SIMD Re-Convergence At Thread Frontiers

Gregory Diamos
Georgia Institute of Technology
School of ECE
Atlanta, GA, USA
gregory.diamos@gatech.edu

Benjamin Ashbaugh
Intel
Visual Computing Group
Folsom, CA
bashbaugh@intel.com

Subramaniam Maiyuran
Intel
Visual Computing Group
Folsom, CA
subramaniam.maiyuran@intel.com

Andrew Kerr
Georgia Institute of Technology
School of ECE
Atlanta, GA, USA
arkerr@gatech.edu

Haicheng Wu
Georgia Institute of Technology
School of ECE
Atlanta, GA, USA
hwu36@gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
School of ECE
Atlanta, GA, USA
sudha@ece.gatech.edu

ABSTRACT

Hardware and compiler techniques for mapping data-parallel programs with divergent control flow to SIMD architectures have recently enabled the emergence of new GPGPU programming models such as CUDA, OpenCL, and DirectX Compute. The impact of branch divergence can be quite different depending upon whether the program's control flow is structured or unstructured. In this paper, we show that unstructured control flow occurs frequently in applications and can lead to significant code expansion when executed using existing approaches for handling branch divergence.

This paper proposes a new technique for automatically mapping arbitrary control flow onto SIMD processors that relies on a concept of a *Thread Frontier*, which is a bounded region of the program containing all threads that have branched away from the current warp. This technique is evaluated on a GPU emulator configured to model i) a commodity GPU (Intel Sandybridge), and ii) custom hardware support not realized in current GPU architectures. It is shown that this new technique performs identically to the best existing method for structured control flow, and re-converges at the earliest possible point when executing unstructured control flow. This leads to i) between $1.5 - 633.2\%$ reductions in dynamic instruction counts for several real applications, ii) simplification of the compilation process, and iii) ability to efficiently add high level unstructured programming constructs (e.g., exceptions) to existing data-parallel languages.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*processors, compilers*; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*multiple data stream architectures*

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil.

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

The transition to many core computing has coincided with the growth of data parallel computation and the evolution of graphics processing units (GPUs) from special purpose devices to programmable cores. The emergence of low cost programmable GPU computing substrates from NVIDIA, Intel, and AMD have made data parallel architectures a commodity from embedded systems through large scale clusters such as the Tsubame [1] and Keeneland systems [2] hosting thousands of GPU chips.

The dominant programming model involves the use of bulk-synchronous-parallel computing models [3] embodied by languages such as CUDA, OpenCL, and DirectX Compute. These data-parallel languages implement *single instruction stream multiple thread* (SIMT) models of computation that specify a large number of data-parallel threads that can be readily exploited by hardware multi-threading and *single-instruction-multiple-data* (SIMD) cores. Modern GPU architectures have demonstrated support for efficient, automatic mappings of arbitrary control flow onto SIMD processors, and in fact these languages rely on it to retain many of the control flow abstractions found in modern high level languages and simplify the task of programming SIMD processors. Performance is maximized when all of the threads mapped to the same SIMD unit (henceforth *warp*) take the same path through the program. However, when threads in a warp do diverge on a branch, i.e., different threads take different paths, performance suffers. This is referred to as *branch divergence* and its behavior is critical to SIMD processor performance.

Previous work by Wu et al. [4], assessed the impact of unstructured control flow in existing applications and commonly used benchmarks. They found that unstructured control flow occurs quite often (in 40% of the Parboil, Rodinia and Optix benchmarks) in applications and is either introduced by the programmers (e.g. using goto statements) or by the compiler optimizations (e.g., function inlining, short-circuit, etc). Usually, the more complex the program, the

more likely the existence of unstructured control flow. They investigated techniques for converting unstructured control flow to structured control flow and found that these techniques can substantially increase static and dynamic code size. Further, the analysis reported in this paper found that the dynamic code expansion in the presence of unstructured control flow (with no transformation) could be substantial (see Section 6). Finally, we note that several desirable language features (e.g., exceptions) naturally lead to unstructured control flow. We show that this commonly leads to performance degradation, even if exceptions are not encountered at runtime. Consequently, we argue that support for unstructured control flow is justified by its frequency of occurrence, performance impact, and the functionality it enables.

This paper proposes a novel scheme - *thread frontiers* - to address branch divergence in SIMD processors. For each basic block, the thread frontier of that block is informally defined as the set of other basic blocks where all other diverged threads **may** be executing. At runtime, if a thread is executing basic block BB, all other threads from the same warp **will** be found in the thread frontier of BB. The principle contribution of thread frontiers is its ability to efficiently handle unstructured control flow in SIMD processors. Threads in a warp are permitted to re-converge earlier than achieved in existing schemes thereby eliminating code expansion due to branch divergence. Thread frontiers can in fact be implemented on some modern GPUs. We report on the performance from modeling this implementation as well as simulation results of a proposed native implementation. Our evaluation has considered 8 existing highly tuned applications and 5 microkernels representing (desirable) language features not typically supported in GPUs . Specifically, this paper makes the following contributions.

- We show how unstructured control flow causes dynamic and static code expansion (1.5%-633%) for existing hardware and compiler schemes for handling control flow on SIMD processors. We show that this code expansion can severely degrade performance in cases that do occur in real applications.
- To handle unstructured control flow effectively, we introduce a new concept of a **thread frontier** that can be used conservatively by a compiler or optimistically with hardware support to identify the earliest re-converge point of any divergent branch, thereby avoiding **any** code expansion when executing programs with unstructured control flow.
- We describe how thread frontiers can be implemented using existing hardware on a modern GPU - Intel Sandybridge - and provide an empirical evaluation of performance using a Sandybridge emulator.
- We evaluate the performance of thread frontiers using native hardware support modeled using extensions to a GPU emulator.
- We show how support for thread frontiers can make it acceptable, from a performance perspective, to support language features such as divergent function calls and exceptions in SIMD processors. These features produce unstructured control flow.

1.1 Organization

The remainder of the paper first introduces existing reconvergence mechanisms, and then thread frontiers by way of an illustrative example. The following sections describe the compiler and hardware support. The paper concludes with a description of the results of the evaluation of a benchmark suite of CUDA applications with unstructured control flow, lessons learned and directions for future research.

1.2 Terminology

Terms used to describe GPU abstractions such as data-parallel threads and groups of threads that are mapped to a SIMD unit vary in the literature depending on the specific GPU being considered. The NVIDIA terminology is adopted in this paper to maintain consistency with related work. A warp refers to the threads executing on a SIMD processor - one thread per SIMD lane. When reading Intel documentation, it is necessary to use the term `channel` to refer to an NVIDIA `thread`, and `thread` to refer to an NVIDIA `warp`. This means that the `per-thread program counters` (PTPCs) used in this paper are referred to as `per-channel instruction pointers` (PCIPs) by Intel.

2. RE-CONVERGENCE MECHANISMS

2.1 Predicate Stack

In 1982 the CHAP [5] graphics processor introduced the concept of a stack of predicate registers combined with explicit `if-else-endif` and `do-while` instructions to transparently enable the execution of structured control flow on a SIMD processor. In their work concerning dynamic warp formation, Fung et al [6] also describe a technique that they refer to as immediate post-dominator re-convergence (PDOM), which extends the concept of a predicate stack to support programs with arbitrary control flow using **branch** and **re-converge** instructions. This is done by finding the immediate post-dominator for all potentially divergent branches and inserting explicit re-converge instructions. When a divergent branch is executed, a predicate mask and PC is computed for each unique branch target and pushed onto a stack. Execution continues using the top entry on the stack, and the stack is popped when a re-converge point is hit. In order to resume execution after all paths have reached the post-dominator, the program counter of the warp executing the branch is adjusted to the instruction immediately after the re-converge point before the first entry is pushed onto the stack.

2.2 Per-Thread Program Counters (PTPCs)

Although previous generations of Intel GPUs have used the predicate stack mechanism for control flow, a new mechanism was introduced starting with the Sandybridge processor. According to the ISA reference manual [7], the mechanism changes by removing the predicate stack and instead introducing a program counter (PC) for each thread along with the main PC for each warp. When an instruction is issued the per-thread PC is checked against the warp PC. If they match, that thread is enabled and its PC is advanced along with the warp PC. Otherwise, it is disabled and it is not modified. The warp PC is always modified, either by incrementing to the next instruction in the basic block or by updating it according to the outcome of branch instructions. This is discussed in greater detail in Section 5.

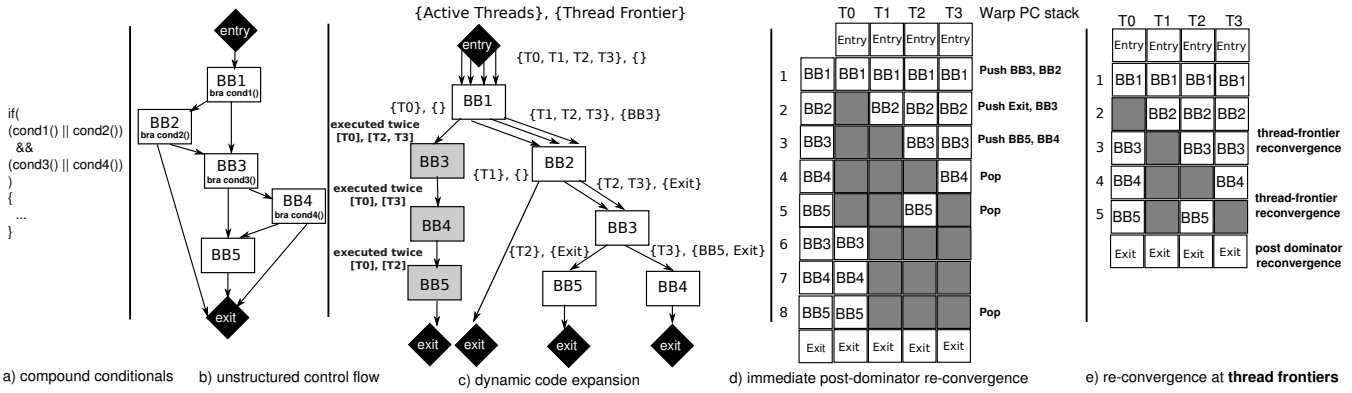

Figure 1: An example of an application with unstructured control flow leading to dynamic code expansion.

The ISA is augmented with explicit `if-else-endif` and `do-while` instructions that modify the per-thread PCs accordingly to handle control flow divergence. For example, `if-else-endif` is implemented by advancing threads that evaluate the `if`-condition to be False to the first instruction of the `else` block. If all threads branch then the warp PC is advanced immediately to the `else` block, skipping the `then` block. However, if the branch is divergent, the remaining threads (along with the warp PC) fall through and execute the body of the statement until they reach an explicit `else` instruction. This instruction advances the PTPCs of the active threads to the first instruction after the `end-if`. The warp PC then advances to the first instruction of the `else` block and any waiting threads are picked up. Eventually, they will reach the `end-if` and find the disabled threads waiting there. At that time, the PCs of all threads will match the warp PC and re-convergence will occur.

3. POTENTIAL OF THREAD FRONTIERS

This section presents an example that illustrates the potential of re-convergence at thread frontiers compared to existing schemes. Consider the C sourcecode shown in Figure 1 (a) and the corresponding program control flow graph (CFG) shown in Figure 1 (b). This code segment represents an example of unstructured control flow that can occur due to compiler optimizations. It is possible for threads in a warp to take different paths through the program resulting in some serialization of thread execution on a SIMD processor and increase in dynamic instruction count.

In the context of SIMD processors, threads in a warp that take different (divergent) paths through the code must always be joined (re-converged). The scheme that is currently used by the majority of commodity GPUs for determining re-convergence points for divergent SIMD threads is referred to as immediate post-dominator (PDOM) re-convergence [6]. Informally, the immediate post-dominator of a branch in a control flow graph (CFG) is the node through which all paths from the branch pass through and which does not post-dominate any other post dominators. Thus, they present natural opportunities for re-converging threads. In our example, with PDOM all threads that enter at the *Entry* node are re-converged at the *Exit* node and have a execution schedule shown in Figure 1 (d). This can lead to inefficiencies as described below, and explored in greater detail by Wu et al. [4].

Consider a warp with four threads executing this code segment. Consider the specific case where threads T0, T1, T2, and T3 take execution paths (BB1, BB3, BB4, BB5), (BB1, BB2), (BB1, BB2, BB3, BB5), and (BB1, BB2, BB3, BB4) respectively. On entry all threads in the warp execute BB1 which ends in a divergent branch. Threads T1, T2, and T3 (designated [T1, T2, T3]) branch to BB2 while thread T0 ([T0]) branches to BB3. Note that under existing schemes, these threads *will not re-converge* until the Exit node, which is the immediate post-dominator. Now, assume the thread scheduler selects BB2 for execution [1]. Thread [T0] is disabled while [T1, T2, T3] are active; they execute BB2 to reach the divergent branch at the end of BB2. Now we have [T1] branching to Exit while [T2, T3] branch to BB3. Assume that the thread schedule selects BB3 to be executed by [T2, T3]. Note that at this time, thread [T0] is still (also) disabled waiting to execute BB3. Under PDOM [T0] will be enabled and executed at some future point in time. Thus BB3 would be executed twice - first for threads [T2, T3] and then for [T0]. Under thread frontiers, threads [T2, T3], and [T0] can be re-converged immediately and executed. Hence BB3 is fetched and executed only once.

In general, over time the warp will have a pool of disabled threads waiting to be enabled and active threads executing a basic block. The key observation here is that in the presence of unstructured control flow, divergent paths pass through common basic blocks (BB3 in our example). Code expansion occurs if they re-converge at a later block. Here, the threads [T2, T3] and [T0] will execute at different points in time depending on the thread scheduler and therefore BB3 will be executed twice. Note the schedule of execution in Figure 1 (d) for PDOM where blocks BB3, BB4 and BB5 are fetched and executed twice at two different points in time. The PDOM approach suffers from such code expansion in the presence of unstructured control flow.

At any point in time the set of all basic blocks where divergent threads may be executing is referred to as the current *thread frontier*. In this example, and at the point described above when [T2, T3] are about to execute BB3, the thread frontier is comprised of blocks BB3 ([T0]), Exit ([T1]), and BB3 ([T2, T3]). The key insight is that if we keep track of the thread frontier, we can re-converge threads at the earliest point in time. In this example, [T0] could be re-converged with warp [T2, T3] and the new warp [T0, T2, T3] can enter BB3. Block BB3 would only be executed once.

[1] In most implementations of PDOM [6], the next PC is chosen from the top entry in the predicate stack.

```
Input: Set of basic blocks sorted from high to low
       priority
Output: Mapping from basic block to thread frontier
1 tset := {};
2 foreach basic block b in sorted set do
3     if b is in tset then
4         remove b from tset;
5     frontier(b) := {tset};
6     if b ends with a divergent branch then
7         foreach target t of this branch do
8             if priority(t) < priority(b) then
9                 add t to tset;
```

Algorithm 1: Computes the thread frontier of each BB.

In order for this to be feasible, warps [T0] and [T2, T3] need to reach BB3 at the same time. This can be accomplished by placing constraints on the scheduling order of threads - in this paper, this is achieved by assigning a scheduling priority to each basic block. Using an emulator, we i) evaluate the direct microarchitecural support for this prioritization mechanism, and ii) model and evaluate a proposed software implementation for the Intel Sandybridge GPU. This implementation is shown to produce significant gains in execution time of kernels with unstructured control flow.

Overview: In summary, re-convergence at thread frontiers is composed of the following elements.

1. A priority that is assigned to each basic block in the program (compiler).
2. A scheduling policy that ensures threads execute basic blocks in this priority order (hardware).
3. The preceding steps ensure that the thread frontier of a divergent branch can be statically computed. Insert the re-convergence checks at appropriate points in the code (compiler).
4. On encountering a compiler-placed re-convergence check during execution, check for disabled threads in the thread frontier and re-converge if possible (hardware).

The following sections describe each of these steps.

4. COMPILER SUPPORT

This section describes the compiler support for thread frontiers. While alternative HW/SW implementation techniques are feasible, all thread frontiers implementations must preserve the following two properties.

- If any thread in a warp is disabled, it is waiting in the thread frontier of the basic block being executed.
- If a partially enabled warp (at least one thread is disabled) enters a block in its thread frontier, a check must be made for re-convergence.

The thread frontier is constrained by 1) the control flow structure of a program, and 2) the scheduling order of threads. The control flow structure restricts the thread frontier to the region between the divergent branch and the immediate post-dominator. Similarly, the thread schedule restricts the thread frontier to the set of unscheduled blocks with predecessors that have already been scheduled. A proof is omitted due to space constraints. Consequently, our approach is to i) have the compiler assign a priority to basic blocks that determines the thread scheduling order, ii) determine the thread frontier of each basic block corresponding to this scheduling order, and iii) use the scheduling order to determine where to insert branch and re-converge instructions. These steps are described in the following subsections.

4.1 Thread Frontier Construction

This section describes the basic algorithm for constructing the thread frontier at compile time. The thread frontier of the current instruction can be computed using Algorithm 1. This algorithm starts with a program that has been sorted in a best effort topological order (reverse post order) and assigns blocks priorities in this order. This is the order in which blocks must be scheduled for execution. Based on this scheduling order, the algorithm traverses the control flow graph to determine the thread frontier of every basic block, i.e., the set of blocks at which disabled threads in a warp may be waiting to execute when this block is executing. The specific details of the construction are described below.

The topological order of the program shown in Figure 1 is (BB1, BB2, BB3, BB4, BB5). This is the priority order that drives the thread scheduler. Based on this scheduling order, Algorithm 1 can statically compute the thread frontier of each basic block as follows.

1. Algorithm 1 begins with an empty thread frontier, {}, which is assigned to BB1. This is because all threads in a warp will execute BB1, i.e., there can be no disabled threads in a warp when BB1 is being executed. BB1 has two successors, {BB2, BB3}. They are added to the thread frontier set (*tset* in Algorithm 1). This represents a set of basic blocks across which divergent threads in a warp may be executing after the warp has executed BB1.
2. The key insight is that if the execution of BB2 is scheduled next, then its thread frontier will comprise of {BB3} and vice versa. In our example, BB2 is prioritized higher than BB3 and thus will be scheduled next. Therefore, its thread frontier can be computed by removing it from the current set, *tset* and the remaining block, {BB3}, is set as the thread frontier of BB2. BB2 ends in a (potentially) divergent branch with successor blocks Exit and BB3. Therefore after BB2 executes, there may be divergent threads executing across Exit and BB3. I.e., they are in each others' thread frontier. These are added to the set *tset* which now grows to {BB3, Exit}.
3. The next highest priority block to be scheduled is BB3, and it is removed from *tset*. So the thread frontier of BB3 is now the contents of *tset* which is {Exit}. BB3 also ends with a (potentially) divergent branch. Therefore, the successors BB4 and BB5 are added to *tset*.
4. The next highest priority block to be scheduled is BB4 and it is removed from *tset*, and it is assigned the blocks that remain in *tset* as its frontier which is {BB5, Exit}.

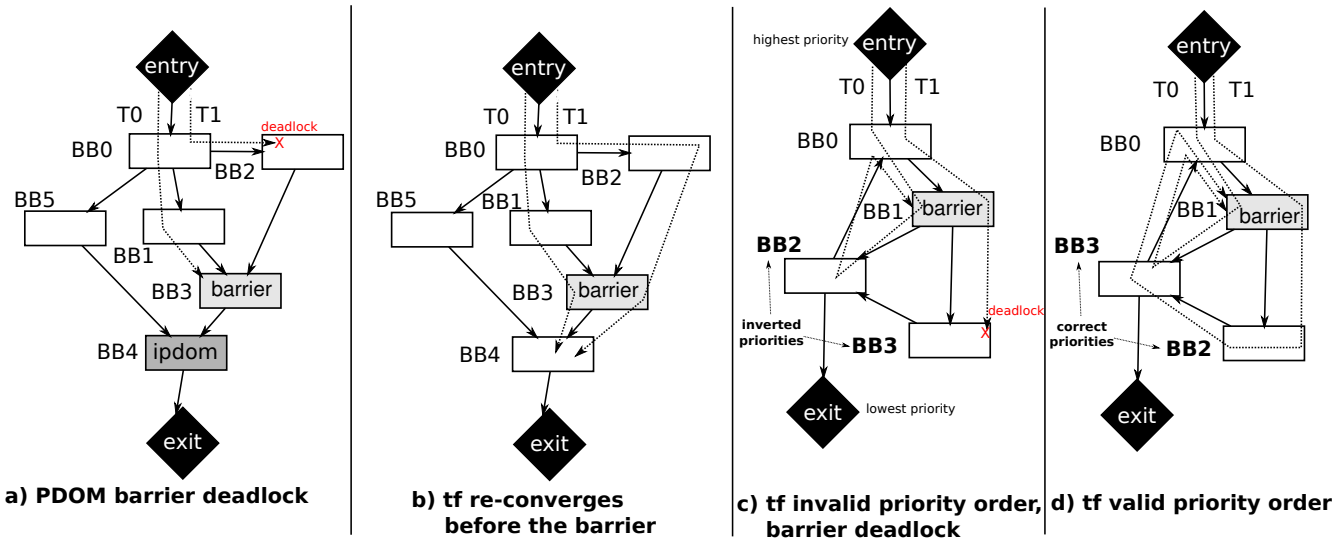

Figure 2: Interaction between re-convergence and barriers. (a) with PDOM, (b,c,d) with TF.

5. Finally, BB5 is removed from the list and assigned {Exit} as its thread frontier.

6. At the Exit block, all threads must be convergent.

In the preceding sequence of steps, at any point *tset* represents the set of blocks on which threads in a warp may be waiting to execute. Picking one basic block from *tset* places the remaining blocks in its thread frontier. This order in which blocks are picked from *tset* is the priority order in which blocks are executed and which is enforced by the thread scheduler. Consequently, at runtime when a basic block is being executed, we can ensure that threads can only be waiting on basic blocks in its thread frontier.

Finally, the placement of re-convergence checks becomes straightforward. On a branch, re-convergence checks are placed for all basic blocks that are in the thread frontier of the source block. To conclude the preceding example, checks for re-convergence are added to the branches BB2→BB3 and BB4→BB5 because the targets are contained within the thread frontier of the respective source block.

4.2 Analysis

This section examines some consequences of implementing thread frontier (TF) on commodity GPUs e.g., Intel's SandyBridge as opposed to native hardware support we advocate in Section 5. We also examine additional requirements created by the placement of barriers and potential overheads of thread frontiers

Interaction With Barriers.

Barriers are problematic because most GPUs do not support normal MIMD barrier semantics (i.e. threads execute independently until all threads reach a barrier) when a divergent warp encounters a barrier. Rather than suspending the current set of active threads and switching to another set of disabled threads, most GPUs (Sandybridge, Fermi, etc) simply suspend the entire warp [8]. This prevents these GPUs from correctly executing programs where re-convergence does not occur before the barrier is encountered. To the best of our knowledge, the CUDA programming manual indicates correct barrier semantics correspond to how the program could be realized on a MIMD processor and do not require barrier placement accommodate current hardware implementations on GPUs with SIMD data paths.

Unstructured control further complicates this problem for PDOM. It creates situations where deadlocks occur not because threads cannot re-converge before a barrier, but because the barrier is placed before the immediate post-dominator. Consider the example in Figure 2 a), where basic block labels correspond to priorities (e.g. BB0 has the highest priority). In this example, which may correspond to a situation where an exception could be thrown before a barrier, threads T0 and T1 diverge at the first block and can re-converge at BB3, which allows them to encounter the barrier in lock-step. The presence of a barrier in BB3 may be considered an assertion by the programmer that either zero or all threads within the CTA will evaluate the condition in BB0 uniformly. However, the immediate post-dominator is placed at BB4. This causes the threads to encounter the barrier one at a time, **even when the exception is not thrown**, creating a deadlock as shown in Figure 2 a). This example would not deadlock on MIMD hardware. On SIMD hardware, this problem is avoided by re-converging at thread frontiers as shown in Figure 2 b).

Figures 2 c) and 2 d) show that care must also be taken when using re-convergence at thread frontiers for programs with barriers. Figures 2 c) shows two threads, T0 and T1, that execute in a loop where priorities have been assigned incorrectly. T0 takes path (BB0, BB1, BB2, BB0, BB1) and T1 takes the path (BB0, BB1, BB3, BB2, BB0, BB1). In this case, T1 will stall on BB3 because it has lower priority than BB2, and T0 will encounter BB1 without re-converging again. Figure 2 d) shows a new assignment of priorities to blocks that avoids the problem.

In general, re-convergence at thread frontiers can ensure correct barrier semantics for all programs by giving blocks with barriers lower priority than any block along a path that can reach the barrier. This occurs because all paths that can reach the barrier will be scheduled before the barrier.

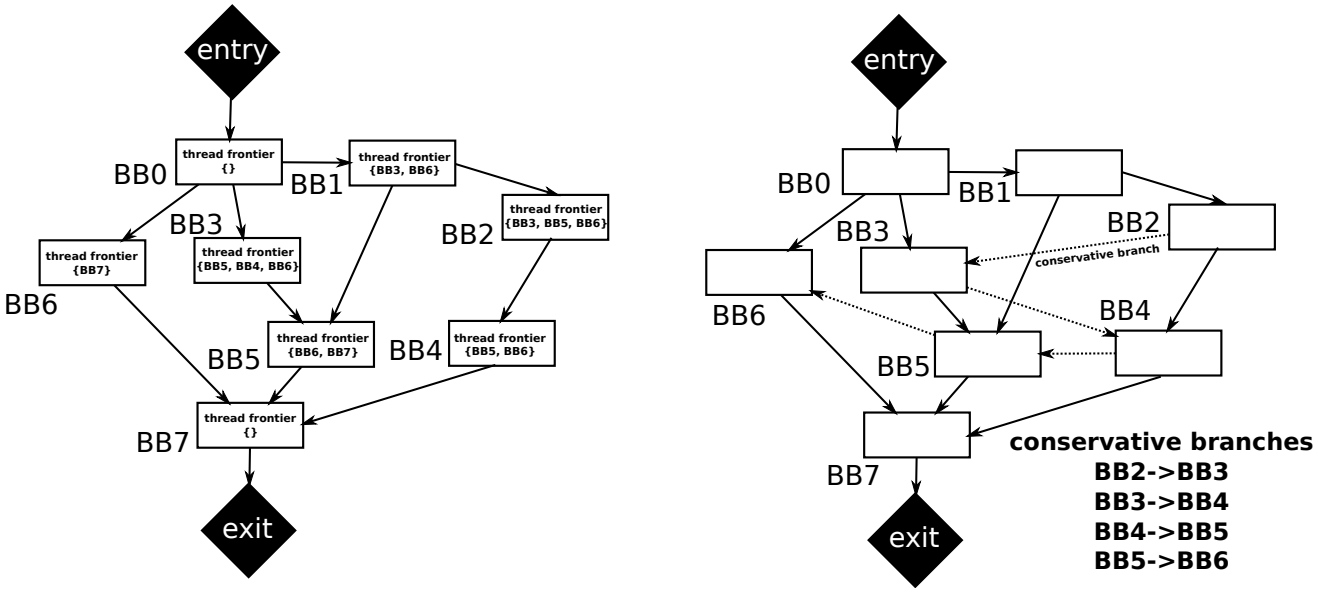

Figure 3: Thread frontiers of a control-flow graph (on the left) showing conservative branches as dotted edges (on the right).

Conservative Branches.

A core requirement of re-convergence techniques based on thread frontiers is hardware or compiler support for explicitly checking for stalled threads in the thread frontier. For example, consider the case depicted in Figure 3. In this example, basic blocks are assigned priorities according to their ID. So BB0 has the highest priority and BB7 has the lowest. Consider the case where there are two threads, T0 and T1. T0 takes the path (BB0, BB1, BB2, BB4, BB7) and T1 takes the path (BB0, BB3, BB5, BB7). According to the block priorities, T0 should execute up to BB2 and then the scheduler should immediately switch to T1 at BB3, since it is the first block with an active thread in the thread frontier. However, without support for checking the PC of all disabled threads, it is not possible to determine whether or not there are threads waiting at BB3. In this case, it may be necessary to jump to BB3 and then execute a series of instructions for which all threads are disabled until T0 is encountered again at BB4. This case is referred to as a **conservative branch**, and it can potentially degrade the performance of thread frontier schemes without hardware support for determining the PC of the next disabled thread. For GPUs without this hardware support the performance degradation of executing instructions that are effectively no-ops may counteract any potential benefits gained from re-converging before the immediate post-dominator. This case only occurs for the Intel Sandybridge GPU, and the experimental evaluation section of this paper explores it in detail.

5. HARDWARE SUPPORT

At this point, the compiler has determined the basic block priorities and the thread frontier of each basic block and accordingly has inserted instructions to check for re-convergence. Hardware support for re-convergence at thread frontiers includes the following requirements:

- **Requirement 1:** Implementation of basic block priorities
- **Requirement 2:** Using block priorities to schedule threads on a divergent branch - a thread is assigned the priority of the block it is to execute.
- **Requirement 3:** Checking for stalled threads waiting in the thread frontier (alternative software implementation can use conservative branches)

Implementing the third requirement using straightforward implementations of the first two require fully associative comparisons and reduction trees to implement in hardware. The second requirement can be achieved in part by using branch instructions that encode the preferred directions since block priorities are statically determined. However, in general correctness requires that on encountering a divergent branch the hardware scheduler uses priorities according to the following rules:

1. Branches to a block with higher priority than the current block proceed normally.
2. Branches to a block with lower priority than current block instead branch to the basic block in the branch's thread frontier with the highest priority.

The preceding scheduling rules ensure that the warp is always executing the highest priority block. Note that all disabled threads will be waiting in the thread frontier of the current warp.

While custom support would be the natural implementation vehicle, thread frontiers can be implemented on existing GPUs that support some reduced (from a thread frontiers perspective) functionality. This section begins with baseline hardware support that is already available in the Intel Sandybridge GPU [7], and an associated possible thread frontiers implementation. The next section details native

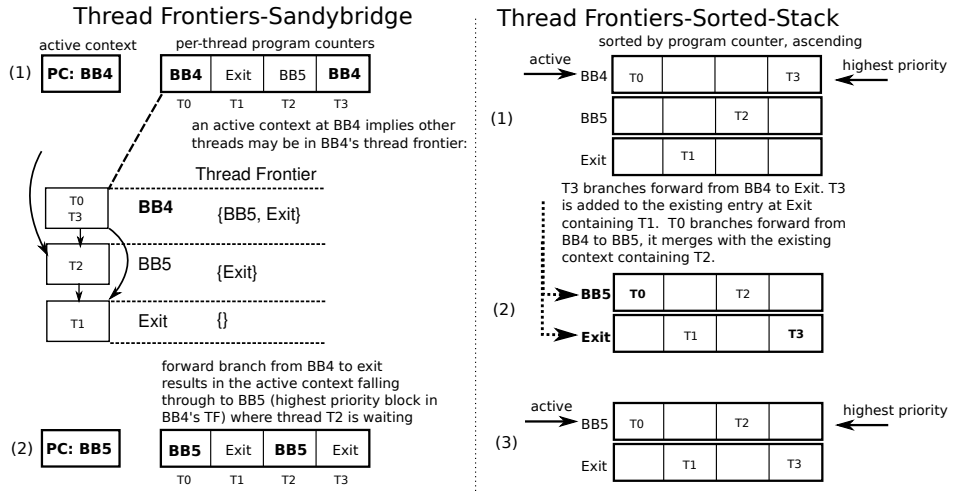

Figure 4: Example execution using thread frontiers on Sandybridge and Sorted Stack hardware.

hardware support we would envision. These hardware extensions are less complex in terms of power and area than those implemented in Sandybridge, while still including all three necessary capabilities.

5.1 Sandybridge - PTPCs

This section describes how thread frontier re-convergence can be implemented entirely as a compiler transformation on an existing commodity SIMD processor, specifically the GPU included in the Intel Sandybridge processor. The first issue concerns how block priorities can be implemented (Requirement 1). In this case, we use the block PC to represent its priority. After the priority of a block is computed as described in Section 4.1, we create a layout of the code such that the PC of the block can be used as its priority, i.e., the PCs of each block have relative values that are in the same order as block priorities. For example, if Block BB1 has a higher priority than BB2, the PC of the first instruction in BB1 is lower than the PC value of the first instruction in BB2. Note that this use of code layout is merely an optimization to use this GPU and not an inherent requirement for thread frontiers.

As indicated in the Sandybridge Programmer's Reference Manual, Sandybridge includes *per-thread program counters* (PTPCs) that are used to selectively enable or disable threads in the same warp [7]. The manual states that a write-enable predicate is computed for all instructions by comparing the PTPC of each thread against the warp's PC every cycle. If they match then the instruction executes, otherwise the thread is disabled and the instruction does not produce a result. Branch instructions (**continue**, and **if**) are available for evaluating a condition setting the PTPCs to a target, and then setting the warp PC to either another target or the fall-through path depending on type of branch. These instructions can be used to implement thread frontiers for backwards branches (using the **if** instruction) and forward branches (using the **continue** instruction) (Requirement 2). The major limitation of the hardware support in Sandybridge is that there is no support for detecting the block with the highest priority and at least one active thread. This forces the compiler to conservatively issue branches to the highest priority block in the frontier regardless of where threads may actually be waiting (Requirement 3).

The left section of Figure 4 shows an example of the application in Figure 1 executing on Sandybridge.

5.2 Native Implementation - Sorted Stack

Computing the highest priority block of any disabled thread (the one with the minimum PC) is potentially a very expensive operation to perform in hardware. Given a hypothetical GPU with 32-lane SIMD pipelines, computing the minimum PC of one thread mapped to each lane could be accomplished using a 32-input reduction tree of comparators and muxes. However, straightforward implementations would require approximately 64 32-bit comparators and 2-input 32-bit muxes.

A potentially more efficient implementation is based on the idea of maintaining information about the set of basic blocks (and their PCs) on which threads are waiting to execute. In general, the size of the required data structure would be equal to the largest thread frontier. Further, since the blocks in a thread frontier have a priority order in which they are scheduled for execution the data structure can be organized as a stack with the highest priority block (and its threads) occupying the top of the stack. Therefore we propose the use of a stack of predicate registers.

Now consider the problems of i) checking for re-convergence and ii) determining the set of disabled threads with the minimum PC. For i) re-convergence would always occur, if at all, with the first entry on the stack (the highest priority block in the thread frontier), which would require a single comparator between the current PC (when using PCs as priorities) and that of the first entry. For ii) this set of threads would always correspond to the first entry on the stack, so no additional hardware would be required. The most difficult problem with this approach is actually maintaining a sorted stack with the goals of minimizing hardware power and area, as well as supporting fast push and pop operations.

The approach that is proposed and evaluated in this paper is based on an empirical observation that the number of unique entries in such a stack is never greater than three in real workloads, even for a simulated SIMD processor with infinite lanes. This approach implements the stack as an SRAM or register file with one entry for each SIMD lane,

where each entry contains a program counter and a predicate register with one bit for each lane. A linked list containing pointers to corresponding entries in the predicate stack enables efficient sorting by block priority without exchanging activity masks. For divergent branches, a new entry is created and inserted in-order into the stack. This is accomplished by first comparing the entry's PC against the first entry in the list and either i) combining the predicate masks using a bitwise-or operation if the PCs match, ii) inserting the new entry if the current entry's PC is greater, or iii) moving on to the next entry if the current entry's PC is less. Assuming that this operation can be performed in a single cycle, this approach will require at most one cycle for each SIMD lane and at best one cycle if the first entry is selected. The right section of Figure 4 shows an example of the application in Figure 1 executing using this scheme. The performance of this scheme is further explored in Section 6.

6. EXPERIMENTAL EVALUATION

This section evaluates the performance of techniques using thread-frontiers over a set of unstructured CUDA applications.

6.1 Applications

Seven existing CUDA applications were found to experience dynamic code expansion due to unstructured control flow. They were assembled together into an unstructured benchmark suite that is examined in detail in the performance evaluation section and listed as follows here:

Mandelbrot from the CUDA SDK computes a visualization of the mandelbrot set. The kernel partitions a complex cartesian space into pixels and assigns several pixels to each thread. The unstructured control flow comes from early exit points in the inner loop, where either the next pixel is chosen or the next iteration for the current pixel is begun. **Pathfinding** [9] performs multi-agent path planning by exploring multiple paths in parallel. The code makes heavy use of conditional tests nested inside loops with early exit points, creating unstructured control flow. **GPU-Mummer** [10] performs DNA sequence alignment using suffix tree searches. Unstructured control flow arises from the traversal over the suffix tree, where the suffix links represent interacting edges. It is worth noting that this is the only application that uses gotos. **Photon-Transport** [11] models the movement of photons through materials such as tissues, blood, or water. The stochastic nature of the test creates data dependent control flow, and the use of break/continue statements inside of conditional tests creates unstructured control flow. **Background subtraction** [12] is an application that uses the extended gaussian mixture model to partition foreground objects out from background objects in a video. Compound conditions in this application create short-circuit branches and early loop exit points create interacting out-edges.

MCX [13] is another medical imaging application that focuses on the efficient implementation of a random number generator that feeds a stochastic model. Unstructured control flow is used in very long sequences of conditional expressions (9 or more terms) embedded in loops with early return points. **CUDA Renderer** [14] is a ray tracing program written in CUDA where each pixel is assigned to a thread and traverses a bounding volume hierarchy. The author used template meta-programming to inline a 32-level recursive function, each level containing short circuit branches and early return points. **Optix** [15] is an interactive real-time ray tracing engine. This application is unique because it is not written in CUDA. Rather, it uses a just-in-time PTX compiler to insert user-defined shaders into a larger ray tracing engine that performs the ray traversal over the acceleration structure, periodically calling back into the user-defined shaders. The final PTX program is loaded by a custom runtime that interacts directly with the NVIDIA driver to setup and execute the kernel. Like the CUDA Renderer, programs contain unstructured control flow in the scene graph traversal, as well as in the callbacks to the user-defined shaders, which are inlined. **Microbenchmarks** were also written to stress the performance of the re-convergence mechanism using different styles of unstructured control flow. The short-circuit benchmark simulates an object oriented program that makes a divergent virtual function call to one of several possible functions. Some of these functions make another call to a shared second function. The second benchmark makes heavy use of short circuit branches. The final three microbenchmarks contain code which may throw exceptions from within conditionals, loops, and nested function calls. Executions of these benchmarks do not result in exceptions being triggered, but their presence impacts the location of PDOM reconvergence and thus causes dynamic code expansion of GPU kernels.

Application	Forward Copies	Backward Copies	Cut Transformations	Code Expansion (%)	Avg TF Size	Max TF Size	TF Join Points	PDOM Join Points
short-circuit	12	0	0	390%	1.0	2	14	1
exception-loop	1	0	1	13.7%	.77	2	7	1
exception-call	2	0	0	81.8%	1.1	2	10	2
exception-cond	1	0	0	0.44%	.77	2	6	1
mandelbrot	1	0	0	11.7%	2.3	5	38	12
gpumummer	36	0	0	122%	3.6	8	11	7
path-finding	2	0	0	9.89%	3.1	7	22	13
photon-trans	35	0	0	13.6%	16	33	101	100
background-sub	8	0	0	7.63%	2.7	6	29	14
mcx	1433	0	18	97.4%	4.3	8	71	34
raytrace	179	0	943	58.9%	4.9	8	134	33
optix	22	0	22	29.0%	4.3	7	18	9

Figure 5: Unstructured application statistics.

6.2 Methodology

The Ocelot open source compiler infrastructure [16] was used to implement compiler support for thread frontiers. Ocelot generates code in NVIDIA's parallel thread execution (PTX) 2.3 virtual ISA. The Ocelot PTX emulator was modified to emulate the hardware support found in Intel Sandybridge and the extensions proposed in Section 5.2. A Sandybridge emulator was used for this evaluation rather than prototypes running on native hardware due to the lack of a public domain assembler and driver interface for launching compute binaries. An architecture model derived from the Intel GEN6 programmer reference guide [7] was used as a basis for our emulator support. NVCC 4.0 was used as the front-end compiler and Ocelot 2.2.2431 was used as the back-end code generator and GPU emulator. Ocelot's trace generator interface was used to attach performance models to dynamic instruction traces produced by the emulator. Since these performance models are deterministic, all results

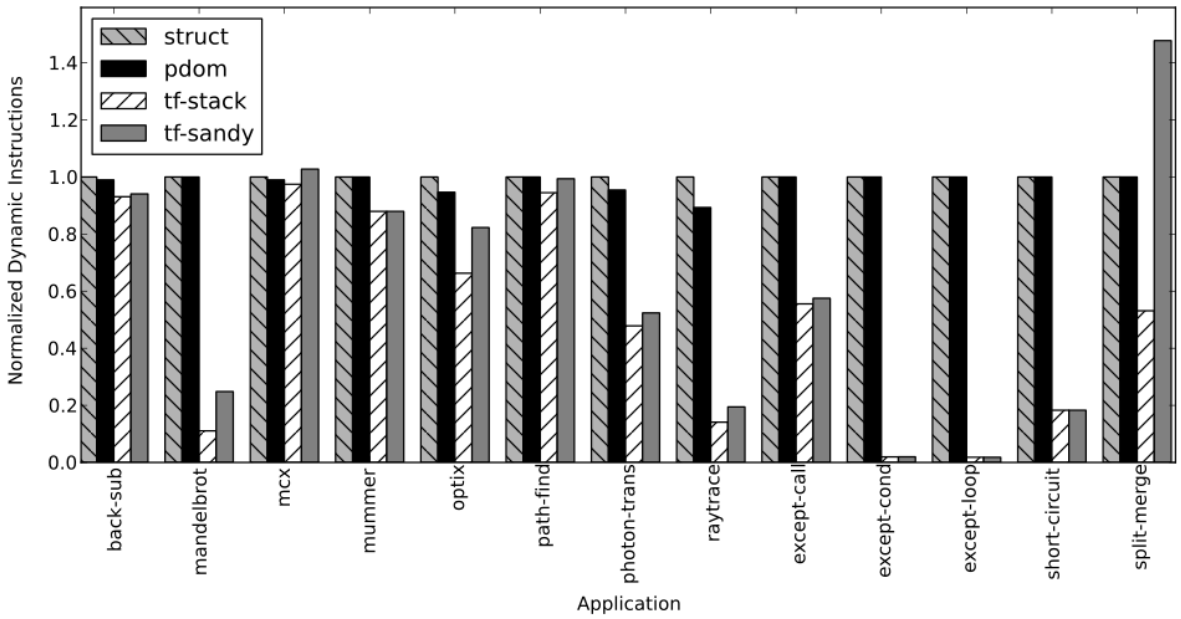

Figure 6: Normalized dynamic instruction counts.

are reported directly rather than as sampled statistics from a real system. Three sets of results are presented: application static characteristics, application dynamic characteristics, and evaluation of new language features (exceptions and virtual functions).

In the context of these experiments, **TF-STACK** refers to re-convergence at thread frontiers using the stack based hardware described in Section 5, **TF-SANDY** refers to re-convergence at thread frontiers using the modeled Sandybridge GPU, **PDOM** refers to re-convergence at the immediate post dominator, and **STRUCT** refers to applying a structural transform to remove all unstructured control flow and then execution using PDOM.

6.3 Application Static Characteristics

This section focuses on static characteristics of applications which are summarized in Table 5.

Experiment: Transformation to Structured Form.

This experiment applies the techniques described by Wu et at. [4] that extends Zhang and Hollander's three structural transforms to convert unstructured CFGs to structured CGFs. The table records the number of times each transform was applied and the dynamic code expansion that resulted. The former loosely corresponds to the number of interacting branches (causes of unstructured control flow).

Experiment: Thread Frontier Size.

The next experiment measures the minimum, maximum, and average thread frontier size for each basic block in the program. On average the thread frontier of a divergent branch is relatively small, containing only 2.55 blocks in the program where divergent threads could be executing. Photon transport is an outlier. There are 16.24 blocks in the thread frontier of the average divergent branch, up to 33 in the worst case. This implies that the structure of the CFG includes a large degree of fan out through many independent paths before they are finally merged back together.

Insight: Small Stack Size.

The fact that the average and maximum thread frontiers are relatively small indicates that the warp context stack will usually only have a few entries active. This can be leveraged in the hardware optimization process to devote more hardware resources to the management of the first few entries. For example, only the first few entries can be cached on-chip and the remaining entries can be spilled to memory. Alternatively, fully associative lookups can be avoided in the common case by only checking the first few entries before moving on to the remaining entries.

Experiment: Re-convergence Points.

The final experiment measures the number of re-converge (join) points in the program (see Table 5). These re-converge points represent opportunities for joining divergent threads back together. In all cases, there are more re-converge points for thread frontiers than for PDOM. By definition, these occur earlier than in PDOM. Except for *Photon-Transport*, most applications have 2-3x more re-converge points using thread frontiers.

6.4 Application Dynamic Characteristics

The next set of experiments examines the dynamic behavior of the four re-convergence schemes. It begins with the CUDA applications and follows with the microbenchmarks.

6.4.1 CUDA Applications

Experiment: Dynamic Instruction Counts.

Dynamic Instruction Counts are a good measurement of the benefit of a re-convergence technique because it captures the number of instructions that are redundantly ex-

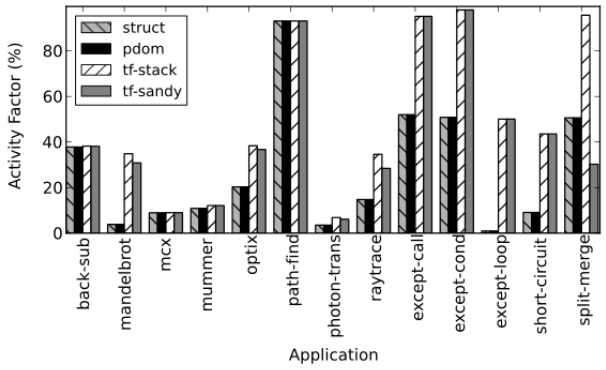

Figure 7: Activity Factor - the percentage of active threads per warp.

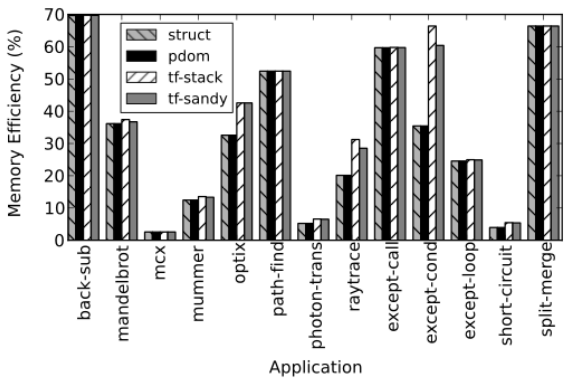

Figure 8: Memory Efficiency - the inverse of the average number of transactions required to satisfy a memory operation for a warp.

ecuted by multiple threads. Figure 6 shows the dynamic instruction counts for each of the unstructured benchmarks. All of the applications execute the fewest dynamic instructions using TF-STACK, with individual reductions ranging from 1.5% for MCX to 633% for CUDA Renderer. In general, STRUCT performs the worst, although it executes at most 5.63% more instructions than PDOM for the Optix benchmark. For TF-SANDY some of the benefits of early re-convergence are offset by the overheads of conservative branches. In the MCX application, these overheads actually cause a 3.8% slowdown compared to PDOM. Overall, the thread frontier techniques are significantly faster than the structural transform or immediate post-dominator applications when executing unstructured applications.

Experiment: Activity Factor.

Activity Factor is a measurement of SIMD efficiency defined by Kerr et al. [17]. It is defined to be the ratio of active threads in a warp to total threads in a warp, assuming an infinitely wide SIMD machine. It is a measure of the impact of divergent control flow on hardware utilization. Figure 7 shows the activity factor of each of the benchmarks. Several of the applications have activity factors of less than 20%, indicating a high degree of control flow divergence. In general, applications with a very low activity factor experience the greatest improvements with TF-STACK, whereas applications like path-finding that already have a high (80%) activity factor have little room for improvement. This suggests that thread frontier re-convergence will be the most beneficial to applications that experience significant branch divergence when executing on SIMD GPUs.

Experiment: Memory Efficiency.

Memory Efficiency is a measurement of memory access coalescing. It is defined as the average number of transactions required to satisfy a memory operation executed by all threads in a warp. Ideally, only one transaction is required if all threads in the warp access uniform or contiguous addresses. However, as an access becomes less coalesced, with different threads accessing non-contiguous addresses, more transactions are required to satisfy the request, placing greater demands on the memory subsystem. Figure 8 shows the memory efficiency for each of the unstructured applications.

Insight: Memory and SIMD Efficiency.

These results suggest that the improvements in SIMD efficiency gained from early re-convergence at thread frontiers also improve memory efficiency. Threads that diverge and then make memory accesses will always issue multiple memory transactions. Allowing threads to execute in lock-step more frequently provides additional opportunities for coalescing accesses from multiple threads into a single transaction.

6.4.2 New Application Features

The purpose of the experiments in this section is to explore desirable new language semantics that are not widely used in existing GPU workloads with the expectation that significantly increasing their efficiency may make their use in performance sensitive applications acceptable.

Experiment: Divergent Function Calls.

This experiment examines the case where each thread in warp executes a different function (via a function pointer), resulting in full divergence. Then, in the body of each function, some threads call the same shared function. The immediate post-dominator of this code will be at the return site of the first function call, serializing execution through the shared function. The split-merge application in Figure 6 shows that TF-Stack is able to re-converge earlier and execute the shared function cooperatively across several threads.

Insight: Unstructured Call Graphs.

The modular decomposition of an application into libraries with commonly used functions introduces unstructured control flow in the program call graph. As GPU applications evolve to include more function calls, the problem of branch divergence and re-convergence will expand beyond the program control flow graph to the program call graph. In this new regime, the prevalence of library routines such as operations on STL containers will lead to increased opportunities for re-convergence that can be leveraged by tracking thread frontiers.

Experiment: Exceptions.

The next example explores the performance of an application that makes heavy use of try-catch style exceptions. As CUDA does not currently support C++ try/catch style exceptions, they are implemented in this example directly using goto statements. Three microbenchmarks are considered. *Exception-call* throws an exception from within a divergent function call, *exception-cond* throws an exception from within a divergent conditional statement, and *exception-loop* throws an exception from within a divergent loop. Exceptions prevent re-convergence before the catch block using PDOM because control can transfer to it immediately, skipping the function exit block, loop exit block, or end of the conditional statement. Figure 6 shows that TF-Stack suffers no performance degradation in these cases. It is also interesting to note that merely including throw statements degrades the performance of PDOM, even if they are never encountered at runtime. The reason is that the unstructured control flow forces part of the code in the try block to run multiple times if PDOM is used. Thus, investing a technique such as thread frontier to support exceptions is worthwhile.

7. RELATED WORK

Techniques for mapping control flow onto SIMD processors date back to the first SIMD machines. ILLIAC V [18] included a single predicate register that supported unnested if-then-else conditionals and do-while loops. The CHAP [5] GPU first introduced the concept of a stack of predicate registers. The ISA is augmented with explicit control flow instructions for handling if-else-endif and do-while statements. These statements could be nested by performing push-pop operations on the stack, and many modern GPUs such as Intel Ironlake and AMD Evergreen [19] use this technique. However, this technique requires a structural transform [20] to execute programs with unstructured control flow.

Immediate post dominator re-convergence (PDOM) described by Fung et al. [6] expands on the concept of a predicate stack to support arbitrary branch instructions by placing push operations on divergent branches and pop operations on re-converge instructions. This technique simplifies the compilation process, but still leads to excessive thread serialization when executing unstructured control flow.

Recent work has focused on improving SIMD utilization and hiding latency by changing the mapping from threads to warps using dynamic warp formation [6] and allowing divergence on other high latency events such as cache misses using dynamic warp subdivision [21].

Thread block compaction [22] identifies that many of these techniques end up trading better SIMD utilization for worse memory access regularity, which actually degrades performance. The authors propose the use of a CTA-wide predicate stack to periodically synchronize threads at immediate post-dominators, and encourage lock-step execution among multiple warps. These techniques are orthogonal and complementary to thread frontiers because they all rely on PDOM for identifying re-convergence points.

Very recent work has begun to address the interaction of branch divergence and unstructured control flow. The authors of [22] augment TBC with *likely convergence points* (LPCs). These are locations with interacting control-flow edges in which re-convergence is probable, but PDOM results in conservative placement of re-converge instructions. This work requires additional stack entries to accommodate likely convergence points and similar hardware comparators to merge entries. LCPs are identified through either static analysis or through profiling; they have some similarities with the re-convergence points inserted at thread frontiers. However, a generic method for inserting them that handles all unstructured control flow is not presented.

A recent patent by Coon and Lindholm [23] introduces explicit break and continue instructions to handle certain classes of unstructured control flow in NVIDIA GPUs. These instructions work by distinguishing between loop exit points and other branches. When a loop exit point is encountered, it re-converges with the post dominator of the loop entry point rather than the post dominator of the last divergent branch. This allows normal branches to ignore loop exit branches when finding the PDOM. This approach is effective at certain classes of unstructured branches, but it is not generic: a jump into a loop body, a short-circuited conditional inside a loop, or another unstructured pattern would defeat it. Furthermore, this approach still requires a structural transform [20] to distinguish between loop exits and other branches, which is a high overhead algorithm.

8. CONCLUSIONS

Existing SIMD re-convergence mechanisms (PDOM) are efficient when executing structured control flow. However, they force unnecessary serialization when threads execute unstructured control flow. Re-convergence at thread frontiers allow a compiler to identify the earliest possible re-convergence point of any divergent branch even if it occurs in unstructured control flow. Once identified, one of several possible hardware schemes can be used to re-converge threads as they enter the frontier. This paper demonstrates that the existing hardware support used in Intel Sandybridge can be re-purposed to re-converge at thread frontiers. It also proposes new lightweight hardware support that can perform this re-convergence more efficiently. Eight existing CUDA applications are found to execute $1.5 - 633.2\%$ fewer dynamic instructions using thread-frontiers, showing that re-convergence at thread frontiers is highly effective at mapping SIMT programs onto SIMD processors.

Additionally, several control flow patterns that were once very inefficient on SIMD processors are shown to become significantly faster using thread frontiers. Exceptions used in conditional statements, loops, or nested function calls no longer degrade performance. Warps that diverge before calling the same function can re-converge and execute that shared function in lock-step. It is our hope that this technique will make GPUs more amenable to highly unstructured applications such as scene graph traversal used in ray tracing, state machine transitions common to non-deterministic finite automata, or traversals of highly unstructured data structures such as grids or graphs with data-dependent split and join points.

9. ACKNOWLEDGEMENTS

The authors are grateful to Tor Aamodt, Nathan Clark, Wilson Fung, and Norman Rubin for discussions leading to the refinement of Thread Frontiers, and to the anonymous reviewers for their particularly insightful suggestions. This research was supported in part by NSF under grants IIP-1032032, CCF-0905459, by LogicBlox Corporation, and equipment grants from NVIDIA Corporation.

10. REFERENCES

[1] T. Hatazaki, "Tsubame-2 - a 2.4 pflops peak performance system," in *Optical Fiber Communication Conference*. Optical Society of America, 2011.

[2] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying numa and contention effects in multi-gpu systems," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 11:1–11:7.

[3] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, 1990.

[4] H. Wu, G. Diamos, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in gpu applications," in *The First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*. ACM, June 2011.

[5] A. Levinthal and T. Porter, "Chap - a simd graphics processor," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 77–82, 1984.

[6] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[7] Intel, *Intel HD Graphics OpenSource Programmer Reference Manual*, June 2010.

[8] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, 2010, pp. 235 –246.

[9] Wen-mei W. Hwu, "Real-time adaptive gpu multi-agent path planning," in *GPU Computing Gems*, Wen-mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2.

[10] C. Trapnell and M. C. Schatz, "Optimizing data intensive gpgpu computations for dna sequence alignment," *Parallel Computing*, vol. 35, pp. 429–440, August 2009.

[11] A. Badal and A. Badano, "Accelerating monte carlo simulations of photon transport in a voxelized geometry using a massively parallel gpu," *Medical Physics 36*, 2009.

[12] V. Pham, P. Vo, H. V. Thanh, and B. L. Hoai, "Gpu implementation of extended gaussian mixture model for background subtraction," *International Conference on Computing and Telecommunication Technologies*, 2010.

[13] Q. Fang and D. A. Boas, "Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units," *Optical Express 17*, vol. 17, pp. 20 178–20 190.

[14] T. Tsiodras, "A real-time raytracer of triangle meshes in cuda," Tech. Rep., February 2011. [Online]. Available: http://users.softlab.ntua.gr/~ttsiod/cudarenderer-BVH.html

[15] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics*, vol. 29, pp. 66:1–66:13, July 2010.

[16] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of PACT '10*, 2010.

[17] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *IISWC09: IEEE International Symposium on Workload Characterization*, Austin, TX, USA, October 2009.

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

[19] AMD, *Evergreen Family Instruction Set Architecture Instructions and Microcode*, 2010.

[20] F. Zhang and E. H. D'Hollander, "Using hammock graphs to structure programs," *IEEE Trans. Softw. Eng.*, pp. 231–245, 2004.

[21] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 235–246.

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

[23] B. W. Coon and E. J. Lindholm, "System and method for managing divergent threads in a simd architecture," Patent US 7 353 369, April, 2008.