The Dual-Path Execution Model for Efficient GPU Control Flow

Minsoo Rhu
Electrical and Computer Engineering Department
The University of Texas at Austin
minsoo.rhu@utexas.edu

Mattan Erez
Electrical and Computer Engineering Department
The University of Texas at Austin
mattan.erez@mail.utexas.edu

Abstract

Current graphics processing units (GPUs) utilize the single instruction multiple thread (SIMT) execution model. With SIMT, a group of logical threads executes such that all threads in the group execute a single common instruction on a particular cycle. To enable control flow to diverge within the group of threads, GPUs partially serialize execution and follow a single control flow path at a time. The execution of the threads in the group that are not on the current path is masked. Most current GPUs rely on a hardware reconvergence stack to track the multiple concurrent paths and to choose a single path for execution. Control flow paths are pushed onto the stack when they diverge and are popped off of the stack to enable threads to reconverge and keep lane utilization high. The stack algorithm guarantees optimal reconvergence for applications with structured control flow as it traverses the structured control-flow tree depth first. The downside of using the reconvergence stack is that only a single path is followed, which does not maximize available parallelism, degrading performance in some cases. We propose a change to the stack hardware in which the execution of two different paths can be interleaved. While this is a fundamental change to the stack concept, we show how dual-path execution can be implemented with only modest changes to current hardware and that parallelism is increased without sacrificing optimal (structured) control-flow reconvergence. We perform a detailed evaluation of a set of benchmarks with divergent control flow and demonstrate that the dual-path stack architecture is much more robust compared to previous approaches for increasing path parallelism. Dual-path execution either matches the performance of the baseline single-path stack architecture or outperforms single-path execution by 14.9% *on average and by over* 30% *in some cases.*

1 Introduction

Current graphics processing unit (GPU) architectures balance the high efficiency of single instruction multiple data (SIMD) execution with programmability and dynamic control. GPUs group multiple logical threads together (32 or 64 threads in current designs [3, 22]) that are then executed on the SIMD hardware. While SIMD execution is used, each logical thread can follow an independent flow of control, in the *single instruction multiple thread* execution style. When control flow causes threads within the same group to *diverge* and take different control flow paths, hardware is responsible for maintaining correct execution. This is currently done by utilizing a reconvergence predicate stack, which partially serializes execution. The stack mechanism partitions the thread group into subgroups that share the same control flow. A single subgroup executes at a time, while the execution of those threads that follow a different flow is masked [18, 19]. While this mechanism is effective and efficient, it does not maximize available parallelism because it serializes the execution of different subgroups, which degrades performance in some cases.

In this paper we present a mechanism that requires only small changes to the current reconvergence stack structure, maintains the same SIMD execution efficiency, and yet is able to increase available parallelism and performance. Unlike previously proposed solutions to the serialized parallelism problem [6, 20], our *dual-path execution* (DPE) model requires no heuristics, no compiler support, and is robust to changes in architectural parameters. With DPE, hardware does not serialize all control flows, and instead is able to interleave execution of the taken and not-taken flows. To summarize our most significant contributions:

- We propose the first design that extends the reconvergence stack model, which is the dominant model for handling branch divergence in GPUs. DPE maintains the simplicity, robustness, and high SIMD utilization advantages of the stack approach, yet is able to exploit more parallelism by interleaving the execution of diverged control-flow paths.
- We describe and explain the microarchitecture in depth and show how it can integrate with current GPUs.
- We show that DPE always either matches or exceeds the performance and SIMD efficiency of the baseline design with up to a 42% improvement and an average of 14.9% speedup for those benchmarks that can benefit from DPE. This is in stark contrast to prior work [12, 20], which significantly degrades performance and efficiency in several cases.

This is the authors version of the work. The authoritative version will appear in the Proceedings of HPCA 2013.

2 Background

The typical current GPU consists of multiple processing cores ("streaming multiprocessors (SMs)" and "SIMD units" in NVIDIA and AMD terminology respectively), where each core consists of a set of parallel execution lanes ("CUDA cores" and "SIMD cores" in NVIDIA and AMD). In the GPU execution model, each core executes a large number of logically independent threads, which are all executing the same code (referred to as a *kernel*). These parallel lanes operate in a SIMD/vector fashion where a single instruction is issued to all the lanes for execution. Because each thread can follow its own flow of control while executing on SIMD units, the name used for this hybrid execution model is *single instruction multiple thread* (SIMT). In NVIDIA GPUs, each processing core schedules a SIMD instruction from a *warp* of 32 *threads* while AMD GPUs currently schedule instructions from a *wavefront* of 64 *workitems*. In the rest of the paper we will use the terms defined in the CUDA language [24]. This execution model enables very efficient hardware, but requires a mechanism to allow each thread to follow its own thread of control, even though only a single uniform operation can be issued across all threads in the same warp.

In order to allow independent branching, hardware must provide two mechanisms. The first mechanism determines which single path, of potentially many control paths, is *active* and is executing the current SIMD instruction. The technique for choosing the active path used by current GPUs is *stack-based reconvergence*, which we explain in the next subsection. The second mechanism ensures that only threads that are on the active path, and therefore share the same program counter (PC) can execute and commit results. This can be done by associating an *active mask* with each SIMD instruction that executes. Threads that are in the executing SIMD instruction but not on the active control path are masked and do not commit results. The mask may either be computed dynamically by comparing the explicit PC of each thread with the PC determined for the active path, or alternatively, the mask may be explicitly stored along with information about the control paths. The GPU in Intel's Sandy Bridge [17] stores an explicit PC for each thread while GPUs from AMD and NVIDIA currently associate a mask with each path.

2.1 Stack-Based Reconvergence

A significant challenge with the SIMT model is maintaining high utilization of the SIMD resources when the control flow of different threads within a single warp diverges. There are two main reasons why SIMD utilization decreases with divergence. The first is that masked operations needlessly consume resources. This problem has been the focus of a number of recent research projects, with the main idea being that threads from multiple warps can be combined to reduce the fraction of masked operations [11, 12, 21, 27]. The second reason is that execution of concurrent control paths is serialized with every divergence potentially decreasing parallelism. Therefore, care must be taken to allow them to reconverge with one another. In current GPUs, all threads that reach a specific diverged branch reconverge at the *immediate post-dominator* instruction of

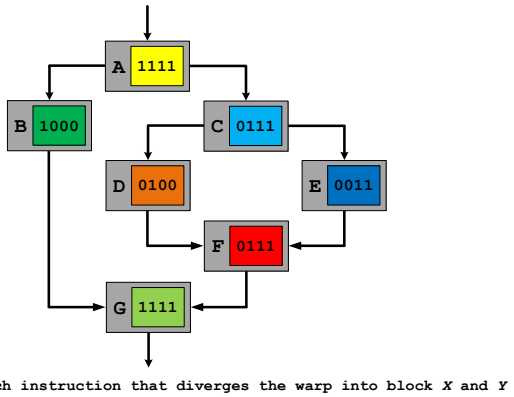


Figure 1: Example control-flow graph. Each warp consists of 4 threads and ones and zeros in the control-flow graph designate the active and inactive threads in each path. When discussing DWS, we assume that all basic blocks execute exactly three instructions, with the exception of block F that executes a single instruction.

that branch [29]. The *post-dominator* (PDOM) instruction is the first instruction in the static control flow that is guaranteed to be on both diverged paths [12]. For example, in Figure 1, the PDOM of the divergent branch at the end of basic block A (BR_{B-C}) is the instruction that starts basic block G. Similarly, the PDOM of BR_{D-E} at the end of basic block C is the instruction starting basic block F.

An elegant way to implement PDOM reconvergence is to treat control flow execution as a serial stack. Each time control diverges, both the taken and not taken paths are *pushed* onto a stack (in arbitrary order) and the path at the new top of stack is executed. When the control path reaches its reconvergence point, the entry is *popped* off of the stack and execution now follows the alternate direction of the diverging branch. This amounts to a serial depth-first traversal of the control-flow graph. Note that only a *single* path is executed at any given time, which is the path that is logically at the top of the stack. There are multiple ways to implement such stack model, including both explicit hardware structures and implicit traversal with software directives [4, 8]. We describe our mechanisms in the context of an explicit hardware approach, which we explain below. According to prior publications, this hardware approach is used in NVIDIA GPUs [8]. We discuss the application of our technique to GPUs with software control in Section 7.

The hardware reconvergence stack tracks the program counter (PC) associated with each control flow path, which threads are active at each path (the active mask of the path), and at what PC should a path reconverge (RPC) with its predecessor in the control-flow graph [12]. The stack contains the information on the control flow of all threads within a warp. Figure 2 depicts the reconvergence stack and its operation on the example control flow shown in Figure 1. We describe this example in detail below.

When a warp first starts executing, the stack is initialized with a single entry: the PC points to the first instructions of the kernel (first instruction of block A), the active mask is full, and the RPC (reconvergence PC) is set to the end of the kernel. When a warp executes a conditional branch, the predicate values for both the taken and non-taken paths

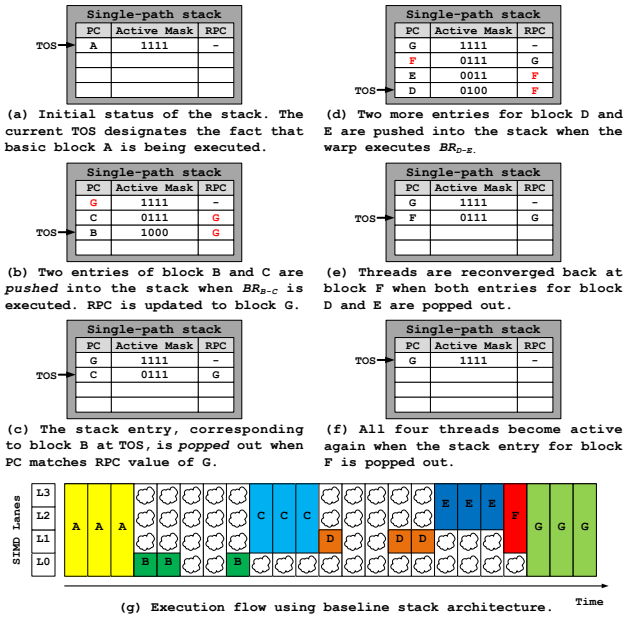


Figure 2: High-level operation of SIMT stack-based reconvergence when executing the control-flow graph in Figure 1. The ones/zeros inside the active mask field designate the active threads in that block. *Bubbles* in (g) represent idle execution resources (masked lanes or zero ready warps available for scheduling in the SM).

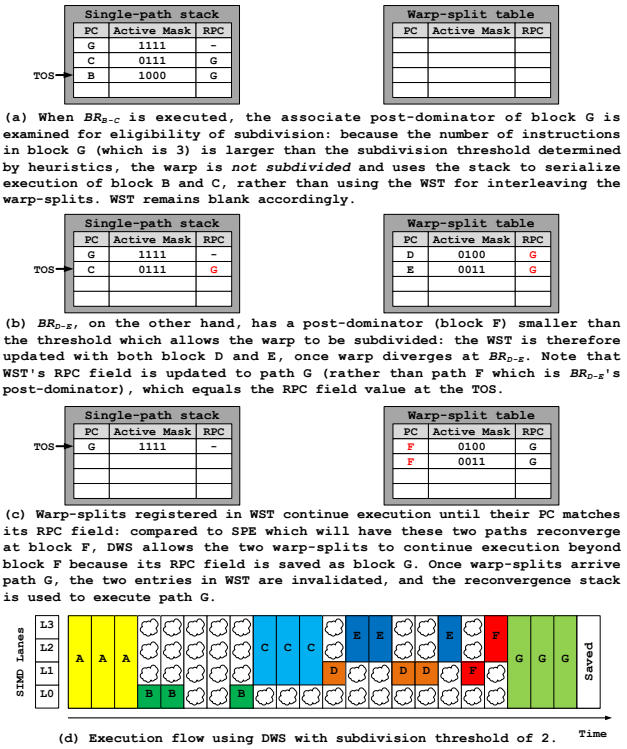


Figure 3: High-level operation of DWS with a subdivision threshold of 2. We assume the same control-flow graph in Figure 1.

(left and right paths) are computed. If control diverges with some threads following the taken path and others the non-taken path, the stack is updated to include the newly formed paths (Figure 2 (b)). First, the PC field of the current *top of the stack* (TOS) is modified to the PC value of the reconvergence point, because when execution returns to this path, it would be at the point where the execution reconverges (start of block G in the example). The RPC value is explicitly communicated from software and is computed with a straightforward compiler analysis [29]. Second, the PC of the right path (block C), the corresponding active mask, and the RPC (block G) is pushed onto the stack. Third, the information on the left path (block B) is similarly pushed onto the stack. Finally, execution moves to the left path, which is now at the TOS. Note that only a single path per warp, the one at the TOS, can be scheduled for execution. For this reason we refer to this baseline architecture as the *single-path execution* (SPE) model.

When the current PC of a warp matches the RPC field value at that warp's TOS, the entry at the TOS is popped off (Figure 2 (c)). At this point, the new TOS corresponds to the right path of the branch and the warp starts executing block C. As the warp encounters another divergent branch at the end of block C, the stack is once again updated with the left and right paths of blocks D and E (Figure 2 (d)). Note how the stack elegantly handles the nested branch and how the active masks for the paths through blocks D and E are each a subset of the active mask of block C. When both left and right paths of block D and E finish execution and corresponding stack entries are popped out, the TOS points to block F and control flow is reconverged back to the path that started at block C (Figure 2 (e)) – the active mask is set correctly now that the nested branch reconverged. Similarly, when block F finishes execution and the PC equals the reconvergence PC (block G), the stack is again popped and execution continues along a single path with a full active mask (Figure 2 (f)).

This example also points out the two main deficiencies of the SPE model. First, SIMD utilization decreases every time control flow diverges. SIMD utilization has been the focus of active research (e.g., [11, 12, 21, 27]) and we do not discuss it further in this paper[1]. Second, execution is serialized such that only a single path is followed until it completes and reconverges (Figure 2 (g)). Such SPE model works well for most applications because of the abundant parallelism exposed through multiple warps within cooperative thread arrays. However, for some applications, the restriction of following only a single path does degrade performance. Meng et al. proposed *dynamic warp subdivision* (DWS) [20], which selectively deviates from the reconvergence stack execution model, to overcome the serialization issue. We explain DWS below as we use it as a comparison point in our evaluation of the dual-path execution model. More recently, Brunie et al. [6] proposed a mechanism that is able to increase intra-warp parallelism, but relies on significant modifications to the underlying GPU architecture described above. Because it targets a different baseline design, we discuss this technique in Section 8.

[1] As such compaction-based GPU architectures trade-off SIMD utilization with thread-level parallelism, our DPE model provides additional benefits by enhancing path parallelism. We leave its evaluation to future work.

2.2 Dynamic Warp Subdivision

Dynamic warp subdivision (DWS) was proposed to allow warps to interleave the scheduling of instructions from concurrently executable paths [20]. The basic idea of DWS is to treat both the left and right paths of a divergent branch as independently schedulable units, or *warp-splits*, such that diverging path serialization is avoided and *intra*-warp latency tolerance is achieved[2]. With DWS, a divergent branch may either utilize the baseline single-path stack mechanism (Figure 3 (a)), or instead, ignore the stack and utilize an additional hardware structure, the *warp-split table* (WST), which is used to track the independently-schedulable warp-splits (Figure 3 (b)). Nested branches within a split warp cause further splits. As with the stack, this successively degrades SIMD efficiency. Unlike the stack, split warps are dynamically and concurrently scheduled and may not reconverge as early as the post-dominator.

To prevent very low SIMD lane utilization, DWS uses a combination of three techniques. First, the WST also contains a reconvergence PC like the stack. This RPC, however, is not the PDOM of the diverging branch, but rather the PDOM of the last entry in the stack. Because the stack cannot be used to track warp-splits, further subdivisions also use this same RPC value and miss many opportunities for reconvergence. This decision increases parallelism and potential latency hiding at the expense of reduced SIMD utilization (the stack could have reconverged nested branches whereas the WST cannot). Second, to reduce the impact of late reconvergence and recursive subdivision, DWS attempts to dynamically and opportunistically recombine warp-splits when two splits happen to reach the same PC. Unlike the PDOM reconvergence stack mechanism, this opportunistic merging is not guaranteed and may never occur, as illustrated with block F executing twice in Figure 3 (c–d). Therefore, third, DWS also relies on a heuristic for determining whether to split a warp in the first place: a warp is subdivided only if the divergent branch's immediate post-dominator is followed up by a short basic block of no more than N instructions. Meng et al. suggest that this *subdivision threshold* (N) should be 50 instructions [20], which we refer to as DWS_{50}.

As we show in our evaluation, DWS cannot consistently balance increased parallelism and SIMD utilization and often degrades performance when compared to the baseline SPE. The threshold heuristic is sensitive, with small values of N often preventing splits and not increasing thread-level parallelism (TLP) significantly, while high values of N split too aggressively and exhibit low SIMD utilization. Further, the WST adds complexity and the compiler may need to change heuristics based on the specific parameters of the hardware and application. In contrast, dual-path execution is very robust; it does not degrade performance compared to the baseline and outperforms DWS in all but one experiment despite DWS exposing greater parallelism.

[2]Meng et al. [20] also propose to subdivide warps upon memory divergence, which is orthogonal to subdivision at control divergence. Our work only focuses on control divergence subdivision, although we briefly discuss how memory divergence subdivision can be incorporated into the dual-path execution model in Section 7.

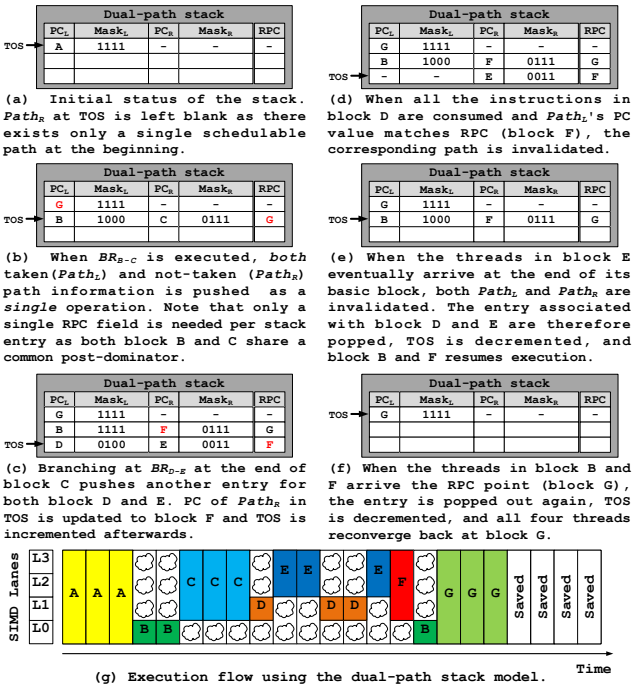


Figure 4: High-level operation of dual-path execution model. We assume the same control-flow graph and assumptions in Figure 1.

2.3 Limitation of Previous Models

As discussed in previous subsections, both SPE and DWS are able to address only one aspect of the control divergence issue while overlooking the other. SPE uses simple hardware and an elegant execution model to maximize SIMD utilization with structured control flow, but *always* serializes execution with only a single path schedulable at any given time. DWS can interleave the scheduling of multiple paths and increase TLP, but this sacrifices SIMD lane utilization. Our proposed *dual-path execution* (DPE) model, on the other hand, always matches the utilization and SIMD efficiency of the baseline SPE while still enhancing TLP in some cases. DPE keeps the elegant reconvergence stack model and the hardware requires only small modifications to utilize up to two interleaved paths. In the following section, we detail the microarchitectural aspects of DPE, followed by a detailed evaluation and discussion.

3 Dual-Path Execution Model

This work is motivated by the key observation that previous architectures either rely on stack-based reconvergence and restrict parallelism to a single path at any given time, or that stack-based reconvergence is abandoned leading to much more complex implementations and possible degradation of SIMD efficiency (with DWS). Our approach maintains the simplicity and effectiveness of stack-based reconvergence but exposes greater parallelism to the scheduler. With DPE, the execution of up to two separate paths can be interleaved, while reconvergence is identical to the baseline stack-based reconvergence. Support for DPE is only required in a small number of components within the GPU

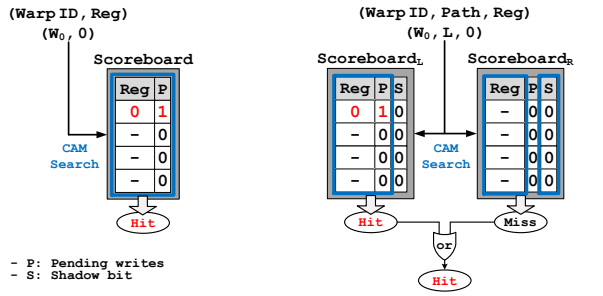


Figure 5: Comparison of scoreboards used for single-path and dual-path execution. Each scoreboard reflects the status after executing the first *load* instruction in path A of Figure 6

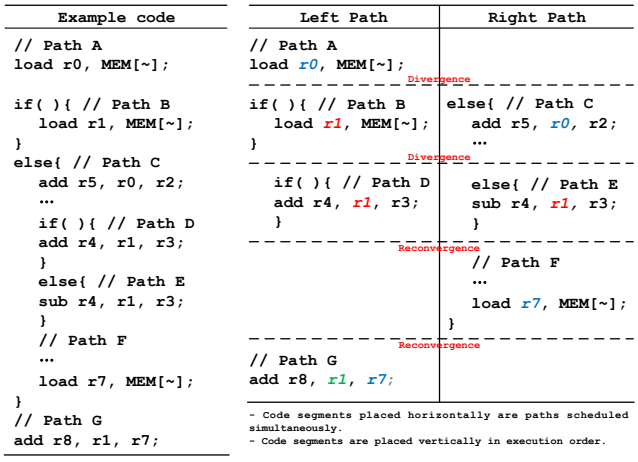


Figure 6: Data dependencies across different execution paths (control flow of Figure 1)

microarchitecture and requires no support from software. Specifically, the stack itself is enhanced to provide up to two concurrent paths for execution, the scoreboard is modified to track dependencies of two concurrent paths and to correctly handle divergence and reconvergence, and the warp scheduler is extended to handle up to two schedulable objects per warp. We detail the DPE microarchitecture below and use a running example of the control flow in Figure 1 corresponding to the code shown in Figure 6.

3.1 Dual-Path Stack Structure

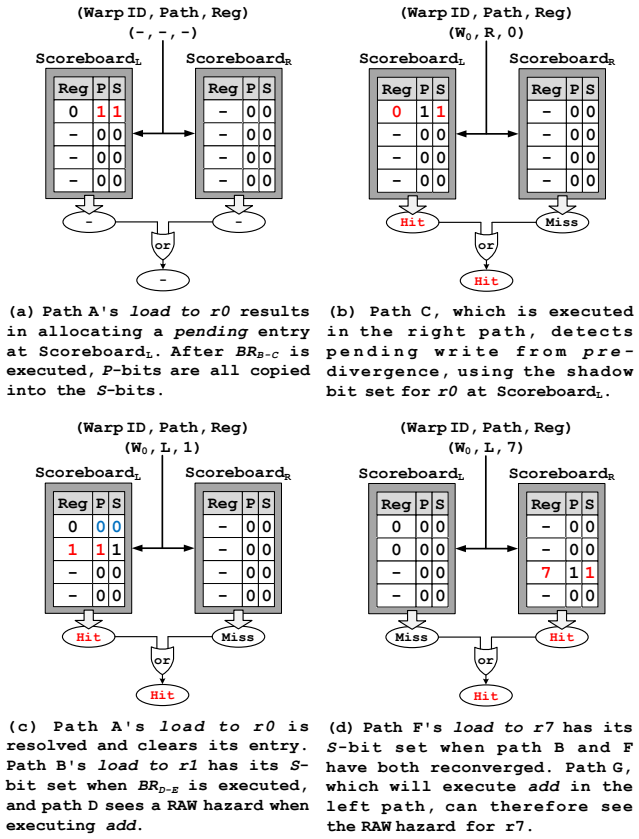
DPE extends the hardware stack used in many current GPUs to support two concurrent paths of execution. The idea is that instead of pushing the taken and fall-through paths onto the stack one after the other, in effect serializing their execution, the two paths are maintained in parallel. A stack entry of the dual-path stack architecture thus consists of three data elements: a) PC and active mask value of the left path ($Path_L$), b) PC and active mask value of the right path ($Path_R$), and c) the RPC (reconvergence PC) of the two paths. We use the generic names left and right because there is no reason to restrict the mapping of taken and non-taken paths to the fields of the stack. Note that there is no need to duplicate the RPC field within an entry because $Path_L$ and $Path_R$ of a divergent branch have a common reconvergence point. Besides the data fields that constitute a stack entry, the other components of the control flow hardware, such as the logic for computing active masks and managing the stack, are virtually identical to those used in the baseline stack architecture. The dual-path stack architecture exposes the two paths for execution on a divergent branch and can improve performance when this added parallelism is necessary, as shown in Figure 4, and described below for the cases of divergence and reconvergence.

Handling Divergence. A warp starts executing on one of the paths, for example the left path, with a full active mask, the PC set to the first instruction in the kernel and the RPC set to the last instruction ($Path_L$ in Figure 4 (a)). The warp then executes in identical way to the baseline single-path stack until a divergent branch executes. When the warp executes a divergent branch, the dual-path stack architecture pushes a single entry onto the stack, which represents both sides of the branch, rather than pushing two distinct entries as done with the baseline SPE. The PC field of the block that diverged is set to the RPC of both the left and right paths (block G in Figure 4 (b)), because this is the instruction that should execute when control returns to this path. Then, the active mask and PC of $Path_L$, as well as the same information for $Path_R$ are pushed onto the stack, along with their common RPC and updating the TOS (Figure 4 (b)). Because it contains the information for both paths, the single TOS entry enables the warp scheduler to interleave the scheduling of active threads at both paths as depicted in Figure 4 (g). If both paths are active at the time of divergence, the one to diverge (block C in Figure 4 (b)) first pushes an entry onto the stack, and in effect, suspends the other path (block B in Figure 4 (c)) until control returns to this stack entry (Figure 4 (e)). Note that the runtime information required to update the stack entries is exactly the same as in the baseline single-stack model.

Handling Reconvergence. When either one of the basic blocks at the TOS arrives at the reconvergence point and its PC matches the RPC, the block is invalidated (block D in Figure 4 (d)). Because the right path is still active, though, the entry is not yet popped off of the stack. Once both paths arrive at the RPC, the stack is popped and control is returned to the next stack entry (Figure 4 (e–f)).

3.2 Scoreboard

Recent GPUs from NVIDIA, such as Fermi, allow threads within the same warp to be issued back to back using a per-warp scoreboard to track data dependencies. One possible implementation of the scoreboard ([9]) is a content-addressable-memory (CAM) structure that is indexed with a register number and a warp ID which returns whether that register is pending write-back for that warp (Figure 5 (a)). When an instruction is decoded, the source and destination registers are searched in the scoreboard and only instructions with no RAW/WAW hazards are considered for scheduling. Once an instruction is scheduled for



(a) Path A's *load to r0* results in allocating a *pending* entry at Scoreboard$_L$. After BR_{B-C} is executed, *P*-bits are all copied into the *S*-bits.

(b) Path C, which is executed in the right path, detects pending write from *pre-divergence*, using the shadow bit set for *r0* at Scoreboard$_L$.

(c) Path A's *load to r0* is resolved and clears its entry. Path B's *load to r1* has its *S*-bit set when BR_{D-E} is executed, and path D sees a RAW hazard when executing *add*.

(d) Path F's *load to r7* has its *S*-bit set when path B and F have both reconverged. Path G, which will execute *add* in the left path, can therefore see the RAW hazard for r7.

Figure 7: Example of how the proposed scoreboard handles data dependencies across different paths in Figure 6.

execution, the scoreboard is updated to show the instruction's destination register as pending. When the register is written back, the scoreboard is updated and the pending bit is cleared. To support multiple concurrent paths per warp, the scoreboard must be modified to track the register status of both the right and left paths of each warp independently while still correctly handling divergence and reconvergence when dependencies are crossed from one path to the other.

We accomplish this with two modifications to the scoreboard. First, we extend the scoreboard to track the left and right path separately (Figure 5 (b)). This, in essence, doubles the scoreboard so that the two paths can execute concurrently with no interference. Second, we add a *shadow* bit to each scoreboard entry, which is used to ensure correct execution when diverging and reconverging. To explain how the shadow bits are used, we first introduce the issues with divergence and reconvergence. There are four scenarios that must be considered (Figure 6):

1. Unresolved pending writes before divergence (e.g., $r0$ on path A) should be visible to the other path (e.g., $r0$ on path C) after divergence, and further, both paths need to know when $r0$ is written back. Ignoring either aspect will lead to either incorrect execution or deadlock. If a register that is not yet ready is used, an incorrect result may be generated. Conversely, if a register is assumed pending and is never marked as ready, execution will deadlock.
2. Unresolved pending writes before reconvergence (e.g., $r7$ on path F) should be visible to the other path ($r7$ at path G) after reconvergence.
3. If a register number is the destination register of an instruction past a divergence point, then this register should not be confused with the same register number on the other path. Treating this as a false dependency between the paths may hurt performance but does not violate correctness (e.g., $r1$ on path B is a different register than $r1$ on paths D/E).
4. Similarly to the case above, if the register number on two different paths is a destination in both paths concurrently, then writes to this register number from the two paths should not be confused with one another. Thus, enforcing a false dependency is a poor design option because it will lead to the two paths being serialized as one path waits to issue until after the other path writes back.

Maintaining separate left and right scoreboards addresses the fourth scenario listed above and allows two independent paths to operate in parallel, but on its own cannot handle cross-path dependencies resulting from divergence and reconvergence. Our scoreboard design handles the first three cases conservatively by treating a pending write from before a divergence or reconvergence as pending on both paths after divergence/reconvergence, regardless of which path it originated in. This guarantees that no true dependency will be violated. To achieve this behavior, when a path diverges, the pending bits of its scoreboard are copied to its shadow bits. When querying the scoreboard for a register in one path, the shadow bits in the other path are also examined. If either the path's scoreboard or the shadow in the other scoreboard indicate a pending write, the path stalls (Figure 5 (b)). In our example, this mechanism ensures that path C correctly waits for the pending write of $r0$ from path A (Figure 7 (a–b)). Upon a writeback, both the shadow and pending bits of the original scoreboard of the instruction are cleared, freeing instructions on both paths to schedule (Figure 7 (c)). This requires propagating a single additional bit down the pipeline to indicate whether a writeback is to the left or right scoreboards. A similar procedure is followed for reconvergence to guarantee correct cross-path dependencies, as shown with the dependency on $r7$ from path F to path G (Figure 7 (d)).

At the same time, our design does not create dependencies between concurrent left and right paths. For example, after the divergence of BR_{D-E}, the shadow bits for $r4$ are not set, and thus, $r4$ is tracked independently in the left and right scoreboards. While this mechanism ensures correct execution and avoids serialization as described above, it may introduce false dependencies that partially stall execution. For example, the write to $r1$ on path B is unrelated to the reads of $r1$ on paths C and D. The shadow bit for $r1$ on the left scoreboard is set when the paths diverge at BR_{D-E}, which unnecessarily stalls the execution of blocks D and E. On the other hand, this false dependency also ensures that $r1$ generated in path B is written back before the dependent instruction in path G executes (Figure 7 (d)).

While a much more sophisticated scoreboard structure that can filter out such false dependencies can be designed, our experiments indicate it will provide little benefit of a maximum 1% performance improvement (Section 6.4). The cost of a non-conservative scoreboard, on the other hand, would be high because it would require more information to propagate in the pipeline and additional logic to decide when and when not to wait. Our proposed scoreboard is simple in both design and operation.

3.3 Warp Scheduler.

In the baseline SPE architecture, the warp scheduler chooses which of the ready-to-execute warps should be issued in the next cycle. Because of the large number of warps and execution units, some GPUs utilize multiple parallel schedulers with each scheduler responsible for a fixed subset of the total warps and also a subset of the execution lanes [22, 26]. For example, NVIDIA's Fermi GPU has two warp schedulers within each core (streaming multiprocessor); one scheduler for even-numbered warps and the other for odd-numbered warps with each scheduler responsible for scheduling an instruction for 16 of the 32 lanes within the core. DPE can expose up to twice the number of schedulable units as each warp can have both a left and a right path at the same time. We assume that the scheduler can be extended to support this greater parallelism by simply doubling the number of entries. Because each warp has two entries, a single additional selection level to choose which of the two entries competes with other ready warps is all that is required from the logic perspective.

In addition to this expanded scheduler that has twice as many schedulable entries, we also experiment with a constrained warp scheduler that maintains the same number of entries as SPE. In this constrained configuration, each warp is allocated a single entry and only one path, which is determined in the previous cycle, can be considered for scheduling at any time. In order to not lose scheduling opportunities when only one path is available, or when only one path is ready, we do not alternate between the paths on every cycle. Instead, we rotate which path will be available for scheduling whenever the current schedulable path encounters a cache miss or executes another long latency operation (e.g. a transcendental function).

3.4 Summary of the Benefits of DPE

As described above, the dual-path execution model extends current GPU designs with greater parallelism at very low cost. It requires no change to the underlying execution model and does not sacrifice SIMD efficiency. The extension to the stack itself is simple and only requires small modifications to existing structures. The warp scheduler also requires only a straightforward extension to support the greater level of parallelism exposed. The most significant change is to the scoreboard, and we show how to extend the baseline scoreboard to support two paths in a cost-effective manner. While our solution amounts to replicating the scoreboard structure, it does not add significant complexity because the left and right scoreboard do not directly interact and the critical path of the dependency-tracking mechanism is only extended by a multiplexer to select the pending or shadow bits and the OR-gate shown in Figure 5 (b). In the following sections we demonstrate the advantages of the extra parallelism over single-path execution, as well as added robustness and performance compared to DWS, an earlier attempt at increasing path parallelism.

Table 1: GPGPU-Sim configuration.

Number of SMs	15
Threads per SM	1536
Threads per warp	32
SIMD lane width	32
Registers per SM	32768
Shared memory per SM	48KB
Number of warp schedulers	2
Warp scheduling policy	Round-Robin
L1 cache (size/associativity/line size)	16KB/4-way/128B
L2 cache (size/associativity/line size)	768KB/8-way/256B
Number of memory channels	6
Memory bandwidth	29.6 GB/s per channel
Memory controller	Out-of-order (FR-FCFS)

4 Methodology

We model the microarchitectural components of dual-path execution using GPGPU-Sim (version 3.1.0) [1, 5], which is a detailed cycle-based performance simulator of a general purpose GPU architecture supporting CUDA version 3.1 and its PTX ISA. In addition to the baseline scoreboard provided as a default with GPGPU-Sim, we model the conservative scoreboard and the warp scheduler that can schedule both the left and right paths arbitrarily. We also implemented an optimistic scoreboard that does not add any false dependencies and a constrained warp scheduler that alternates between the left and right path of each warp (all four mechanisms described in Section 3). DWS with PC-based reconvergence has been implemented and simulated as described in Section 2.2 and by Meng et al. [20]. We do not constrain DWS resources and model its warp scheduler and scoreboard as perfect; i.e., there are enough scoreboard resources to track an arbitrary number of warp splits, no false dependencies are introduced, and any number of warp splits can be scheduled together with no restriction. Because DWS is sensitive to the heuristic guiding subdivision, we simulated DWS with a range of subdivision threshold values. In general, the simulator is configured to be similar to NVIDIA's Fermi architecture using the configuration file provided with GPGPU-Sim [2]. The key parameters used are summarized in Table 1 and we explicitly mention when deviating from these parameters for sensitivity analysis.

5 Benchmarks

We use 27 benchmarks selected from Rodinia [7], Parboil [16], CUDA-SDK [23], the benchmarks provided with GPGPU-Sim [5], a few applications from CUDA Zone [25] that we found to work with GPGPU-Sim, and [15]. The benchmarks studied are ones whose kernel can execute to completion on GPGPU-Sim. We report the results of the first 5 iterations of the kernel of MCML (each iteration results in IPC with near zero variation among different iterations) due to its large simulation time. Note that we only report the detailed results for the 14 benchmarks shown in Table 2, because the other 13 benchmarks execute in an identi-

Table 2: Simulated Benchmarks.

Interleavable			
Name	**Description**	**#Instr.**	**Ref.**
LUD	LU Decomposition	39M	[7]
QSort	Quick Sort	60M	[25]
Stencil	3D Stencil Operation	115M	[16]
RAY	Ray Tracing	250M	[5]
LPS	Laplace Solver	72M	[5]
MUMpp	MUMmerGPU++	148M	[13]
MCML	Monte Carlo for ML Media	303B	[15]
Non-interleavable			
Name	**Description**	**#Instr.**	**Ref.**
DXTC	DXT Compression	18B	[23]
BFS	Breadth-First Search	16M	[5]
PathFind	Path Finder	639M	[7]
NW	Needleman-Wunsch	51M	[7]
HOTSPOT	Hot-Spot	110M	[7]
BFS2	Breadth-First Search 2	26M	[7]
BACKP	Back Propagation	190M	[7]

cal way with SPE, DPE, and DWS, as represented by DXTC and BACKP. The reason for the identical behavior is that the structure of the control flow in these kernels does not expose any added parallelism with DPE and that the heuristic that guides DWS always results in no warp splits. Within the 14 benchmarks we discuss, half do not benefit from DPE because the branch structure does not result in distinct left and right paths that can be interleaved (categorized as non-interleavable in Table 2). We discuss this further in the next section. Note that these benchmarks are impacted by DWS and we evaluate their behavior with DWS.

6 Experimental Results

In this section we detail the evaluation of the dual-path execution model, including its impact on TLP, resource utilization, number of idle cycles, overall performance, sensitivity to key parameters, and implementation overheads.

6.1 TLP and Lane Utilization

The goal of DPE is to increase the parallelism available to the warp scheduler by allowing both the taken and non-taken paths of a branch to be interleaved. Not all divergent branches, however, are *interleavable* because many branches have only an *if* clause with no *else*. With such branches, the reconvergence point and the fall-through point are the same and the non-taken path is empty. BFS, for instance, is known to be highly irregular (average SIMD lane utilization of only 32%) with significant portion of its branches diverging. All its divergent branches, however, are ones with only *if* and no *else* clause (Figure 8), which leads to all threads in $Path_R$ arriving at the reconvergence point immediately: threads branching into block E at BR_{B-E} and ones branching into block D at BR_{C-D} all have their next PC equal to RPC upon divergence and are inactivated until threads in the other path reconverge. We refer to such divergent branches as *non-interleavable*, and to branches that result in both non-empty left and right paths as *interleavable* (Figure 9). A benchmark often contains a mix of interleavable and non-interleavable branches, and only the interleavable ones have potential for interleaving with DPE. We therefore use Avg_{Path} (Equation 1) to quantify each benchmark's potential for interleaving, where *N* is the

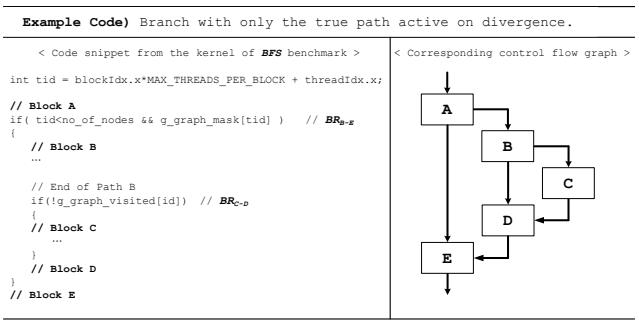


Figure 8: Example of non-interleavable branches.

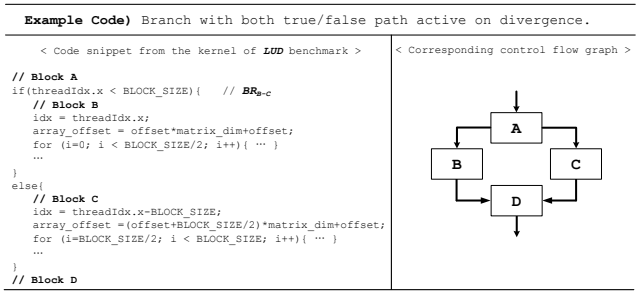


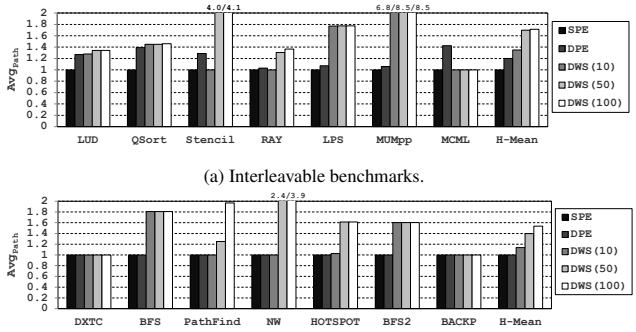
Figure 9: Example of an interleavable branch.

total number of warp instructions issued throughout the execution of the kernel and $NumPath_i$ is the *total number of concurrently schedulable paths available within the issued warp* when the *i-th* warp is issued.

$$Avg_{Path} = \frac{\sum_{i=1}^{N} NumPath_i}{N} \quad (1)$$

SPE, which can only schedule the single path at the TOS, always has $NumPath_i$ equal to 1, and hence, has $Avg_{path} = 1$ for all benchmarks. DPE, on the other hand, has $Avg_{path} > 1$ for interleavable benchmarks, as $NumPath_i$ equals 2 when an interleavable branch, which generates both $Path_L$ and $Path_R$ at the TOS, is executed. Note that Avg_{Path} is 1 with DPE as well when all the divergent branches within the benchmark are non-interleavable. When DWS is used, $NumPath_i$ equals 1 when the warp is scheduled in non-subdivided mode as it uses the conventional stack to serialize execution. When a warp is subdivided, however, $NumPath_i$ is equal to the total number of valid entries (hence the number of valid warp-splits) in the WST. Accordingly, non-interleavable benchmarks *can* have an Avg_{Path} value larger than 1 when DWS is used.

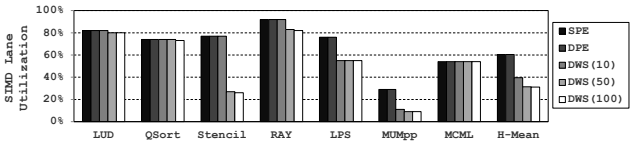
Figure 10 shows Avg_{Path} for all 14 benchmarks with three different subdivision thresholds, with DWS_{10} being the most conservative about subdividing warps and DWS_{100} the most aggressive. Overall, both DPE and DWS are able to achieve significant increases in Avg_{Path} value across the interleavable benchmarks (an average increase of 20% and 71% for DPE and DWS_{100}, respectively), thereby exposing more TLP for the warp scheduler. DWS_{100} and DWS_{50} expose significantly more TLP than DPE and also increase TLP for non-interleavable benchmarks. As discussed in Section 3, the improvement in TLP with DWS comes at



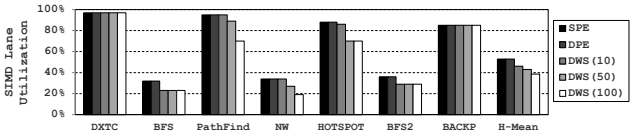
(a) Interleavable benchmarks.

(b) Non-interleavable benchmarks.

Figure 10: Average number of concurrently schedulable paths per warp.



(a) Interleavable benchmarks.



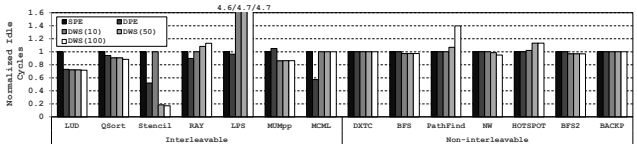
(b) Non-interleavable benchmarks.

Figure 11: SIMD Lane Utilization.

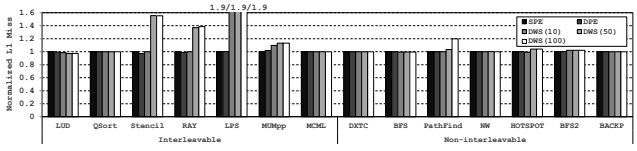
the cost of decreased SIMD utilization. Figure 11 summarizes the average SIMD lane utilization achieved across these benchmarks. While, as expected, DPE shows no loss in SIMD lane utilization across all benchmarks, DWS sacrifices a large fraction of SIMD utilization in many cases. With the exception of LUD, QSort, MCML, DXTC, and BACKP, DWS_{50} and DWS_{100} reduce SIMD efficiency by a large amount for all benchmarks: an average 48.1%/48.5% loss for interleavable benchmarks and 18.6%/27.1% loss for non-interleavable ones, respectively. This implies that subdivision was performed far too aggressively, sacrificing efficiency for increased TLP. In other words, DWS_{50} and DWS_{100}'s high Avg_{Path} is obtained at the cost of having basic blocks that would have been executed once (when using the reconvergence stack of SPE or DPE) to instead execute as many warp-splits. DWS_{10} loses less SIMD efficiency because it splits fewer warps, but SIMD utilization is still significantly decreased (24.6% on average overall) and the conservative heuristic fails to improve TLP in some cases.

6.2 Idle Cycles and Impact on Memory Subsystem

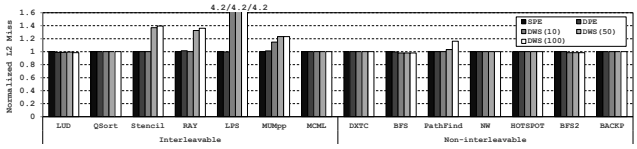
Figure 12 illustrates the impact the different schemes have on the number of idle cycles and L1/L2 cache misses. Overall, DPE is able to reduce the number of idle cycles by an average of 19% for interleavable benchmarks while matching SPE with non-interleavable ones. The only exception is MUMpp where the interleaving of diverging paths disrupts the access pattern to the L1 cache and increases the



(a) Number of idle cycles accumulated throughout all cores.



(b) Number of L1 misses.



(c) Number of L2 misses.

Figure 12: Changes in idle cycles and L1/L2 cache misses when using different mechanisms (all normalized to SPE).

number of misses by 2% and idle cycles by 5%.

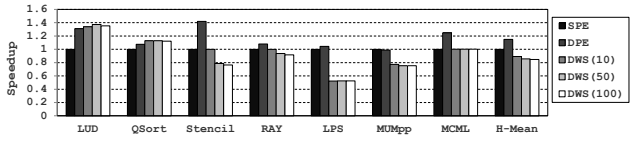
DWS, in general, can decrease idle cycles because it significantly increases Avg_{Path} to enable better interleaving. However, the significantly lower SIMD utilization achieved with DWS, makes a comparison of idle cycles between DWS and DPE meaningless. While the GPU executes instructions on more cycles, additional cycles are required to execute the many warp-splits of DWS. LUD and QSort can be directly compared because they have similar SIMD utilization with DWS and DPE, and also have very similar total idle cycles.

Counter intuitively, RAY, LPS, PathFind, and HOTSPOT, which have significant improvements of TLP with DWS, suffer from many more idle cycles compared to SPE. The reason for this behavior is that the many interleaved warp-splits present a memory access pattern that performs poorly with the cache hierarchy. As shown in Figure 12 (b–c), these four benchmarks have increased miss rates in both L1 and L2. The added TLP is not sufficient to counter-weigh the added memory latency. Stencil and MUMpp also suffer from worse caching with DWS, but have high-enough TLP to still reduce the number of idle cycles.

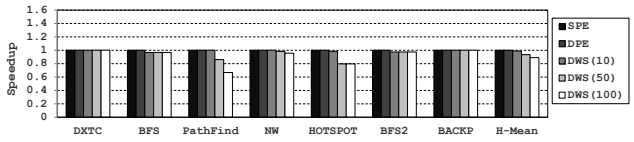
6.3 Overall Performance

Figure 13 shows the overall performance of DPE compared to that of SPE and DWS. Except fot MUMpp, whose IPC is degraded by 1.1%, due to its increased L1 miss rate, DPE provides an improvement in performance across all the interleavable workloads (14.9% on average) while never degrading the performance of non-interleavable ones.

DWS is able to obtain significant IPC improvement for LUD and QSort (37.3%/12.8% increase over SPE and 5.2%/4.9% over DPE when using DWS_{50}), thanks to the significant increase in Avg_{Path} while maintaining similar SIMD lane utilization. The other 12 benchmarks, however, fail to balance Avg_{Path} and SIMD lane utilization and either suffer from degraded performance due to excessive subdi-

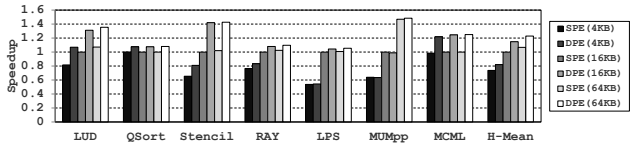


(a) Speedup over SPE among interleavable benchmarks.

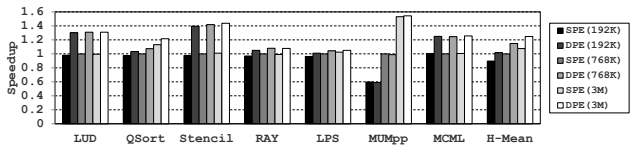


(b) Speedup over SPE among non-interleavable benchmarks.

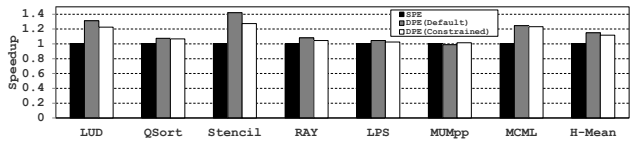
Figure 13: Performance of the DPE model compared to SPE and DWS (all normalized to SPE).



(a) Performance sensitivity to different L1 cache size (normalized to SPE(16KB)).



(b) Performance sensitivity to different L2 cache size (normalized to SPE(768K)).



(c) Performance sensitivity when warp scheduler has limited context resources (normalized to SPE).

Figure 14: Performance sensitivity to cache size and warp scheduler visibility.

vision or do not subdivide at all despite having potential for interleaving (MCML). It is interesting to observe that despite large increase in Avg_{Path} achieved with DWS, the significant loss in SIMD lane utilization always outweighed the benefits of increased interleaving. This is mainly because the increase in Avg_{Path} (and hence increased interleaving capability) is only beneficial upto the point where there exists any latency to hide, after which the loss in SIMD lane utilization is too severe. GPUs are designed to tolerate high latency, so this is, in fact, expected behavior.

6.4 Sensitivity Study

Figure 14 (a–b) shows the speedup of DPE over SPE for the 7 interleavable benchmarks with different cache sizes. With smaller cache, we could expect a higher need for hiding latencies because of the larger fraction of long-latency memory operations. Overall, the relative IPC improvement remains stable within $\pm 2\%$ when varying the size of the L2 cache and $\pm 4\%$ for L1 cache size variation, with the exception of Stencil. When the L1 cache is reduced to 4KB, Stencil becomes much more memory bound, which results in a significant increase of idle time. In this case, while DPE can still reduces idle cycles by the same absolute number of cycles, the relative improvement is smaller because the overall execution time is so large.

As discussed in Section 3, we also tested DPE with a more constrained scheduler and with a scoreboard that does not introduce false dependencies. The more aggressive scoreboard improved performance by at most 1% and we do not show the results for brevity. *DPE (Constrained)* in Figure 14 (c) can only track a single path's context per warp so the schedulable path is rotated (between $Path_L$ and $Path_R$ after a long-latency instruction executes). Overall, speed-up is reduced from 14.9% to 11.7% for the constrained mode of the warp scheduler.

6.5 Implementation Overhead

As discussed in Section 3, implementing DPE requires modifications to the reconvergence stack, scoreboard, and the scheduler, as well as a minimal extension to propagate a single bit to indicate whether an instruction originated from the left or the right path. We estimate that using the dual-path stack has negligible overhead compared to the single-path stack. Each stack entry with DPE requires 160 bits to store the PC and mask of each path (32-bits each per path and the single RPC (32 bits)). While this is more bits per entry compared to the SPE stack, which requires 96 bits, fewer stack entries are needed to represent the same number of paths. The maximum stack depth observed with the 14 benchmarks we evaluated in detail was 11 with SPE and 7 with DPE, with a very similar overall size of structure.

The DPE scoreboard requires independent left and right scoreboards, the addition of the single shadow bit to each entry, and logic for setting the shadow bits and selecting whether the pending or shadow bit should be used when querying. The additional logic is very simple and should have minimal overhead. The extra shadow bit accounts for $7 - 14\%$ of the scoreboard storage, depending on the maximum number of registers per thread, which increased from 64 to 256 between NVIDIA's Fermi [22] and Kepler [26] GPUs. Maintaining the information for the two paths roughly doubles the cost of the scoreboard in area and power. While the scoreboard is significantly more expensive, the logic and paths surrounding the scoreboard are unchanged, and logic and layout optimizations can reduce the overall cost. Note that a single scoreboard is used for all SIMD lanes, amortizing its cost. The warp scheduler hardware also roughly doubles in size because decoded instructions from both left and right paths require instruction-buffer storage. Like the scoreboard, the scheduler is amortized across all lanes.

7 Discussion

7.1 Path-Forwarding

DPE exposes two paths for scheduling when the TOS entry has both a left and a right path. When one of these two paths reconverges and the other is still active, only a single path is available for scheduling. However, in some cases an independent path, which can be concurrently scheduled with the current active path, may exist in entries below the

TOS. In the example shown in Figure 4 (d), interleaving block B and block E does not break correctness, but is not done because block B is not at the TOS. A possible optimization of DPE to such issue is to *forward* the information from a lower stack entry up (including its RPC) when a slot at the TOS is available. We omit the details of how this forwarding can be achieved with reasonable logic circuits, because our evaluation of this forwarding technique indicates a small overall potential for improvement. We observed a maximum of $< 2\%$ performance improvement on the 7 interleavable benchmarks. The interleavable benchmarks tended to have a shallow stack and balanced branches, which limits the opportunities for forwarding. While proven ineffective in our benchmarks, path-forwarding will provide significant benefits when the taken and non-taken paths are not balanced with one path executing significantly longer than the other.

7.2 DPE for Memory Divergence

While DWS increases TLP when branches diverge, an important benefit it can provide is to increase memory-level parallelism when some threads in a warp experience a cache miss while others hit in the cache. When such a case occurs, the warp can be split into two groups of threads: those that completed the load and continue to execute and those that must wait for main memory to supply the value. DPE hardware can also be used to increase TLP in such cases by utilizing the left and right slots for the mask of those threads that completed the loads and those that did not. DPE is not as flexible as DWS because the stack must still correctly reflect control flow reconvergence, which means the memory-split warp must wait at the first divergent branch or when the RPC is reached. We leave the evaluation and optimization of this use of DPE for future work.

7.3 DPE with a Software-Managed Reconvergence Stack

The baseline GPU architecture we used to explain DPE uses an explicit hardware reconvergence stack, which maintains the PC, mask, and RPC. One alternative architecture is to maintain only the PC and mask in hardware and control when entries are pushed and popped with explicit software instructions [4]. Applying DPE to this design is straightforward. The only change needed is that a pop instruction only actually pop the stack if the other path is not active; if both paths are active, the first pop instruction disables its path and the second pops the stack. Some current GPUs, such as the GPU of the Intel Sandy Bridge Processor [17], have an entirely implicit stack. Hardware maintains an explicit PC for each thread and dynamically computes predicate masks based on a software-managed warp-wide PC. To support DPE, two warp-wide PCs are required and we leave the details of what the software algorithm required to do so might be to future work.

8 Related Work

As discussed in Section 2, DWS closely relates to DPE. In addition to the simplicity, robustness, and performance advantages we show for DPE over DWS, it is important to note that DWS was developed and evaluated before the scoreboard mechanism of GPUs and the ability to issue back-to-back instructions was widely understood. Therefore, DWS ignored this critical design aspect and did not discuss the implications. We do not attempt to remedy this in this paper and simply evaluate DWS as if ideal scoreboard and scheduling hardware existed.

Another closely related project, the *dual instruction multiple thread* (DIMT) execution model has recently been presented [6, 14]. DIMT can issue two different instructions to the SIMD pipeline at the same time by expanding the instruction broadcast network, the register-file structure, and others. Brunie et al. [6] explored the microarchitectural aspects of adopting the DIMT concept to GPGPU architectures. Their DIMT-based architecture is conceptually similar to DPE in that a maximum of two concurrent paths are chosen for scheduling. Also like DPE, the scoreboard and scheduler are enhanced to track the larger number of schedulable units. Unlike DPE, DIMT does not work with the stack model. Instead, the more complex model of *thread frontiers* [10] serves as the baseline architecture. Support for thread frontiers requires significant changes to the hardware, including explicit tracking of per-thread PCs, new instructions, compiler support, and a hardware-managed heap structure that takes the place of the simple reconvergence stack. In addition, DIMT introduces a complex scoreboard design with significant additional storage, and new logic functionality. While thread frontiers have advantages over the stack model for applications that make heavy use of unstructured control flow, they do present a more complex design point. DPE, in contrast, integrates smoothly with current execution models and designs and extends the reconvergence stack rather than replacing it. To the best of our knowledge, ours is the first microarchitecture that is able to utilize intra-warp parallelism of this type with a reconvergence stack. DWS uses the stack only until warps are split and then abandons the design until a warp is merged again, and DIMT assumes the heap-based threads frontier model. In fact, Brunie et al. [6] explicitly state that a motivation for adopting thread frontiers in their design is that intra-warp TLP is very challenging with the stack model.

In addition to these closely-related research projects, we briefly summarize additional related work below. Tarjan et al. [30] proposed *adaptive slip* mechanism to address memory divergence issues, where a subset of a warp is enabled to continue executing while other threads are waiting on memory. This work was superseded by DWS [20], tackling both control and memory divergence, which we directly compare with our dual-path execution model. Diamos et al. [10] suggested *thread frontiers* as an alternative mechanism to the immediate post-dominator reconvergence algorithm, which does not guarantee the earliest reconvergence point in an unstructured control flow [31]. Rogers et al. [28] recently introduced a *cache-conscious* warp scheduling mechanism that dynamically adjusts the level of thread-level parallelism such that thrashing in the cache is minimized.

9 Conclusions

In this paper we explain the potential for utilizing the intra-warp parallelism resulting from diverging structured

control flow to improve SIMD efficiency and overall performance. DPE is the first mechanism that maintains the elegant control-flow execution of the GPU reconvergence stack, yet is able to exploit intra-warp parallelism. Unlike prior approaches to this issue, DPE does not require an extensive redesign of the microarchitectural components, and instead extends the stack to support two concurrent execution paths. The scoreboard and scheduler must also be enhanced, and we show how this can be done relying mostly on replicating current structures rather than adopting a completely new model. We show that the combination of these spot-enhancements can provide significant efficiency and performance benefits and never degrades performance compared to the baseline GPU architecture.

We analyze a range of architectures and benchmarks and find that a significant fraction of the benchmarks we were able to run on the simulator ($\frac{7}{27}$) can benefit from DPE. The maximum speedup across these benchmarks is 42% with an average of 14.9%. We also discuss potential improvements to our design with more aggressive and less constrained hardware. We do not evaluate these in detail because the potential high cost and complexity of these modifications yields little performance improvement. The reason is that, with DPE, the additional latency hiding capability is already significant, and additional minor increases are insignificant, improving performance by no more than an additional 2%.

Acknowledgements

We would like to thank the developers of GPGPU-Sim and the anonymous reviewers, who provided excellent feedback for preparing the final version of this paper.

References

[1] GPGPU-Sim. http://www.gpgpu-sim.org.

[2] GPGPU-Sim Manual. http://www.gpgpu-sim.org/manual.

[3] AMD Corporation. AMD Radeon HD 6900M Series Specifications, 2010.

[4] AMD Corporation. R700-Family Instruction Set Architecture, February 2011.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2009)*, April 2009.

[6] N. Brunie, S. Collange, and G. Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC-2009)*, October 2009.

[8] Collange, Sylvain. Stack-less SIMT Reconvergence At Low Cost, 2011.

[9] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Siu. Tracking Register Usage During Multithreaded Processing Using a Scoreboard Having Separate Memory Regions And Storing Sequential Register Size Indicators. In *US Patent 7434032*, October 2008.

[10] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD Re-Convergence At Thread Frontiers. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[11] W. W. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *17th International Symposium on High Performance Computer Architecture (HPCA-17)*, February 2011.

[12] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[13] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC-2010)*, November 2010.

[14] Glew, A. Coherent Vector Lane Threading. In *Berkeley Parlab Seminar*, 2009.

[15] T. D. Han and T. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *4-th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, March 2011.

[16] IMPACT Research Group. The Parboil Benchmark Suite. http://www.crhc.uiuc.edu/IMPACT/parboil.php, 2007.

[17] Intel Corporation. Intel HD Graphics OpenSource Programmer Reference Manual, June 2011.

[18] A. Levinthal and T. Porter. Chap - A SIMD Graphics Processor. In *11th Anual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'84)*, July 1984.

[19] R. A. Lorie and H. R. Strong. Method For Conditional Branch Execution in SIMD Vector Processors. In *US Patent 4435758*, March 1984.

[20] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *37th International Symposium on Computer Architecture (ISCA-37)*, 2010.

[21] V. Narasiman, C. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *44th International Symposium on Microarchitecture (MICRO-44)*, December 2011.

[22] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[23] NVIDIA Corporation. CUDA C/C++ SDK CODE Samples, 2011.

[24] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011.

[25] NVIDIA Corporation. NVIDIA CUDA Zone, 2012.

[26] NVIDIA Corporation. Whitepaper: NVIDIA GeForce GTX 680, 2012.

[27] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *39th International Symposium on Computer Architecture (ISCA-39)*, June 2012.

[28] T. Rogers, M. O'Connor, and T. Aamodt. Cache-Conscious Wavefront Scheduling. In *45th International Symposium on Microarchitecture (MICRO-45)*, December 2012.

[29] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[30] D. Tarjan, J. Meng, and K. Skadron. Increasing Memory Miss Tolerance for SIMD Cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC-09)*, 2009.

[31] H. Wu, G. Diamos, S. Li, and S. Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. In *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, June 2011.