Thread Block Compaction for Efficient SIMT Control Flow

Wilson W. L. Fung and Tor M. Aamodt
University of British Columbia
wwlfung@ece.ubc.ca, aamodt@ece.ubc.ca

Abstract

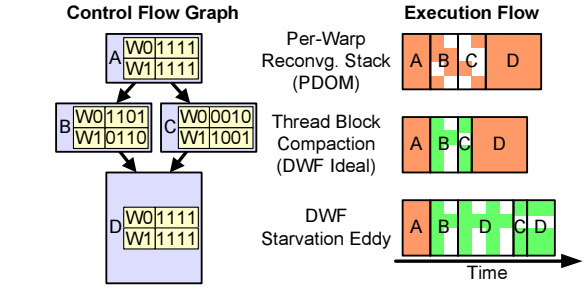
Manycore accelerators such as graphics processor units (GPUs) organize processing units into single-instruction, multiple data "cores" to improve throughput per unit hardware cost. Programming models for these accelerators encourage applications to run kernels with large groups of parallel scalar threads. The hardware groups these threads into warps/wavefronts and executes them in lockstep—dubbed single-instruction, multiple-thread (SIMT) by NVIDIA. While current GPUs employ a per-warp (or per-wavefront) stack to manage divergent control flow, it incurs decreased efficiency for applications with nested, data-dependent control flow. In this paper, we propose and evaluate the benefits of extending the sharing of resources in a block of warps, already used for scratchpad memory, to exploit control flow locality among threads (where such sharing may at first seem detrimental). In our proposal, warps within a thread block share a common block-wide stack for divergence handling. At a divergent branch, threads are compacted into new warps in hardware. Our simulation results show that this compaction mechanism provides an average speedup of 22% over a baseline per-warp, stack-based reconvergence mechanism, and 17% versus dynamic warp formation on a set of CUDA applications that suffer significantly from control flow divergence.

1. Introduction

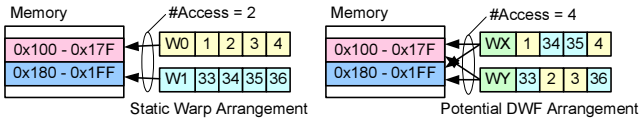
Many throughput computing workloads exhibit data level parallelism [6] yet achieve poor performance on current single-instruction, multiple data (SIMD) based accelerator architectures due to divergent control flow, or branch divergence [12, 9, 19, 13]. While conventional SIMD architectures can support correct execution of divergent control flow through vector lane masking (predication) [4] and/or stack-based reconvergence mechanisms [15, 7, 9], such mechanisms reduce throughput. Given the significant *potential* area and energy consumption advantages of SIMD hardware versus general purpose parallel hardware, improved control-flow mechanisms are desirable.

Proposals for attacking this challenge include novel stream programming language extensions [12], allowing vector lanes to run in scalar mode for short code sequences [14], dynamically re-grouping scalar threads into new "warps" [9], and simply abandoning vector hardware altogether [19, 13]. Each exhibits varying tradeoffs.

Dynamic warp formation (DWF) [9] captures a significant fraction of the benefits of multiple-instruction, multiple data (MIMD) hardware on multi-threaded SIMD processing units employing large multi-banked caches. However, the benefits of DWF can be affected by the scheduling policy used to issue ready warps [9, 10] and memory systems that limit bandwidth to first level memory structures. Figure 1(a) shows an example that helps to illustrate a form of pathological scheduling behaviour that can occur in DWF. This figure compares various single-instruction,

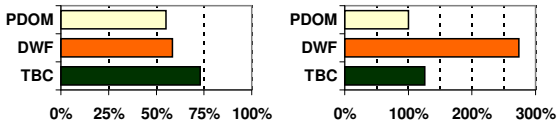


(a) Starvation eddy scheduling problem. While not-reconverging as soon as possible may benefit latency tolerance [20], it is a major cause of reduced SIMD efficiency in dynamic warp formation [10]. In the execution flow, shaded = utilized SIMD processing element, white = idle SIMD processing element.



(b) Extra memory accesses introduced by random thread grouping.

Figure 1. Dynamic warp formation pathologies.



(a) SIMD Efficiency (b) Normalized Memory Stalls

Figure 2. Overall performance (details in Sec. 5).

multiple-thread (SIMT) control flow handling mechanisms for a branch hammock[1] diverging at block A. Each basic block contains the activemask for two warps where a "0" means the corresponding lane is masked off. The stack based reconvergence mechanism (PDOM) executes block B and C with decreased SIMD efficiency, but reconverges at block D. The bottom right of Figure 1(a) illustrates a case where the threads at block C fall behind those at B. While this scheduling can increase latency tolerance [20], it can also lead to a reduction in performance [10] since ideally the warps at block B and C could form fewer warps at block D. We call this fall-behind behaviour a *starvation eddy*, and find it affects the SIMD efficiency of many CUDA applications run with DWF as shown in Figure 2(a) (applications and configuration in Section 5). Here SIMD efficiency is the average fraction of SIMD processing elements that perform useful work on cycles where the SIMD processing unit has a ready instruction to execute. Furthermore, CUDA applications tend to be written assuming threads in a warp will execute together and should therefore access nearby memory locations. DWF tries to optimize control flow behaviour at the potential expense of increasing memory accesses (demonstrated in Figure 1(b)). Across the workloads we study, this leads to 2.7× extra stalls at the memory pipeline (Figure 2(b)).

In this paper, we explore an alternative hardware mechanism for improving the performance of applications that suffer from control flow divergence on GPU-like manycore accelerators. This mechanism maintains the key benefits of DWF of creating new warps to improve SIMD efficiency, while largely eliminating the DWF pathologies described above. The key insight is that typical control flow intensive CUDA applications exhibit sufficient "control flow locality" within the group of scalar threads used for bulk-synchronous parallel execution that full DWF and/or MIMD flexibility is not necessary to regain most of the performance loss due to branch divergence. Extending the existing per-warp reconvergence stack mechanisms to encompass warps executing within a thread block and dynamically compacting threads after divergent branches using simplified DWF hardware achieves more robust performance. The contributions of this paper are:

- It evaluates challenges faced by dynamic warp formation running CUDA applications, and devises an improved scheduling policy to address these challenges.
- It proposes a novel "thread block compaction" (TBC) mechanism that exploits control flow locality among threads within a thread block to robustly provide the benefits of dynamic warp formation.
- It extends immediate post-dominator based reconvergence with *likely-convergence points*.

[1]Note that the mechanisms studied in this paper support CUDA and OpenCL programs with arbitrary control flow within a kernel.

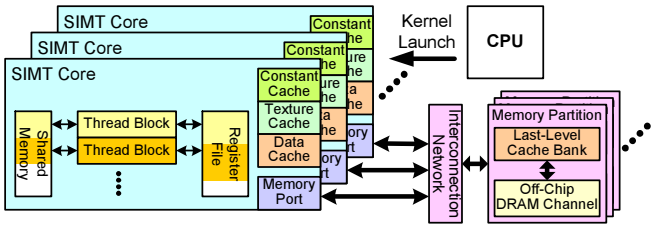


Figure 3. Abstract GPU architecture exposed by the CUDA programming model.

Figure 1(a) shows that TBC eliminates the starvation eddy with the block-wide stack enforcing reconvergence at block D. This leads to a significantly higher SIMD efficiency with TBC versus PDOM or DWF (Figure 2(a)) and fewer memory pipeline stalls compared to DWF (Figure 2(b)).

2. Baseline SIMT Accelerator

This section describes our baseline GPU-like manycore accelerator and dynamic warp formation.

2.1. Architecture

We study GPU-like manycore accelerator architectures running CUDA applications [21, 24]. A CUDA program starts on a CPU then launches parallel *compute kernels* onto the accelerator. Each kernel launch in CUDA dispatches a hierarchy of threads (a *grid* of *blocks* of *warps* of scalar *threads* running the same compute kernel) onto the accelerator. Blocks are allocated as a single unit of work to a single-instruction, multiple thread (SIMT) [16] core which is heavily multi-threaded. Threads within a block can communicate via an on-chip "shared memory". The SIMT cores access banks of a globally coherent last-level (L2) cache and DRAM memory channels via an on-chip network.

With the SIMT execution model, scalar threads are managed as a SIMD execution group called a warp (or wavefront in AMD terminology). In the CUDA programming model, each warp contains a fixed group of scalar threads throughout a kernel launch. This arrangement of scalar threads into warps is exposed to the CUDA programmer/compiler for various control flow and memory access optimizations [24]. In this paper, we refer to warps with this arrangement as *static warps* to distinguish them from the *dynamic warps* that are dynamically created via dynamic warp formation or thread block compaction.

2.2. Microarchitecture

Figure 4 illustrates our understanding of the multi-threaded microarchitecture within a SIMT core of a contemporary GPU that we assumed while developing thread block compaction. We defined this microarchitecture, described below, by considering details found in recent patents [8, 17, 7]. In our evaluation, we approximate some details to simplify our simulation model: We model the fetch unit as always hitting in the instruction cache, and allow an instruction to be fetched, decoded and issued in the same cycle. We

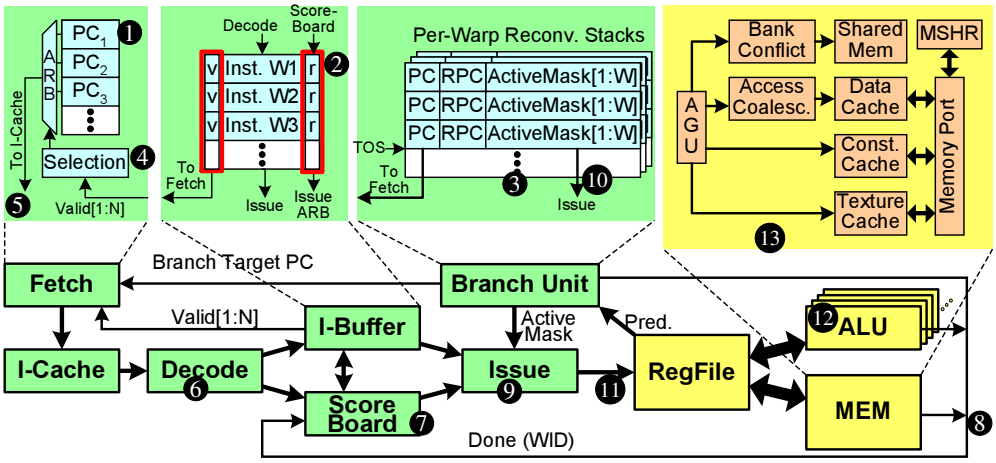


Figure 4. SIMT core microarchitecture of a contemporary GPU. N = #warps/core, W = #threads in a warp.

also employ a simplified scoreboard that forbids concurrent execution of subsequent instructions from the same warp and a unified pipeline that services both ALU and MEM instructions. These restrictions have little performance impact, because the applications we study usually have more warps interleaved on the hardware than pipeline stages.

2.2.1 Fine Grained Multi-Threading

Each SIMT core interleaves up to N warps on a cycle-by-cycle basis (we used N=32). Each warp has a program counter (PC) in the fetch unit (see ❶ in Figure 4), a dedicated slot (❷) in the instruction buffer, and its own stack (❸) to manages branch divergence *within* that warp.

Each slot (❷) in the instruction buffer contains a v-bit indicating when an instruction is present, and an r-bit indicating that (❷) it is ready for execution. Every cycle the fetch unit selects the PC for a warp with an empty instruction slot (❷), and fetches the corresponding instruction from the instruction cache (❺). The instruction is decoded (❻) and placed in an empty slot in the instruction buffer (❷). This instruction waits in the instruction buffer until its ready bit (❷) is set by the scoreboard (❼), indicating prior instructions from this warp have completed. Instructions within a warp execute in-order. Our evaluation employs a simple scoreboard that only tracks prior instruction completion for each warp. This scoreboard can be enhanced to track per-warp register dependencies [8], potentially allowing multiple instructions per warp in the pipeline.

The issue logic (❾) selects a warp with a ready instruction in the instruction buffer to issue for execution. As an instruction issues it acquires the activemask (❿) from the top entry on the corresponding warp's reconvergence stack in the branch unit. The activemask disables threads in the warp that should not execute due to branch divergence. After a branch from a warp is decoded, none of its instructions can be fetched until the branch outcome is known. Once issued (⓫), the slot in the instruction buffer (❷) that contained the instruction is marked *invalid*, signaling the fetch unit that it may fetch the next instruction for this warp.

The issued instruction fetches its operands from the register file. It is then executed in the corresponding pipeline (ALU ⓬ or MEM ⓭). Instructions write their results to the register file and notify the scoreboard (❽). The scoreboard updates the r-bit (❷) of the next fetched instruction of the corresponding warp.

When a memory instruction issues (⓭), the address generation unit (AGU) generates addresses for each thread in the warp. For shared memory accesses[2], the bank conflict handling unit ensures that the shared memory banks process conflicting accesses in successive cycles. For "global" and "local" memory accesses [24], the access coalescing unit merges accesses from different lanes to the same cache line into a single wide access and the data cache processes these wide accesses one per cycle. The constant cache operates similarly, except accesses to different addresses are serialized without coalescing. Texture accesses are serviced by the texture unit, which accesses a texture cache.

2.2.2 Stack-Based Branch Divergence Hardware

Below, we summarize the reconvergence stack mechanism we model in our baseline, which is similar to that described by Fung et al. [9, 10]. If different threads in a warp have different outcomes when a branch executes, i.e., the branch diverges, new entries will be pushed onto the warp's reconvergence stack (❸). Each entry's reconvergence PC (RPC) is set to the immediate post-dominator[3] of the branch. Each bit in the activemask indicates whether the corresponding thread follows the control flow path corresponding to the stack entry. The PC of the top-of-stack (TOS) entry is sent to the fetch unit to initiate execution of the instruction at the target path of the branch. Reaching the reconvergence point is detected when the next PC equals the RPC at the TOS entry. When this occurs, the top of the stack is popped (current GPUs use special instructions to manage the stack [15, 7]).

[2] Shared memory is an on-chip scratchpad memory in NVIDIA GPUs.

[3] Closest point in the program that all paths leaving the branch must go through before exiting the function.

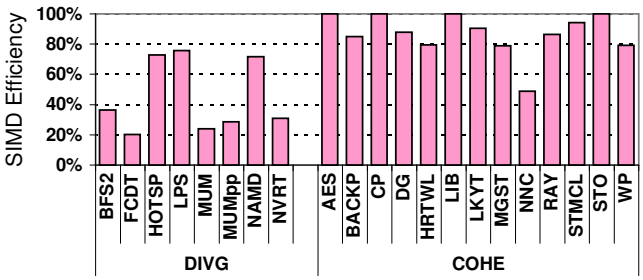


Figure 5. SIMD efficiency of CUDA applications. See Section 5 for methodology.

While this mechanism allows divergent threads to reconverge at the earliest *statically* known convergence point, it *may* result in low SIMD efficiency in applications with deeply nested data dependent control flow or loop bounds that vary across threads in a warp. In these situations different threads within a warp may follow different execution paths. We use SIMD efficiency to identify CUDA applications that diverge heavily (DIVG) but also study a representative set of *coherent* (COHE) applications in which threads in a warp follow the same execution paths (Figure 5). We classify a benchmark as DIVG if it has SIMD efficiency below 76% and COHE otherwise (NNC contains 16 threads per block and no branch divergence).

2.3. Dynamic Warp Formation

Dynamic warp formation (DWF) [9] regroups threads executing the same instruction into new warps to improve SIMD efficiency. Warp scheduling policies significantly impact DWF [9], since a large group of threads needs to progress through the kernel at roughly the same pace. The previous best (on average) policy known, majority, incurs poor performance when a small number of threads "falls behind" the "majority" of threads [10]. Such *starvation eddies* reduce opportunities for such threads to regroup with the "majority" leading over time to lower SIMD efficiency.

Figure 6 compares DWF with majority scheduling against the baseline reconvergence stack mechanism (PDOM) on the DIVG applications. While DWF improves performance significantly on BFS2, FCDT, and MUMpp, other applications suffer a slowdown[4]. One application, NVRT executes incorrectly with DWF, because it uses a single manager thread in each static warp to continually acquire tasks from a global queue (atomically acquire a range of task IDs) for other worker threads in the warp. The per-warp reconvergence stack enforces an implicit synchronization as it forces the worker threads in the warp to wait

[4]In our previous evaluation [9, 10], each SIMT core had large multi-bank L1 caches to buffer the memory system impact of DWF, whereas each SIMT core in this paper only has a much smaller, single banked L1 cache, which may be desirable in practice to reduce the complexity and area of the memory system. The applications evaluated in our prior study [9, 10] also lacked the memory coalescing optimizations found in most CUDA applications (including those evaluated here) masking the impact of thread regrouping on the memory system.

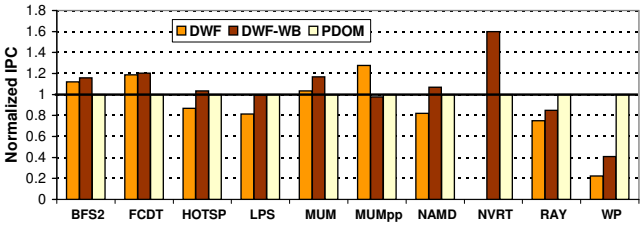


Figure 6. DWF with and without warp barrier compared against baseline per-warp PDOM stack reconvergence.

while this manager thread is acquiring tasks. DWF executes NVRT incorrectly because it does not enforce this behaviour. With DWF the worker threads incorrectly execute ahead with obsolete task IDs when the manager thread is acquiring new tasks. In general, we observed three problems when running CUDA applications on DWF enabled execution model: (1) Applications relying on implicit synchronization in a static warp (e.g. NVRT) execute incorrectly; (2) Starvation eddies may reduce SIMD efficiency; (3) Thread regrouping in DWF increases non-coalesced memory accesses and shared memory bank conflicts.

2.4. Warp Barrier

In this section we propose an extension to DWF, called a *warp barrier*. This mechanism keeps threads in their original static warps until they encounter a divergent branch. After a top-level divergence, threads can freely regroup between diverged warps but a "warp barrier" is created per static warp at the immediate post-dominator of the top-level divergent branch. A *top-level divergence* is a divergent branch that is not control dependent upon an earlier divergent branch. A dynamic warp may contain threads with different warp barriers. When a dynamic warp reaches a warp barrier those threads associated with the barrier are restored to their original static warp and wait until all threads from this static warp have arrived at the barrier. The remaining threads in the dynamic warp continue execution using the original DWF mechanisms [9]. The warp barrier mechanism confines starvation eddies between top-level divergence and reconvergence points while preserving the static warp arrangements reduces memory divergence [20] and shared memory bank conflicts. Warp barriers are distinct from `__syncthreads()` in CUDA – they are created dynamically only at divergent branches and there is one per static warp rather than one per thread block.

Figure 6 compares the performance of DWF with this warp barrier mechanism (DWF-WB) against the original DWF and the baseline per-warp reconvergence stack (PDOM). With the warp barrier, NVRT executes properly and achieves a 60% speedup over PDOM. Three other applications that suffer slowdowns with the original DWF (HOTSP, LPS, NAMD) now achieve speedup. However, MUMpp loses performance with the warp barrier and shows a slight slowdown versus PDOM. In addition, RAY and WP

continue to suffer from starvation eddies while using warp barriers. A deeper investigation suggests these applications require additional barriers between the top-level divergence and reconvergence of a static warp. Such barriers are a a natural property of reconvergence stacks and this led us to propose *thread block compaction*.

3. Thread Block Compaction

In CUDA (OpenCL) threads (work items) are issued to the SIMT cores in a unit of work called a "thread block" (work group). Warps within a thread block can communicate through shared memory and quickly synchronize via barriers. Thread block compaction extends this sharing to exploit control flow locality among threads within a thread block. Warps *within a thread block* share a block-wide reconvergence stack for divergence handling instead of having separate *per-warp* stacks. At a divergent branch, the warps synchronize and their threads are "compacted" into new warps according to the branch outcome of each thread. The compacted warps then execute until the next branch or reconvergence point, where they synchronize again for further compaction. Compaction of all the divergent threads after they have reached the reconvergence point will restored their *original* warp grouping before the divergent branch was encountered.

As threads with a different program counter (PC) value cannot be merged in the same warp, DWF is sensitive to scheduling. When branch divergence occurs, a sufficient number of threads needs to be present at the divergent branch to be merged into full warps. Ideally, threads should be encouraged to be clustered at a local region in the kernel program, but variable memory access latency and complex control flow make this hard to achieve. Moreover, even if this scheduling could be achieved, it will cluster memory accesses, discouraging overlap between memory access and computation and may increase memory latency via increased contention in the memory system.

Thread block compaction simplifies this scheduling problem with block-wide synchronization at divergent branches. This ensures that the maximum number of threads wait at a branch or reconvergence point, while other threads can be scheduled focusing on improving pipeline resource utilization. With the use of a reconvergence stack, we can keep track of the warps that will eventually arrive at the reconvergence point and eliminate the starvation eddy problem described in Section 2.3 for DWF. The synchronization overhead at branches can be covered by switching the execution to a different thread block running on the same SIMT core. In Section 6.2, we explore the performance impact of several thread block prioritization policies.

While sharing a single control flow stack among all warps in a block can in theory reduce performance when threads in *different* warps follow diverging control flow paths, we find this type of code to be rare. It tends to occur where CUDA programmers work around the limitation of one concurrent kernel launch at a time on pre-Fermi NVIDIA GPUs by having different warps in a block execute different code following a top-level "`if`" or "`switch`" statement. If better support were desired for such code, we could employ prior proposals for enabling stack-based reconvergence mechanisms to overlap execution of portions of the same warp that follow different control flow paths [20]. We leave evaluation of such extensions to future work and instead focus on what we observe to be the common case for existing applications.

Example 1 Code that exhibits branch divergence.

```
t = threadIdx.x; // block A
flag = (t==1)||(t==6)||(t==7);
if( flag )
    result = Y;  // block B
else
    result = Z;  // block C
return result;   // block D
```

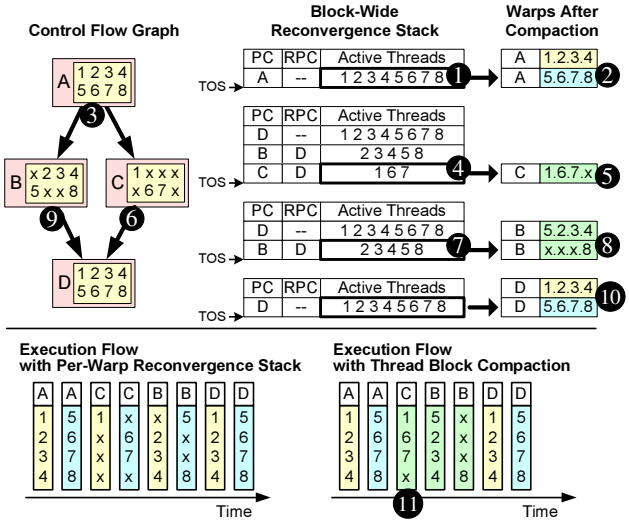


Figure 7. High-level operation of thread block compaction.

3.1. High-Level Operation

Figure 7 illustrates the high-level operation of thread block compaction. The code in Example 1 translates into the control flow graph in Figure 7. In this example, each warp contains four threads and each thread block contains eight threads. The numbers in each basic block of the control-flow graph (left portion of figure) denote the threads that execute that block. All threads execute block A and D, while only threads (1,6,7) execute block C and only threads (2,3,4,5,8) execute block B.

Two warps composed of threads 1-4, and 5-8 begin executing at the start of block A. Since there is no divergence, there is only a single entry in the block-wide reconvergence stack (❶ in Figure 7). Two warps (❷) are created from the active threads (❶). The two warps are scheduled on the pipeline independently until they reach the end of block A

(❸), where they "synchronize" at the potentially divergent branch. This synchronization allows the hardware to determine if any of the warps has diverged at the branch. As an optimization, the programmer/compiler can statically annotate non-divergent branches (such as `bra.uni` in the PTX-ISA [23]) in the kernel, allowing the warps to skip synchronization at these branches. After both warps have executed the branch, two new entries (❹) will have been pushed onto the stack, each containing the active threads that will execute the "taken" or "not taken" side of the branch (block C or B, respectively). The active threads on the top entry are compacted into a single warp that executes basic block C (❺). As this warp reaches the reconvergence point D (❻), its entry is popped from the reconvergence stack (❼), and the active threads that execute basic block B are compacted into two warps (❽). After these two warps have reached the reconvergence point D (❾), their corresponding entry is popped from the stack, and threads resume execution in two full warps with their original arrangements before the divergent branch (❿).

The lower part of Figure 7 compares the execution flow of thread block compaction with the baseline per-warp reconvergence mechanism. In this example, thread block compaction compacts threads (1,6,7) into a single warp at basic block C (⓫). This reduces the overall execution time by 12.5% over the baseline in this example.

The pushing and popping of the entries on and off the block-wide reconvergence stack, as branches and reconvergence points are encountered, uses the same reconvergence points as the per-warp reconvergence stack in the baseline SIMT core.

3.2. Implementation

Figure 8 illustrates the modifications to the SIMT core microarchitecture to implement thread block compaction. The modifications consist of three major parts: a modified branch unit (❶), a new hardware unit called the thread compactor (❷), and a modified instruction buffer called the warp buffer (❸). The branch unit (❶) has a block-wide reconvergence stack for each block. Each entry in the stack consists of the starting PC (PC) of the basic block that corresponds to the entry, the reconvergence PC (RPC) that indicates when this entry will be popped from the stack, a warp counter (WCnt) that stores the number of compacted warps this entry contains, and a block-wide activemask that records which thread is executing the current basic block. The thread compactor (❷) consists of a set of priority encoders that compact the block-wide activemask into compacted warps with thread IDs. The warp buffer (❸) is an instruction buffer that augments each entry with the thread IDs associated with compacted warps.

To retain compatibility with applications that rely on static warp synchronous behaviour (e.g., `reduction` in the CUDA SDK), the thread compactor can optionally be disabled to allow warps to retain their static/compile-time arrangement (e.g., when launching a kernel via an extension to the programming API).

In comparison to DWF [9], thread block compaction accomplishes the lookup-and-merge operation of the "warp LUT" and the "warp pool" [9] with simpler hardware. In DWF, an incoming warp is broken down every cycle and the warp LUT has to locate an entry in the warp pool that can merge with the individual threads. In thread block compaction, warps are only broken down at potentially divergent branches and partial warps are accumulated into block-wide activemasks. The compaction only occurs once after the activemasks have been fully populated and the compacted warps are stored at the warp buffer until the next branch or reconvergence point.

3.3. Example Operation

Figure 9 presents an example of how the hardware in Figure 8 implements thread block compaction. The activemask in the block-wide reconvergence stack is divided into groups, each corresponding to a vector lane in all static warps in the thread block[5]. Threads are constrained to stay in their vector lane during warp compaction to avoid the need to migrate register state and to simplify the thread compactor. Each thread can locate its corresponding bit inside its vector lane group via its associated static warp. For example, thread 5 in the first vector lane of static warp W2 corresponds to the second bit of the first group (❹).

The activemask in the block-wide reconvergence stack is incrementally populated as warps arrive at a branch. Since warps are allowed to execute independently as long as they do not encounter branches or reconvergence points, it is possible for warps to arrive at the branch at the end of block `A` (❶) in an arbitrary order. In the example, warp (W2) arrives at the branch first and creates two target entries on the stack (❷). Each thread in the warp updates one of these entries based upon its branch outcome. For instance, since thread 5 goes to block `C` (❸), it updates the first new entry on the stack (❺). On the other hand, since thread 6 goes to block `B` (❻), it updates the second new entry on the stack (❽). In subsequent cycles, the other warps (W3 and W1) arrive at the branch and update the activemasks of the two new stack entries (❾ and ❿, bits updated at each warp's arrival are shown in bold).

As the warp arrives at the potentially divergent branch, WCnt of the original TOS entry is decremented to keep track of pending warps. When WCnt reaches zero (⓫), the TOS pointer increments to point at the new top target entry. The activemask of this new TOS entry is sent to the thread compactor for warp generation (⓬). WCnt of this entry is also updated to record the number of compacted warps to be generated (calculated by counting the maximum number

[5]Example shows 3 warps, but each thread block can have up to 32 warps/1024 threads [24].

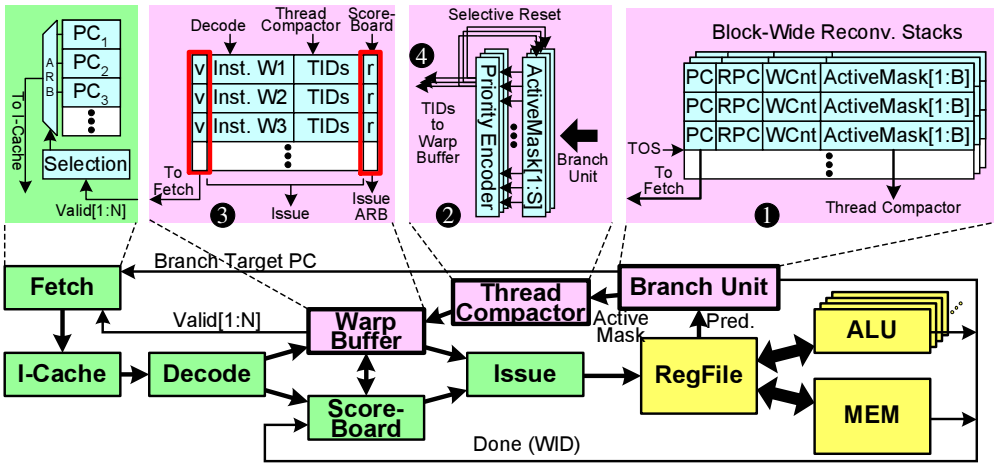


Figure 8. Modifications to the SIMT core microarchitecture to implement Thread Block Compaction. N = #warps, B = maximum #threads in a block. S = B ÷ W where W = #threads in a warp.

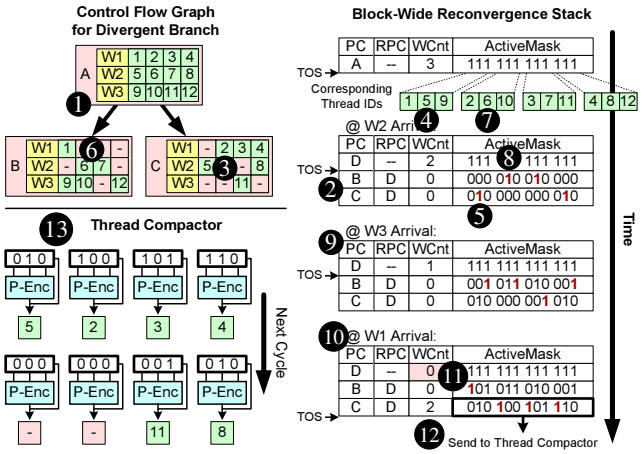


Figure 9. Example showing how the activemask in the block-wide reconvergence stack can be incrementally populated for a divergent branch.

of set bits in all lanes' activemask). A single block-wide activemask can generate at most as many compacted warps as the static warps in the executing thread block.

For branches that NVIDIA's CUDA compiler marks as potentially divergent, we always create two entries for taken and not-taken outcomes when the first warp reaches a branch even if that warp is not divergent. If all threads in a block branch to the same target one of the entries will have all bits set to zero and will be immediately popped when it becomes the top of stack.

Figure 9 also shows the operation of the thread compactor (⓭). The block-wide activemask sent to the thread compactor is stored in multiple buffers, each responsible for threads in a single home vector lane [9]. Each cycle, each priority encoder selects at most one thread from its corresponding inputs and sends the ID of this thread to the warp buffer (❹ in Figure 8). The bits corresponding to the selected threads will be reset, allowing the encoders to select from the remaining threads in subsequent cycles.

When the compacted warps encounter another divergent branch, the process described above repeats, pushing new entries onto the block-wide stack. Eventually, as each of these compacted warps reaches the reconvergence point (block `D` in this example), WCnt of block `C` entry decrements. When this WCnt reaches zero, the entry is popped (TOS pointer decrements). The activemask of the new TOS entry is sent to the thread compactor to generate warps that execute block `B`, and WCnt of this entry is updated accordingly. After these compacted warps have reached the reconvergence point as well, the block-wide stack is popped again, shifting execution to the top-level entry. Similarly, the full activemask of this entry is sent to the thread compactor, with its WCnt updated to the corresponding warp count in preparation for the next divergent branch.

4. Likely-Convergence Points

The post-dominator (PDOM) stack-based reconvergence mechanism [9, 10] uses reconvergence points identified using a unified algorithm rather than by translating control flow idioms in the source code into instructions [15, 7, 2]. The immediate post-dominator of a divergent branch selected as the reconvergence point is the earliest point in a program where the divergent threads are *guaranteed* to reconverge. In certain situations, threads can reconverge at an *earlier point* to improve SIMD efficiency. We believe this observation motivates the inclusion of the `break` instruction in recent NVIDIA GPUs [7].

The code in Example 2 exhibits this earlier reconvergence. It results in the control flow graph in Figure 10 where edges are marked with the probability with which individual scalar threads follows that path. Block `F` is the immediate post-dominator of `A` and `C` since `F` is the first location where *all* paths starting at `A` (or `C`) coincide. In the baseline mechanism, when a warp diverges at `A`, the reconvergence point

Example 2 Example for likely-convergence.

```
while (i < K) {
  X = data[i];      // block A
  if( X = 0 )
    result[i] = Y; // block B
  else if ( X = 1 )// block C
    break;         // block D
  i++;             // block E
}
return result[i];  // block F
```

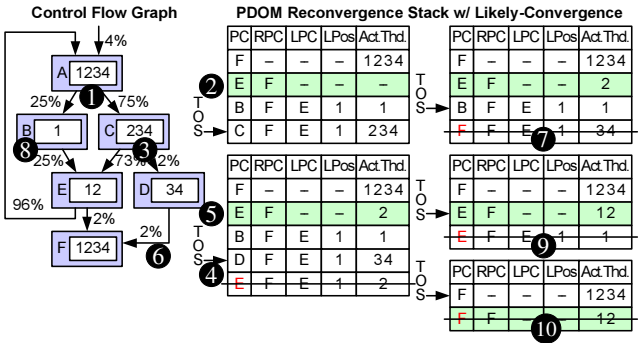


Figure 10. Likely-convergence points improve individual warp SIMD efficiency by reconverging *before* the immediate post-dominator.

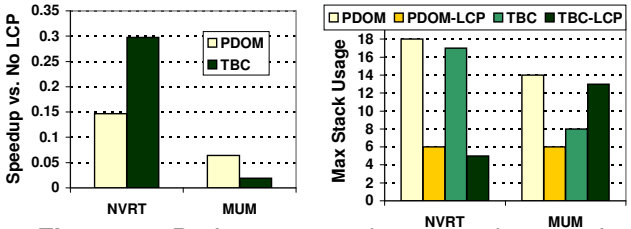


Figure 11. Performance and resource impact of likely-convergence points on both baseline per-warp reconvergence stack (PDOM) and thread block compaction (TBC).

is set to F. However, the path from C to D is rarely followed and hence in *most* cases threads can reconverge at E.

We extend the PDOM reconvergence stack with *likely-convergence points* to capture the potential performance benefits from such "probabilistic" reconvergence. Two new fields per stack entry are added, one for the PC of the likely-convergence point (LPC) and the other (LPos) that records the stack position of a special likely-convergence entry created when a branch has a likely-convergence point that differs from the immediate post-dominator. The likely-convergence point of each branch can be identified with either control flow analysis or profile information (potentially collected at runtime). Figure 10 shows a warp that diverges at A (❶). When the divergence is detected three entries are pushed onto the stack. The first entry (❷) is created for the likely-convergence point E[6]. Two other entries for the taken and fall through of the branch are created as in the baseline mechanism. The warp diverges again at C (❸), and two[7] new entries are created (❹). Execution continues with the top entry until it reaches E, and the likely-convergence point is detected since PC == LPC. When this occurs, the top entry is popped and merged with the likely-convergence entry (❺) as the LPos field indicates. When thread 3 and 4 reach F (❻), since PC == RPC, the stack is popped (❼). Thread 1 then executes B (❽) and its entry is popped at E (❾), when PC == LPC. Finally, the likely-convergence entry executes until it reaches the immediate post-dominator, where it is popped (❿).

Likely-convergence points (LCP) are applicable to both the baseline per-warp reconvergence stack (PDOM) and the block-wide stack used in thread block compaction (TBC). Figure 11 shows the performance impact of extending PDOM and TBC with likely-convergence points (-LCP). Only data for MUM and NVRT are shown because we have only identified likely-convergence points that differ from the immediate post-dominators in these two applications. The impact of LCP for MUM is minimal: 2% speedup for TBC and 5% for PDOM. In contrast, it greatly benefits NVRT: 30% speedup for TBC, 14% for PDOM. Although LCP pushes an extra entry onto the stack for each divergent branch applicable, it can reduce the stack capacity requirement if multiple divergent entries are merged into fewer likely-convergence entries. This happens in most cases, except for TBC running MUM, where the unused likely-convergence entries increases the maximum stack usage.

5. Methodology

We model our proposed hardware changes using a modified version of GPGPU-Sim (version 2.1.1b) [3]. We evaluate the performance of various hardware configurations on the CUDA benchmarks listed in Table 1. Most of these benchmarks are selected from Rodinia [5] and the benchmarks used by Bakhoda et al. [3]. We did not exclude any benchmarks due to poor performance on thread block compaction but excluded some Rodina benchmarks that do not run on our infrastructure due to their reliance on undocumented behaviour of barrier synchronization in CUDA. We also use some benchmarks from other sources:

Face Detection is a part of Visbench [19]. Recommended optimizations [18] for SIMD efficiency were applied.

MUMMER-GPU++ improved MUMMER-GPU [26] reducing data transfers with a novel data structure [11].

NAMD is a popular molecular dynamics simulator [25].

Ray Tracing (Persistent Threads) dynamically distributes workload in software to mitigate hardware inefficiencies [1]. We render the "Conference" scene.

[6] In our experimental evaluation, we restricted likely-convergence points to the closest enclosing backwards taken branch to capture the impact of "break" statements within loops [7].

[7] Since likely-convergence and immediate post-dominator are the same.

Table 1. Benchmarks

Name	Abbr.	BlockDim	#Instr.	Blocks/core
Divergent Set				
BFS Graph Traversal [5]	BFS2	(512x1x1), (256x1x1)	28M	2
Face Detection [19]	FCDT	2x(32x6x1)	1.7B	5,4
HotSpot [5]	HSP	(16x16x1)	157M	2
3D Laplace Solver [3]	LPS	(32x4x1)	81M	6
MUMMER-GPU [3]	MUM	(256x1x1)	69M	4
MUMMER-GPU++ [11]	MUMpp	(192x1x1)	140M	3
NAMD [25]	NAMD	2x(64x1x1)	3.8B	7
Ray Tracing (Persistent Threads) [1]	NVRT	(32x6x1)	700M	3
Coherent Set				
AES Cryptography [3]	AES	(256x1x1)	30M	2
Back Propagation [5]	BACKP	2x(16x16x1)	193M	4,4
Coulumb Potential [3]	CP	(16x8x1)	126M	8
gpuDG [3]	DG	(84x1x1), (112x1x1), (256x1x1)	569M	4,5,6
Heart Wall Detection [5]	HRTWL	(512x1x1)	8.9B	1
LIBOR [3]	LIB	2x(64x1x1)	162M	8,8
Leukocyte [5]	LKYT	(175x1x1)	6.8B	5,5,1
Merge Sort [5]	MGST	(96x1x1) 2x(32x1x1), 2x(128x1x1), 2x(256x1x1), (208x1x1)	2.3B	1,3,8,3,4,2,4
NN_cuda [5]	NNC	(16x1x1)	6M	5,8,8,8
Ray Tracing [3]	RAY	(16x8x1)	65M	3
Stream Cluster [5]	STMCL	(512x1x1)	941M	2
StoreGPU [3]	STO	(128x1x1)	131M	1
Weather Prediction [3]	WP	(8x8x1)	216M	4

Our modified GPGPU-Sim is configured to model a GPU similar to NVIDIA's Quadro FX5800, with the addition of L1 data caches and a L2 unified cache similar to NVIDIA's Fermi GPU architecture [22]. We configure the L2 unified cache to be significantly larger than that on Fermi (1MB vs. 128 kB per memory channel) to make dynamic warp formation more competitive against thread block compaction. In Section 6.4, we explore the sensitivity of both techniques to changes in memory system by reducing the L2 cache to 128kB per memory channel. Table 2 shows the major configuration parameters.

6. Experimental Results

Figure 12 shows the performance of TBC and DWF relative to that of PDOM. TBC uses likely-convergence points whereas PDOM does not and DWF and DWF-WB use majority scheduling [9]. For the divergent (DIVG) benchmark set, TBC has an overall 22% speedup over PDOM. Much of its performance benefits are attributed to speedups on the applications that have very low baseline SIMD efficiency (BFS2, FCDT, MUM, MUMpp, NVRT). While DWF can achieve speedups on these benchmarks as well (except of NVRT, which fails to execute), it also exhibits slowdowns for the other benchmarks that have higher baseline SIMD efficiency (HOTSP, LPS, NAMD), lowering the overall speedup to 4%. DWF with warp barrier (DWF-WB) recovers from most of this slowdown and executes NVRT properly, but loses much of the speedup on MUMpp. Overall, TBC is 17% faster than the original DWF and 6% faster than DWF-WB.

Table 2. GPGPU-Sim Configuration

# Streaming Multiprocessors	30
Warp Size	32
SIMD Pipeline Width	8
Number of Threads / Core	1024
Number of Registers / Core	16384
Shared Memory / Core	16KB
Constant Cache Size / Core	8KB
Texture Cache Size / Core	32KB, 64B line, 16-way assoc.
Number of Memory Channels	8
L1 Data Cache	32KB, 64B line, 8-way assoc.
L2 Unified Cache	1MB/Memory Channel, 64B line, 64-way assoc.
Compute Core Clock	1300 MHz
Interconnect Clock	650 MHz
Memory Clock	800 MHz
DRAM request queue capacity	32
Memory Controller	out of order (FR-FCFS)
Branch Divergence Method	PDOM [9]
Warp Scheduling Policy	Loose Round Robin
GDDR3 Memory Timing	t_{CL}=10 t_{RP}=10 t_{RC}=35 t_{RAS}=25 t_{RCD}=12 t_{RRD}=8
Memory Channel BW	8 (Bytes/Cycle)

Applications in the coherent (COHE) benchmark set are not significantly effected by branch divergence, hence we do not anticipate significant benefits from DWF or TBC. DWF suffers significant slowdowns on some applications in this benchmark set (HRTWL, LKYT, RAY, STO and WP), due to starvation eddy and extra memory stalls from thread regrouping (see Section 6.1). DWF-WB recovers much of this slowdown, however, RAY and WP still suffer from the starvation eddy problem.

Across all benchmarks, TBC obtains an overall 10% speedup over PDOM. The performance benefits of DWF-WB are mostly offset by slowdowns in other applications, making it perform evenly with PDOM.

6.1. In-Depth Analysis

Figure 13(a) and 14(a) show the breakdown of the SIMT core cycle for both DIVG and COHE benchmark sets. At each cycle, the SIMT core can either issue a warp containing a number of active threads (Wn-m means between n and m threads are enabled when a warp issues), be stalled by the downstream pipeline stages (Stall), or not issue any warp because none are ready in the I-Buffer/Warp-Buffer (W0_Mem if the warps are held back by pending memory accesses, and W0_Idle otherwise). This data shows that in some applications (HOTSP, NAMD, HRTWL, MGST, RAY and WP), starvation eddies cause DWF to introduce extra divergence, turning some of the warps with more active threads (W29-32) into warps with fewer active threads. In other applications (e.g. LPS, MUM and NAMD), the extra stalls from DWF undermine the benefit of merging divergent threads into warps. DWF-WB reduces stalls and divergence versus DWF in these applications, but the starvation eddy problem persists with DWF-WB for RAY and WP.

TBC can usually improve SIMD efficiency as well as DWF-WB, and it does not introduce significantly extra stalls or divergences. However, the synchronization overhead at branches can introduce extra W0_idle and

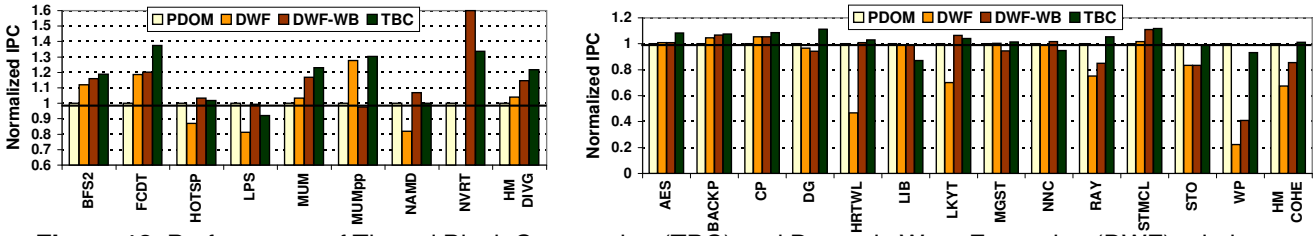
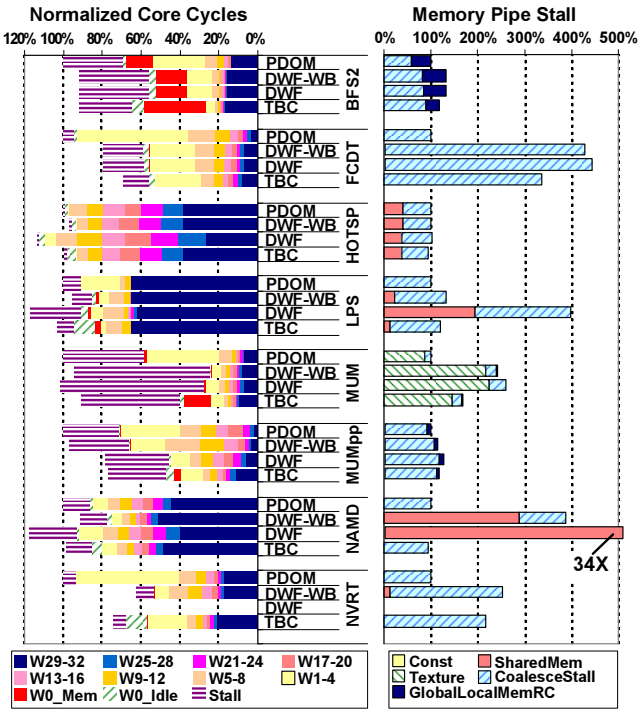


Figure 12. Performance of Thread Block Compaction (TBC) and Dynamic Warp Formation (DWF) relative to baseline per-warp post-dominator reconvergence stack (PDOM) for the DIVG and COHE benchmark sets.



(a) Core Cycle Breakdown (b) Memory Pipe Stall Breakdown

Figure 13. Detail performance data of TBC and DWF relative to baseline (PDOM) for the DIVG benchmark set.

W0_Mem. This is mostly why DWF-WB performs better than TBC for HOTSP, LPS, NAMD and NVRT. For NAMD and NVRT, TBC achieves a lower SIMD efficiency than DWF-WB because DWF can form warps from any threads within a SIMT core, while TBC can only do so from threads within a thread block. BFS, MUM, and MGST transition from compute-bound to memory-bound with TBC (indicated by the extra W0_Mem for them), limiting its benefit.

Figure 13(b) and Figure 14(b) show the breakdown of memory pipeline stalls modeled in our simulator normalized to the baseline and highlights that DWF introduces extra memory stalls. TBC introduces far fewer extra memory stalls. These extra stalls do not out-weight the benefits of TBC to control flow efficiency on several applications (e.g. FCDT, MUMpp, NVRT and HRTWL). Section 6.3 shows how the L1 data cache in each SIMT core absorbs the extra memory accesses generated by TBC, leaving the memory subsystem undisturbed.

6.2. Thread Block Prioritization

Figure 15 compares the performance of TBC among different thread block prioritization policies. A thread block prioritization sets the scheduling priority among warps from different thread blocks, while warps within a thread block are always scheduled with loose round-robin policy [9]:

Age-based (AGE) The warps from the oldest thread block (in the order that thread blocks are dispatched to a SIMT core) have the highest priority. This tries to stagger different thread blocks, encouraging them to overlap each other's synchronization overhead at branches.

Round-robin (RRB) Thread block priority rotates every cycle, encouraging warps from different thread blocks to interleave execution.

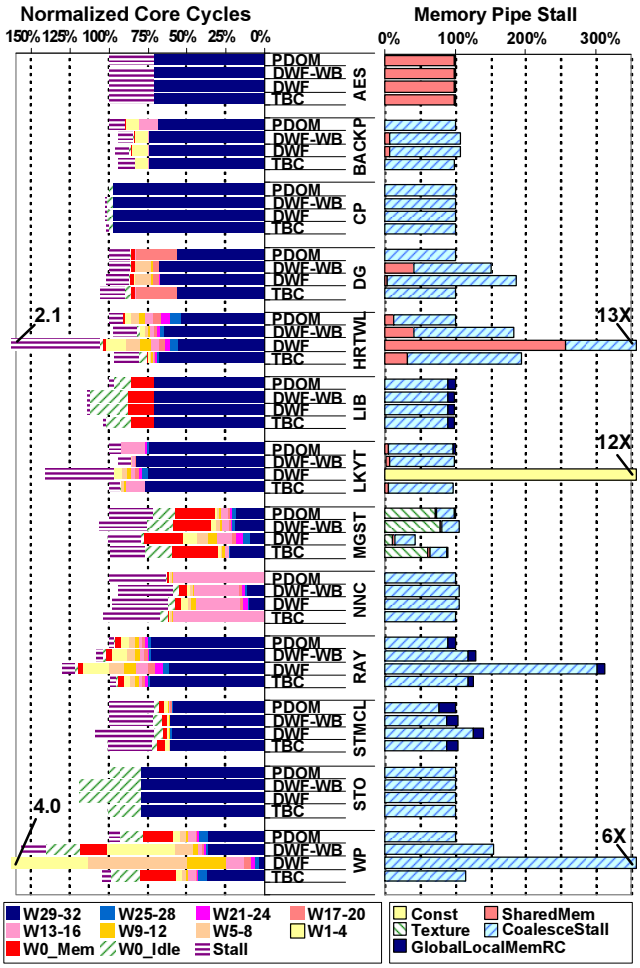
Sticky round-robin (SRR) Warps in a thread block that is currently issuing warps retain highest priority until none of the thread block's warps are ready for issue. Then, the next thread block gets highest priority.

Overall, AGE (default policy for the paper) achieves the highest performance, but it can leave the SIMT cores with a lone-running thread block near the end of a kernel launch. The lack of interleaving reduces overall performance for LPS, NAMD and LIB. RRB encourages even progress among thread blocks, but increases memory system contention in STMCL. SRR provides robust performance.

6.3. Impact on Memory Subsystem

Figure 16 shows the average memory traffic (in bytes for both reads and writes) between a SIMT core and the memory subsystem outside the core with TBC normalized to PDOM. Traffic does not increase significantly with TBC (within 7% of PDOM), indicating that the L1 data cache has absorbed the extra memory pressure due to TBC. If memory accesses from a static warp would have coalesced into a 128B chunk, and this static warp is compacted into multiple dynamic warps by TBC upon a branch divergence, then they will access the same blocks in the L1 data cache.

Memory traffic of DG increases by 2.67X due to increased texture cache misses when using AGE based prioritization (which tends to reduce interleaving from different thread blocks). TBC combined with the RRB policy reduces traffic of DG by increasing the texture cache hit rate.



(a) Core Cycle Breakdown (b) Memory Pipe Stall Breakdown

Figure 14. Detail performance data of TBC and DWF relative to baseline (PDOM) for the COHE benchmark set.

6.4. Sensitivity to Memory Subsystem

Figure 17 shows the speedup of TBC and DWF over PDOM, but with smaller L2 caches (128 kB, 8-way per memory channel). The relative performance between the different mechanisms remains unchanged for the COHE benchmarks. Two of the DIVG benchmarks (MUM and MUMpp) become more memory-bound with a less powerful memory system, lowering the speedup of TBC (15% with RRB and 13.5% with AGE for the DIVG benchmarks). Smaller L2 caches reduce DWF's performance on MUMpp from 28% speedup to 4% slowdown. In comparison, the speedups with DWF-WB and TBC remain robust to the change in the memory system.

7. Implementation Complexity

Most of the implementation complexity for thread block compaction is the extra storage for thread IDs in the warp buffer, and the area overhead in relaying these IDs down the pipeline for register accesses. The register file in each SIMT core also needs to be banked per lane as in dynamic warp formation [9] to support simultaneous accesses from different vector lanes to different parts of the register file.

Table 3. Maximum stack usage for TBC-LCP

DIVG		COHE			
	#Entries		#Entries		#Entries
BFS2	3	AES	1	NNC	3
FCDT	5	BACKP	2	RAY	9
HOTSP	2	CP	1	STMCL	3
LPS	5	DG	2	STO	1
MUM	13	HRTWL	5	WP	8
MUMpp	14	LIB	1		
NAMD	5	LKYT	3		
NVRT	5	MGST	38		

The scheduler complexity that DWF imposes is mostly eliminated via the block-wide reconvergence stacks in thread block compaction. The bookkeeping for thread grouping is done via activemasks in the stack entries. The activemasks can be stored in a common memory array. Each entry in this memory array has T bits (T = max #threads supported on a SIMT core). The T bits in each entry are divided among the multiple thread blocks running on a SIMT core. In this way, the total #bits for activemask payload does not increase over the baseline per-warp stacks.

One potential challenge to TBC is that the block-wide stack can in the theoretical worst case be deeper than with per-warp stacks. Table 3 shows the stack usage for TBC with likely-convergence points across all the applications we study. Most applications use fewer than 16 entries throughout their runtime. For exceptions such as MGST the bottom of the stack could potentially be spilled to memory. We synthesized the 32-bit priority encoders used in thread compactors (sufficient for the maximum thread block size of 1024 threads [24]) in 65 nm technology and found their aggregate area to be negligible ($<< 1mm^2$).

8. Conclusion

In this paper, we proposed a novel mechanism, thread block compaction, which uses a block-wide reconvergence stack shared by all threads in a thread block to exploit their control flow locality. Warps run freely until they encounter a divergent branch, where the warps synchronize, and their threads are compacted into new warps. At the reconvergence point the compacted warps synchronize again to resume in their original arrangements before the divergence. We found that our proposal addresses some key challenges of dynamic warp formation [9]. Our simulation evaluation quantifies that it achieves an overall 22% speedup over a per-warp reconvergence stack baseline for a set of divergent applications, while introducing no performance penalty for a set of control-flow coherent applications.

9. Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported by the National Sciences and Engineering Research Council of Canada.

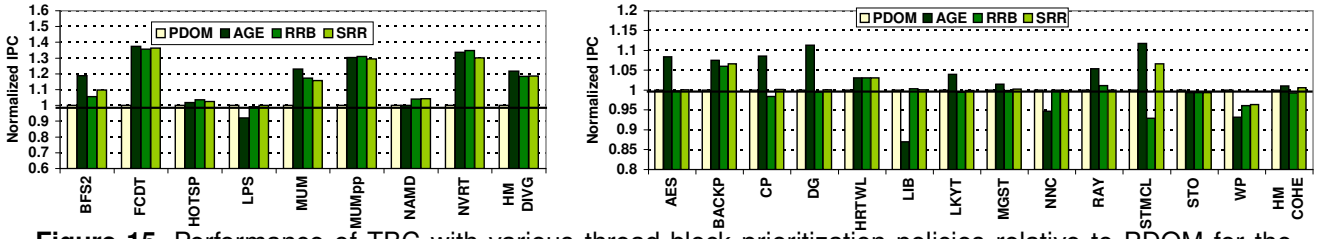


Figure 15. Performance of TBC with various thread block prioritization policies relative to PDOM for the DIVG and COHE benchmark sets.

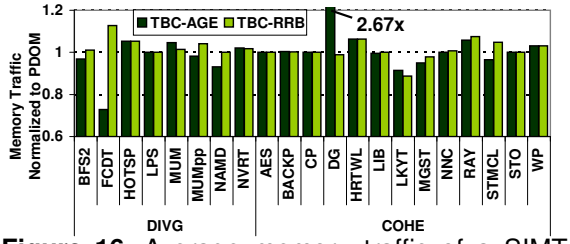


Figure 16. Average memory traffic of a SIMT core for TBC relative to PDOM.

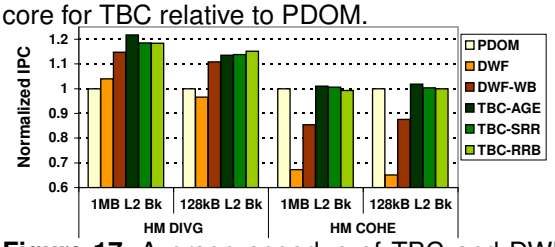


Figure 17. Average speedup of TBC and DWF over PDOM with a smaller (128 kB/Memory Channel) L2 cache. The speedups are normalized to PDOM with the same L2 cache capacity.

References

[1] T. Aila and S. Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *HPG '09*, 2009.

[2] AMD. *R700-Family Instruction Set Architecture*, 1.0 edition, March 2009.

[3] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Int'l Symp. on Perf. Analysis of Systems and Software (ISPASS)*, pages 163–174, April 2009.

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

[5] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 44–54, 2009.

[6] Y.-K. Chen et al. Convergence of Recognition, Mining, and Synthesis Workloads and its Implications. *Proceedings of the IEEE*, 96(5), May 2008.

[7] B. W. Coon et al. United States Patent #7,353,369: System and Method for Managing Divergent Threads in a SIMD Architecture (Assignee NVIDIA Corp.), April 2008.

[8] B. W. Coon et al. United States Patent #7,434,032: Tracking Register Usage During Multithreaded Processing Using a Scorebard having Separate Memory Regions and Storing Sequential Register Size Indicators (Assignee NVIDIA Corp.), October 2008.

[9] W. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. 40th IEEE/ACM Int'l Symp. on Microarchitecture*, 2007.

[10] W. Fung et al. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Trans. Archit. Code Optim.*, 6(2):1–37, 2009.

[11] A. Gharaibeh and M. Ripeanu. Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance. In *IEEE/ACM Supercomputing (SC 2010)*, 2010.

[12] U. J. Kapasi et al. Efficient Conditional Operations for Data-Parallel Architectures. In *Proc. 33rd IEEE/ACM Int'l Symp. on Microarchitecture*, pages 159–170, 2000.

[13] J. H. Kelm et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proc. 36th Int'l Symp. on Computer Arch. (ISCA)*, pages 140–151, 2009.

[14] R. Krashinsky et al. The Vector-Thread Architecture. In *Proc. 31st Int'l Symp. on Computer Arch. (ISCA)*, pages 52–63, 2004.

[15] A. Levinthal and T. Porter. Chap - A SIMD Graphics Processor. In *11th Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 77–82, 1984.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[17] E. Lindholm et al. United States Patent Application #2010/0122067: Across-Thread Out-of-Order Instruction Dispatch in a Multithreaded Microprocessor (Assignee NVIDIA Corp.), May 2010.

[18] A. Mahesri. *Tradeoffs in Designing Massively Parallel Accelerator Architectures*. PhD thesis, University of Illinois at Urbana-Champaign, 2009.

[19] A. Mahesri et al. Tradeoffs in designing accelerator architectures for visual computing. In *Proc. 41st IEEE/ACM Int'l Symp. on Microarchitecture*, pages 164–175, 2008.

[20] J. Meng et al. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proc. 37th Int'l Symp. on Computer Architecture (ISCA)*, pages 235–246, 2010.

[21] J. Nickolls et al. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar.-Apr. 2008.

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

[23] NVIDIA Corporation. *NVIDIA Compute PTX: Parallel Thread Execution ISA Version 1.4*, CUDA Toolkit 2.3 edition, 2009.

[24] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 3.1 edition, 2010.

[25] J. C. Phillips et al. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 2005.

[26] M. Schatz et al. High-Throughput Sequence Alignment Using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474, 2007.