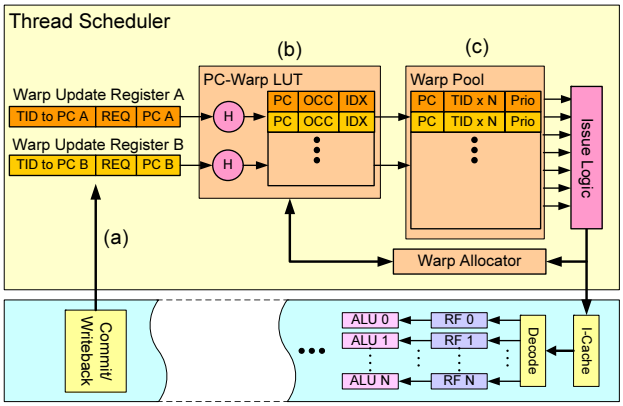

Figure 11. Implementation of dynamic warp formation and scheduling. In this figure, H represents a hash operation. N is the width of the SIMD pipeline.

(Figure 11(b)).

The PC-warp LUT provides a level of indirection to reduce the complexity of locating warps in the warp pool (Figure 11(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 next PC value. Each entry in the warp pool contains the PC value of the warp, N TID entries for N lanes in an N-wide SIMD pipeline, and some heuristic-specific data for issue logic. In order to handle the worst case where each thread diverges to a different execution path, the warp pool must have enough entries for each thread in a shader processor core to have its own entry.

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 a request vector (REQ) that indicates which lanes are required by the threads assigned to this warp. If a required lane is already occupied by a thread, 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. In this case, the PC-warp LUT IDX field is also 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, which may lower the effectiveness of dynamic warping.

A single warp in the warp pool may be issued to the SIMD pipeline every cycle according to one of the heuristics mentioned in the next section. Once issued, the warp will have its warp pool entry returned to the warp allocator.

4.3. Issue Heuristics

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 order in which warps are issued has a critical effect on this. We explored the following policies:

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 from which to create a new warp every cycle. The majority heuristic 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 heuristic tends to leave these threads behind. In the minority heuristic, warps with the least frequent PCs are given priority with the hope that, by doing so, these warps may eventually catch up and converge with other threads.

Time Stamp (DTime): The oldest warp will be issued first.

Post-Dominator Priority (DPdPri): Threads falling behind after a divergence need to catch up with other threads after the immediate post-dominator. If the issue priority is set lower for warps that have gone beyond more post-dominators, then the threads that have yet to go past the post-dominator tend to catch up.

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 issuing priority to warps with smaller PCs, threads lagging behind are given the opportunity to catch up.

We assume an implementation of the *Majority* issue logic constructed with a 32-entry fully associative lookup-table which tracks the number of threads currently in the scheduler for a given PC and keeps them sorted by number of threads. Each cycle, the issue logic searches for or allocates an entry for the PC of each warp entering the scheduler, and increments the associated counter with the number of scalar threads joining the warp pool. To facilitate selection of the most frequent PC value, the updated entry may be swapped with its neighbouring entry. If the number of PCs in flight exceeds the lookup-table capacity, a mechanism such as draining the threads from the majority logic could be implemented. However, we find that with a 32 entry table, the need for this never arises for our benchmarks.

Table 1. Area estimation for dynamic warp formation and scheduling. RP = Read Port, WP = Write Port, RWP = Read/Write Port.

Structure	# Entries	Entry Content	Struct. Size (bits)	Implementation	Area (mm^2)
Warp Update Register	2	TID (8-bit) × 16 PC (32-bit) REQ (8-bit)	336	Register (No Decoder)	0.008
PC-Warp LUT	32	PC (32-bit) OCC (16-bit) IDX (8-bit)	1792	2-Way Set-Assoc. Mem. (2 RP, 2 WP)	0.189
Warp Pool	256	TID (8-bit) × 16 PC (32-bit) Sche. Data (8-bit)	43008	Mem. Array (17 Decoders) (1 RWP, 2 WP)	0.702
Warp Allocator	256	IDX (8-bit)	2048	Memory Array	0.061
Issue Logic (Majority)	32	PC (32-bit) Counter (8-bit)	1280	Fully Assoc. (4RP, 4WP)	1.511
Total			48464		**2.471**

4.4. Area Estimation

We have estimated the area of the five major parts of the hardware implementation of dynamic warp formation and scheduling with CACTI 4.2 [33]: Warp Update Registers, PC-Warp LUT, warp pool, warp allocator, and issue logic. Table 1 list the implementation of these structures and their area and storage estimates. Using our baseline configuration (16-wide SIMD with 256 Threads) listed in Table 2, we have estimated the area of the dynamic warp scheduler in 90nm process technology to be 2.471 mm^2 per core.

To evaluate the overhead of having the individual decoder 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 [33] estimates a register file with 8192 32-bit registers and a single decoder reading a row of 16 registers to be 5.7085 mm^2.

On the other hand, since the SRAM model of CACTI 4.2 does not support banking directly, we decide to estimate the area of a register file with 512 32-bit registers and 1 register accessible per port. This register file is estimated[6] to be 0.3773 mm^2 × 16 = 6.037 mm^2. Notice that both register file configurations have 2 read ports and 1 write port.

With the two estimations above, the overall area consumption of dynamic warp formation and scheduling for each core is 2.799 mm^2. With 8 cores per chip as per our initial configuration, this becomes 22.39 mm^2, which is 4.7% of the total area of the GeForce 8800GTX (estimated from wafer photos to be roughly 480 mm^2).

5. Methodology

While simulators for contemporary GPU architectures exist, none of them model the general-purpose GPU ar-

[6]This may not entirely capture all wiring complexity since the decoders are driven by different sources in our proposal.

Table 2. Hardware Configuration

# Shader Cores	8
SIMD Warp Size	16
# Threads per Shader Core	256
# Memory Modules	8
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
Memory Controller	out of order
Data Cache Size (per core)	512KB 8-way set assoc.
Data Cache Hit Latency	10 cycle latency (pipelined 1 access/cycle)
Default Warp Issue Heuristic	majority

chitecture described in this paper. 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 SimpleScalar version 3.0d [7]. Sim-outorder was modified to offload manually annotated compute kernels to our cycle-accurate GPU performance simulator, which we developed around SimpleScalar's PISA instruction set architecture. Table 2 shows the baseline configuration we simulated.

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

The benchmark applications used for this study were selected from SPEC CPU2006 [32], SPLASH2[36], and CUDA[25]. 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. Our GPU simulator's programing model is similar to that of CUDA [27]. A computing kernel is invoked by a spawn instruction, which signals the SimpleScalar out-of-order core to launch a predetermined number of threads for parallel execution on the GPU simulator.

6. Experimental Results

First we consider dynamic warp formation and scheduling ignoring the impact of lane conflicts.

Figure 12 shows the performance of the different branch handling mechanisms discussed so far in this paper 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. PDOM (reconvergence at the immediate post-dominator described in Section 3.2) achieves a speedup of 93.4% versus not reconverging (Naïve). Dynamic warp formation (DYNB) achieves a further speedup of 22.5% using the Majority heuristic. The difference between average DYNB and MIMD performance is only 4.6%.

Two of the benchmarks (Matrix and FFT) obtain slightly

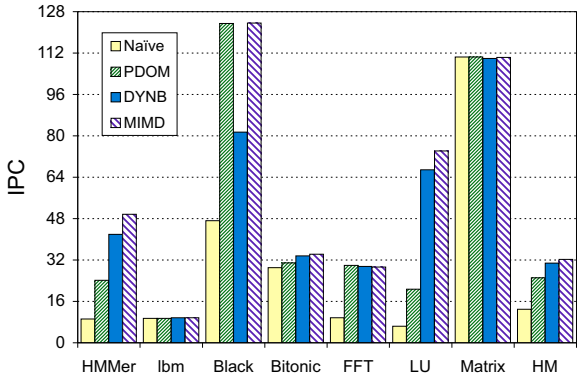

Figure 12. Performance comparison of Naïve, PDOM, and DYNB versus MIMD.

Figure 13. Memory bandwidth utilization.

	HMMer	lbm	Black	Bitonic	FFT	LU	Matrix
PDOM	58.60%	93.20%	4.96%	36.21%	79.90%	0.76%	38.02%
DYNB	56.86%	95.29%	3.06%	39.95%	78.23%	9.58%	37.01%

Figure 14. Cache miss rates.

	HMMer	lbm	Black	Bitonic	FFT	LU	Matrix
PDOM	3.43%	13.35%	1.43%	18.22%	7.35%	0.08%	2.86%
DYNB	1.85%	13.53%	1.34%	24.60%	6.86%	0.30%	2.90%

lower performance with MIMD than with PDOM. We found that this was because the "free running" nature of MIMD tended to lower the spatial locality of memory accesses, resulting a higher cache miss rate (7.67% vs 7.35% for FFT and 3.43% vs 2.86% for Matrix—see Table 14). The benchmarks where PDOM outperforms DYNB (Black, FFT, Matrix) are the ones with limited divergence. Among them, the minor slowdown of DYNB on FFT and Matrix is attributed to an extra stage in the pipeline for register file access overhead for DYNB. The Majority issue heuristic used by DYNB also lacks the consideration of DRAM bank conflicts in these benchmarks, introducing disruptive memory access behaviour and slightly under utilizing the DRAM bandwidth in comparison to PDOM.

The significant slowdown of Black Scholes (Black) is a phenomenon exposing the weakness of our default Majority issue heuristic. Under the Majority issue heuristic, a minority of threads whose control flow behaviors are different from the rest of the threads are starved during execution. Black Scholes has several short and rarely taken branches that suffer from this weakness, leaving behind several groups of minority threads. When these minority threads finally execute after the majority of threads have reached a fence operation, they form incomplete warps and the number of warps formed is insufficient to fill up the pipeline. This can be solved by having a better heuristic.

While benchmark lbm has signicant diverging control flow it is memory bandwidth limited, as shown in Table 13 and therefore sees little gain from dynamic warp formation.

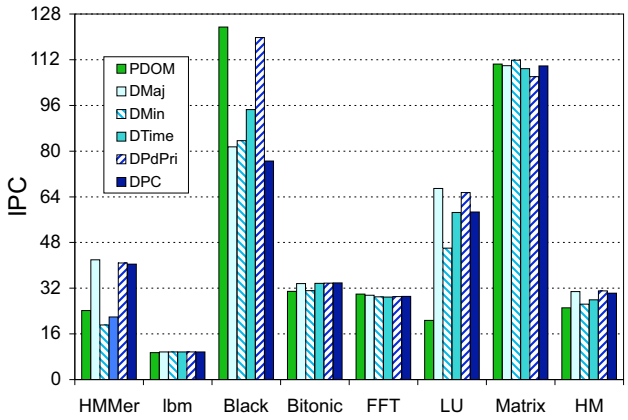

Figure 15. Comparison of warp issue heuristics.

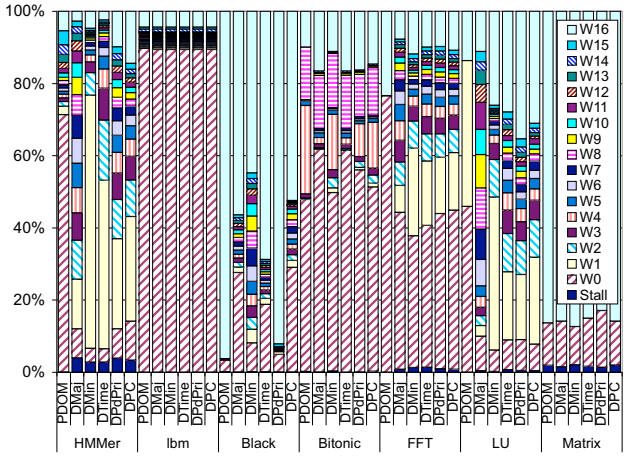

Figure 16. Warp size distribution.

6.1. Effects of Issue Heuristics

Figure 15 compares all the warp issue heuristics described in Section 4.3, again with a realistic memory subsystem, but still ignoring the impact of lane conflicts. Overall, the default Majority (DMaj) heuristic performs well, achieving an average speedup of 22.5%, but in some cases, its performance is not as good as the PC heuristic (DPC) or PDOM Priority (DPdPri) described in Section 4.3. To provide additional insight into the differences, Figure 16 shows the distribution of warp sizes issued each cycle for each heuristic. Each bar is divided into segments labeled W0, W1, ... W16, which indicate if the SIMD hardware executed operations for 0, 1, ... 16 scalar threads on a given cycle. "Stall" indicates a stall due to writeback contention with the memory system (see Figure 3). For heuristics that do well (DMaj, DPdPri, DPC), we see a decrease in the number of low occupancy warps relative to those heuristics which do poorly (DMin, DTime). The data also suggests it may be possible to further improve dynamic warp formation by exploring the scheduler heuristics beyond those proposed in this paper.

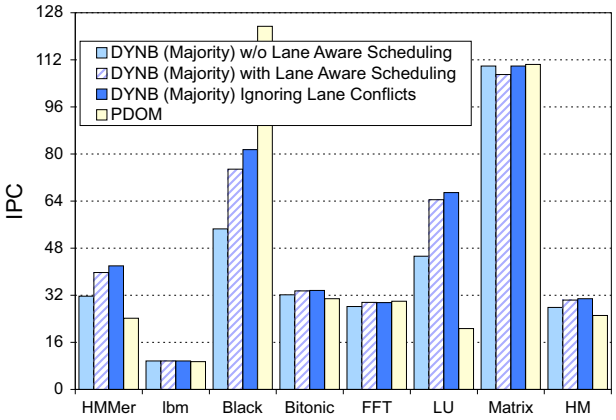

Figure 17. Performance of dynamic warp formation with lane aware scheduling and accounting for register file bank conflicts and scheduler implementation details.

6.2. Effect of Lane Aware Scheduling

To reduce register file design complexity we would like to use the organization in Figure 10(d) which necessitates lane aware scheduling discussed in Sections 4.1 and 4.2. The data in Figure 17 is measured with a detailed scheduler hardware model like that described in Section 4.2 with hardware sized as in Table 1 and employing the thread swizzling mechanism described in Section 4.1. This figure shows the impact on performance of lane aware scheduling and, for comparison, also shows the impact when not using lane aware scheduling but assuming the register file organization in Figure 10(b) and modeling register bank conflicts when multiple threads from the same "home" lane are grouped into a single warp. While lane-aware scheduling is slower than dynamic warp formation assuming no performance penalty due to bank conflicts (by 1.43%), it is still 20.7% faster than PDOM. This speedup justifies the 4.7% area cost (see Section 4.4) of using dynamic warp formation and scheduling on existing GPUs. We also evaluated the performance of PDOM Priority with lane aware scheduling and found performance for Black Scholes improves significantly (IPC of 103) while that of HMMer is reduced (IPC of 25) with an overall average speedup of 14.7% over PDOM (versus 23.7% without lane aware scheduling and when ignoring the performance impact of lane conflicts).

7. Related Work

While supporting branches is a relatively new problem for GPU architecture, it has long been a consideration in

the context of traditional vector computing. Most of the approaches to supporting branches in a traditional SIMD machine have centered around the notion of guarded instructions [5].

A guarded instruction, also known as a predicated or vector masked instruction, is an instruction whose execution is dependent on a conditional mask controlled by another instruction [5]. If the conditional mask is set, the result of the instruction will not be committed, nullifying the instruction. In an SIMD instruction, a vector of conditional masks each controlled by an element in a stream would be functionally equivalent to a data dependent branch while maintaining consistent control-flow. This approach has been employed by ATI's CTM and NVIDIA's CUDA architecture to eliminate short branches and possible branch divergences [1, 27].

Guarded instructions and its variants, however, cannot eliminate 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. describes in detail how threads executing in an SIMD pipeline are serialized to avoid hazards [22], but does not indicate the use of reconvergence points to recover from such divergence. The notion of reconvergence based on control-flow analysis in SIMD branch handling was described by Lorie and Strong [19]. However, the patent proposes to insert the reconvergence point at the beginning of a branch and not at the immediate post-dominator as proposed in this paper.

Besides these two main approaches, Woop et al. proposed a complex SIMD branch instruction which outputs a single final branch condition with a reduction function on the masked element branch condition [37] in their Ray Processing Unit. In this manner, the final branch condition is always consistent for all elements in an SIMD warp. While this approach eliminates branch divergence, it may not be suitable for most general-purpose applications that require the execution path of each shader program to be different.

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 [9] in the context of simultaneous multithreading (SMT) on a general purpose microprocessor that provides SIMD function units for exploiting subword parallelism. The mechanism Cervini proposes requires that tasks have their register values reloaded each time threads are regrouped. To avoid performance penalties, Cervini proposes that the register file contain additional ports to enable the register values to be loaded concurrently with ongoing execution. In addition, Cervini's mechanism uses special "code stops" and tags the control flow state of a SPMD thread with a loop counter list (in addition to the program counter). We point out that in the context of a modern GPU the constant movement of data in this proposal could increase power requirements per processing element, perhaps mitigating the improvements in processing efficiency given the power wall [13]. In contrast, our proposal uses a highly banked large register file and maintains a thread's registers in a single location to eliminate the need for movement of register values.

Clark et al. [10] introduce *Liquid SIMD*, to improve SIMD binary compatibility by forming SIMD instructions at runtime by translating annotated scalar instructions with specialized hardware. In contrast, we focus on improving performance by regrouping threads into new SIMD warps even after threads start executing. Shin et al. [31] examine compiler generation of *branches on superword condition codes* (BOSCCs) for handling control flow for SIMD instruction set extensions. Kapasi [16] introduces *conditional routing*, a code transformation that creates multiple kernels from a single kernel with conditional code and connects them via inter-kernel communication to increase the utilization of an SIMD pipeline. Dynamic warp formation, differs from this approach in that it exploits the dynamic conditional behaviour of each scalar thread.

8. Summary

In this paper, we explore the impact of branch divergence on GPU performance for non-graphics 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 post-dominator improve performance, we found significant performance improvements can be achieved with our proposed dynamic warp formation and scheduling mechanism. We described and evaluated a implementation of the hardware required for dynamic warp formation and tackle the challenge of enabling correct access to register data as thread warps are dynamically regrouped and found performance improved by 20.7% on average over a mechanism comparable to existing approaches—reconverging threads at the immediate post-dominator. Furthermore, we estimated the area of our proposed hardware changes to be around 4.7%.

Our experimental results highlight the importance of careful prioritization of threads 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.

9. Acknowledgments

We thank the reviewers for their helpful comments and Henry Tran for contributions to our simulation infrastruc-

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

References

[1] Advanced Micro Devices, Inc. *ATI CTM Guide*, 1.01 edition, 2006.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th Symp. on Principles of programming languages (POPL '83)*, pages 177–189, 1983.

[3] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Comm. ACM*, 19(10):542–547, 1976.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[5] W. Bouknight, S. Denenberg, D. McIntyre, J. Randall, A. Sameh, and D. Slotnick. The Illiac IV System. *Proc. of the IEEE*, 60(4):369–388, 1972.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH*, pages 777–786, 2004.

[7] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. http://www.simplescalar.com, 1997.

[8] E. Catmuli. Computer display of curved surfaces. In *Proc. Conf. on Comptr. Graphics, Pattern Recognition, and Data Structure*, pages ll–17, May 1975.

[9] S. Cervini. European Patent EP 1531391 A2: System and method for efficiently executing single program multiple data (SPMD) programs, May 2005.

[10] H. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. In *Proc. Int'l Symp. on High Performance Computer Architecture*, pages 216–227, 2007.

[11] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *Proc. of Supercomputing*, 2003.

[12] W. J. Dally and B. Towles. *Interconnection Networks*. Morgan Kaufmann, 2004.

[13] S. N. et al. When Processors Hit the Power Wall (or "When the CPU hits the fan"). In *Digest of Technical Papers, IEEE Int'l Solid-State Circuits Conference (ISSCC)*, pages 16–17, February 2005.

[14] H. Fowler, F. Fowler, and D. Thompson, editors. *The Concise Oxford Dictionary*. Oxford University Press, 9th edition, 1995.

[15] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMer: A Streaming HMMer-Search Implementation. In *Proc. of Supercomputing*, page 11, 2005.

[16] U. J. Kapasi. *Conditional Techniques for Stream Processing Kernels*. PhD thesis, Stanford University, March 2004.

[17] A. Levinthal and T. Porter. Chap - a SIMD graphics processor. In *SIGGRAPH*, pages 77–82, 1984.

[18] E. Lindholm, M. J. Kligard, and H. P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[19] R. A. Lorie and H. R. Strong. US Patent 4,435,758: Method for conditional branch execution in SIMD vector processors, 1984.

[20] D. Luebke and G. Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.

[21] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[22] S. Moy and E. Lindholm. US Patent 6,947,047: Method and system for programmable pipelined graphics processing with branching instructions, 2005.

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmanns, 1997.

[24] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of tow proteins. *Molecular Biology*, (48):443–453, 1970.

[25] NVIDIA Corporation. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[26] NVIDIA Corporation. *Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview*, November 2006.

[27] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 1.0 edition, 2007.

[28] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH*, pages 703–712, 2002.

[29] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proc. 31st Int'l Symp. on Microarchitecture*, pages 3–13, 1998.

[30] M. Shebanow. ECE 498 AL : Programming massively parallel processors, lecture 12. http://courses.ece.uiuc.edu/ece498/al1/Archive/Spring2007, February 2007.

[31] J. Shin, M. Hall, and J. Chame. Introducing control flow into vectorized code. In *(to appear in) Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2007.

[32] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmarks. http://www.spec.org/cpu2006/.

[33] D. Tarjan, S. Thoziyor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Laboratories Palo Alto, June 2006.

[34] M. R. Thistle and B. J. Smith. A processor architecture for Horizon. In *Proc. of Supercomputing*, pages 35–41, 1988.

[35] S. Upstill. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Reading, Mass: Addison-Wesley, 1990.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Int'l Symp. on Computer Architecture*, pages 24–36, 1995.

[37] S. Woop, J. Schmittler, and P. Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph.*, 24(3):434–444, 2005.