Speculative Reconvergence for Improved SIMT Efficiency

Sana Damani
Georgia Institute of Technology, USA

Daniel Johnson
NVIDIA, USA

Mark Stephenson
NVIDIA, USA

Stephen W. Keckler
NVIDIA, USA

Eddie Yan
University of Washington, USA

Michael McKeown
Esperanto Technologies, USA

Olivier Giroux
NVIDIA, USA

Abstract

GPUs perform most efficiently when all threads in a warp execute the same sequence of instructions convergently. However, when threads in a warp encounter a divergent branch, the hardware serializes the execution of diverged paths. We consider a class of convergence opportunities wherein multiple threads are expected to eventually execute a given segment of code, but not all threads arrive at the same time, resulting in serialized duplicate execution of common code subsequences such as function calls and loop bodies. Our goal is to promote convergence by helping threads that execute common code arrive together before allowing execution to proceed. We propose a new user-guided compiler mechanism, Speculative Reconvergence, to help identify and exploit previously untapped convergence opportunities that increase SIMT efficiency and improve performance. For the set of workloads we study, we see improvements ranging from 10% to 3× in both SIMT efficiency and in performance.

CCS Concepts • **Computer systems organization → Single instruction, multiple data**.

Keywords SIMT efficiency, GPU, thread divergence

ACM Reference Format:
Sana Damani, Daniel Johnson, Mark Stephenson, Stephen W. Keckler, Eddie Yan, Michael McKeown, and Olivier Giroux. 2020. Speculative Reconvergence for Improved SIMT Efficiency. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377911

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377911

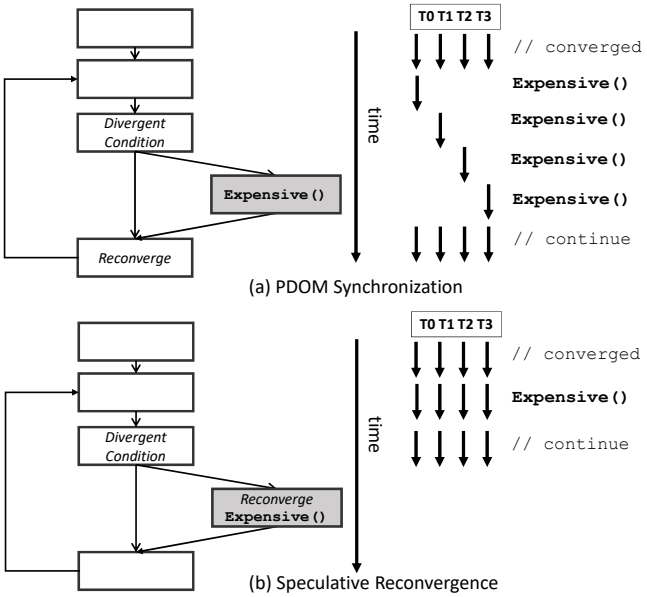


Figure 1. Speculative Reconvergence: an illustration.

1 Introduction

Throughput processors such as Graphics Processing Units (GPUs) achieve high computational density through data parallelism by use of wide single-instruction, multiple-thread (SIMT) pipelines. SIMT efficiency, a measure of parallel execution of threads in a warp, is a key performance metric for GPUs. Control divergence occurs when a warp encounters a branch dependent on a thread-varying value, preventing lock-step execution of conditional code. GPUs handle this by serializing execution of the *then* and *else* paths of the branch, which reduces SIMT efficiency and performance.

In this work, we consider a specific subset of divergence problems: cases where diverged threads execute common subsequences of code. In particular, multiple threads within a warp might execute a given piece of code eventually, but not all threads may arrive at the same point in time. This pattern is common in Monte Carlo and ray tracing applications as we discuss later. Our objective is to promote convergence by helping threads that will execute common code arrive

together before allowing execution to proceed. We focus on a set of specific divergence sub-problems, which can broadly be divided into (1) common code across loop iterations, and (2) common function calls across divergent paths.

Figure 1(a) illustrates case (1). The loop body contains a divergent branch that potentially evaluates to true in a different iteration for each thread. As GPU compilers currently attempt reconvergence at the post-dominator (PDOM) of the divergent region, some threads execute `Expensive()`, while the remaining threads wait idle at the *then* part's predecessor or post-dominator. The common code is therefore executed serially across threads as the cartoon execution diagram shows. Instead of reconverging at the post-dominator of the branch where all threads are guaranteed to arrive before executing the next iteration, threads can wait at the *then* block for other threads to arrive before executing common code as in Figure 1(b). This technique enables convergent execution of the "expensive" common code by deferring the progress of some threads within the warp, and improves both the SIMT efficiency and the total runtime for the program in question.

Our solution uses Speculative Reconvergence to improve SIMT efficiency by collecting threads across loop iterations, or across divergent code paths, before executing common code paths. We propose a user-guided approach to identify common code paths within the program and a compiler solution to manage thread reconvergence. Our approach is particularly effective for improving Monte Carlo simulations, which have a wide variety of real world applications including dose calculation for cancer treatment [18], financial option pricing [4], path tracing [14] and experimental particle physics [5].

Even though postdominator-based synchronization minimizes the static region within which threads diverge in the program, we show how it fails at times to achieve optimal warp execution efficiency because it conservatively synchronizes a warp only at locations where all threads are guaranteed to arrive. Our approach enables reconvergence of all threads that may arrive at an expensive code path. Our contributions include:

- Identification of the class of divergence problems that benefit from alternative reconvergence.
- A unified approach for inserting user-specified reconvergence points for improved convergence.
- A compiler algorithm that inserts synchronization operations based on user-specified reconvergence points.
- A *soft barrier* to collect a subset of threads before executing common code.
- An evaluation of our approach on a modern GPU.

2 Background

The end of Dennard scaling has led to a surge in domain specific architectures, languages, compilers and focused optimizations such as our Speculative Reconvergence approach. At a high level, programmable graphics processing units (GPUs) from different vendors operate according to a similar set of principles. Due to prevalence in the literature, we standardize on NVIDIA terminology, but note that concepts generally apply across architectures.

GPUs organize co-scheduled sets of independent threads into groups called *warps*. Warps issue instructions in a Single Instruction, Multiple Thread (SIMT) manner. Whereas Single-Instruction, Multiple-Data (SIMD) architectures apply one instruction to a fixed vector of data elements using a single thread, SIMT applies one instruction to multiple independent threads in parallel. Thus, GPUs allow for independent thread execution and branching [17].

When data-dependent conditional code is encountered on SIMD architectures, predication may be used to disable execution of certain data paths. SIMT architectures on the other hand allow each thread to branch independently depending on the evaluation of the branch condition. This thread divergence is supported by masking off the execution of a subset of threads where the hardware serializes the execution of the taken and not-taken paths of the branch. When their paths merge, the compiler reconverges the threads in the warp so that they resume parallel execution. Performance in SIMT architectures relies on maximizing parallel execution of code by multiple threads within a warp, which we measure using a metric known as *SIMT efficiency*, the average percentage of active threads per warp.

While pre-Volta GPUs use a stack based mechanism to handle nested control divergence [17], Volta introduces the independent thread scheduling model which allows for finer grained synchronization patterns by maintaining per-thread state information. Further, Volta's convergence optimizer maximizes SIMT efficiency by grouping together threads that execute the same code in parallel for maximum convergence.

The Volta ISA provides synchronization instructions that enable implementation of independent thread scheduling [6, 20, 21], which we use to implement Speculative Reconvergence: *BSSY*, *BSYNC*, and *BREAK*. Threads that encounter a *BSSY* join a convergence barrier which is represented using a barrier register. *BSYNC* synchronizes all threads that previously joined the corresponding convergence barrier. *BREAK* removes a thread from the specified convergence barrier, a mechanism used to handle non-standard convergence points such as in loops or in short-circuit code.

As described in [6], the Volta GPU architecture guarantees forward-progress by ensuring that no thread is suspended indefinitely, to allow for arbitrary synchronization without resulting in a live-lock state. It achieves this by enabling the GPU to yield execution of threads. Furthermore, unlike barriers used for synchronization such as *__syncthreads*, these barriers optimize performance by minimizing divergent execution of code and are not required for correctness of the program.

3 Problem

There are two broad categories of control flow divergence: divergent threads may execute disjoint sequences of code with no commonality, or divergent threads may eventually execute some common code sequences, just not aligned in time. For this work, we focus on the second category: periods of divergent execution where threads within the same warp might eventually execute common sequences of code, but are not able to exploit this commonality today. Broadly, two subclasses of patterns exhibit this behavior:

- Code that is common across diverged paths, including common function calls, common instruction subsequences, or irregular control flow.
- Divergently executed code that is common across loop iterations, such as a divergent branch or an imbalanced loop nested inside a loop.

Figure 2 illustrates several example code patterns which exhibit regions of divergent control flow that execute common code. Typical compiler-inserted reconvergence points located at post-dominators do not necessarily maximize runtime convergence for these cases. Instead of waiting for guaranteed reconvergence at post-dominators, threads could instead reconverge opportunistically at earlier points of execution. In each example, an alternative synchronization pattern can improve runtime convergence and performance. Our goal is to find a way to enable these common code sections to execute convergently to improve execution efficiency.

Divergent Conditions in Loops. Figure 2(a) shows a loop containing a divergent branch. In this example, multiple threads eventually execute the expensive code in the *then* part but do so at different iterations of the loop. The standard reconvergence logic collects all threads at the post-dominator of the divergent region before executing the next loop iteration, and the execution of the expensive conditional code remains serialized across loop iterations, resulting in poor GPU utilization.

Figure 1(b) shows an alternative execution that employs a different synchronization pattern that collects multiple threads across different iterations of the loop before executing conditional code inside the divergent path. Threads wait at the start of the expensive region for all threads to arrive before executing convergently, resulting in improved SIMT efficiency within the conditional and minimal thread idle time. In this example, the condition has no *else* part to execute. In cases where multiple paths exist, we may prefer to reconverge for expensive paths while allowing diverged execution of less expensive paths. We call this alternative synchronization approach *Iteration Delay*.

Loop Trip Count Divergence. Figure 2(b) shows a variation of unnecessary serialization where a loop contains a nested loop with a divergent trip count, i.e. threads exit the inner loop after different numbers of iterations. Threads that exit the inner loop early wait for all threads to arrive before executing the epilog and beginning the next iteration of the outer loop, where threads once again begin to execute the inner loop in a convergent manner. The inner loop body is therefore common across iterations of the outer loop.

Programs that have a non-nested divergent loop may be modified using thread coarsening, i.e. combining work from multiple threads into a single thread by converting a loop into nested loops which can then be optimized as described above. This situation is common in CUDA applications where, unlike in software-driven work distribution, users specify many thousands of independent tasks, and the GPU scheduler distributes work across SMs.

Figure 3 depicts the high-level pseudocode for RSBench, a mini-app that represents the packed-data multipole macroscopic cross section lookup kernel of the Monte Carlo neutron transport algorithm [13, 26]. Given a material and energy, the kernel walks over all the nuclides in the material that the neutron is travelling through and computes the sum of their cross-section data. This computation forms the inner loop of our example. We use thread-coarsening, a technique that merges threads, to create the outer loop that walks over multiple materials per thread. Hence, instead of a single variable length task per thread, we assign a large number of tasks per thread to enable load balancing over time. This transformation also gives us the code pattern required for Speculative Reconvergence, similar to Figure 2(b).

Traditional divergence handling tries to reconverge at the earliest possible point where all threads are guaranteed to arrive, i.e. at the inner loop post-tail, which means that the warp executes some iterations of the inner loop divergently.

Instead, we execute the inner loop convergently by collecting threads across iterations of the outer loop, especially if the inner loop body is expensive, a technique we call *Loop Merge*. However, as shown in Figure 3(b), while this alternative execution sequence ensures parallel execution of expensive common code, it changes the convergence properties of the code outside the divergent region. The prolog and epilog regions of the loop, which were previously executed in a convergent manner, are now executed divergently. Hence, the profitability of our solution depends on program properties, such as the cost of the common code and the cost of executing the prolog and epilog divergently.

Common Function Call. Figure 2(c) shows an example of common code across diverged paths of a non-uniform branch. Both the taken and not taken paths of the branch call *foo()*, i.e., the function call body is common across all paths of the divergent branch. All threads in the warp will eventually execute *foo()*, but not together. The compiler today fails to reconverge at the function body because the calls to *foo()* are made from different locations in the program, and existing post-dominator based analysis fails to recognize this function body as a potential reconvergence point. To improve SIMT efficiency within the function body, we ensure that all threads reconverge at the start of the function

(a) Divergent condition within loop

```
for (i = 0; i < N; i++)
{
  Prolog()
  if (divergent_condition())
  {
      // proposed reconvergence point
      Expensive()
  }
  // original reconvergence point
  Epilog()
}
```

(b) Loop trip count divergence

```
for (i = 0; i < N; i++)
{
  Prolog()
  for (j = 0; divergent_condition(); j++)
  {
      // proposed reconvergence point
      Expensive()
  }
  // original reconvergence point
  Epilog()
}
```

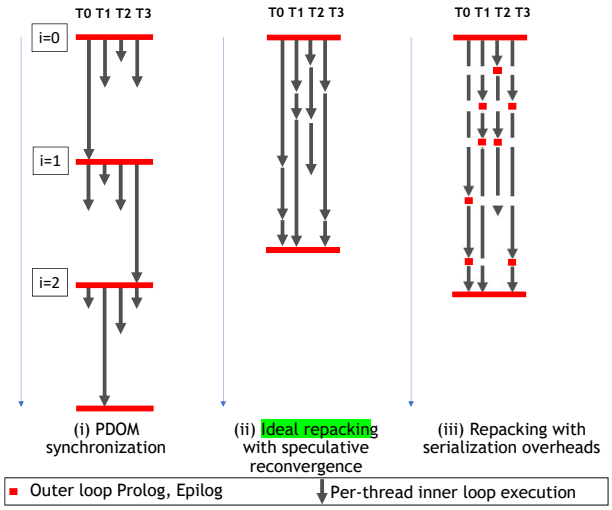
(c) Common function call

```
foo()
{
  // proposed reconvergence point
  Expensive()
}
main()
{
  if (divergent_condition()) { foo(); }
  else { ... foo(); ... }
  // original reconvergence point
}
```

Figure 2. Pseudocode for motivating use cases.

```
RSBench_lookup_kernel() {
  while (true) {
    Prolog:
      material = get_random_material()
    Predict(L1)
    // num nuclides per material ranges from 4 to 321
    for (each nuclide in material) {
      // proposed reconvergence point
      L1:
        accumulate_neutron_cross_sections()
    }
    // original reconvergence point
    Epilog:
      post_processing()
  }
}
```

(a) RSBench: pseudocode [26].



(b) Sample execution pattern for RSBench.

Figure 3. Pseudocode and execution pattern for RSBench.

and execute the body in parallel. Unlike previous examples, reconvergence within the function body does not conflict with the compiler inserted reconvergence point at the post-dominator, nor does it affect convergence properties of the code outside the function body.

Opportunity. As illustrated in Figure 2, each of these examples has a particular portion of the code where threads could execute convergently, but are not converged today. Typical reconvergence at post-dominators does not always capture maximum convergence in these examples. We aim to enable convergent execution of these common code sections to improve execution efficiency when profitable.

4 Compiler Assisted Speculative Reconvergence

Convergence properties for code patterns like those described in Section 3 can be improved by enabling reconvergence in regions of common code. We propose a framework for achieving improved execution convergence by enabling placement of reconvergence hints at non-standard locations, i.e. places where reconvergence is possible, but is not guaranteed. This section describes compiler techniques to implement our proposal to systematically insert and modify synchronization barriers to influence convergence properties.

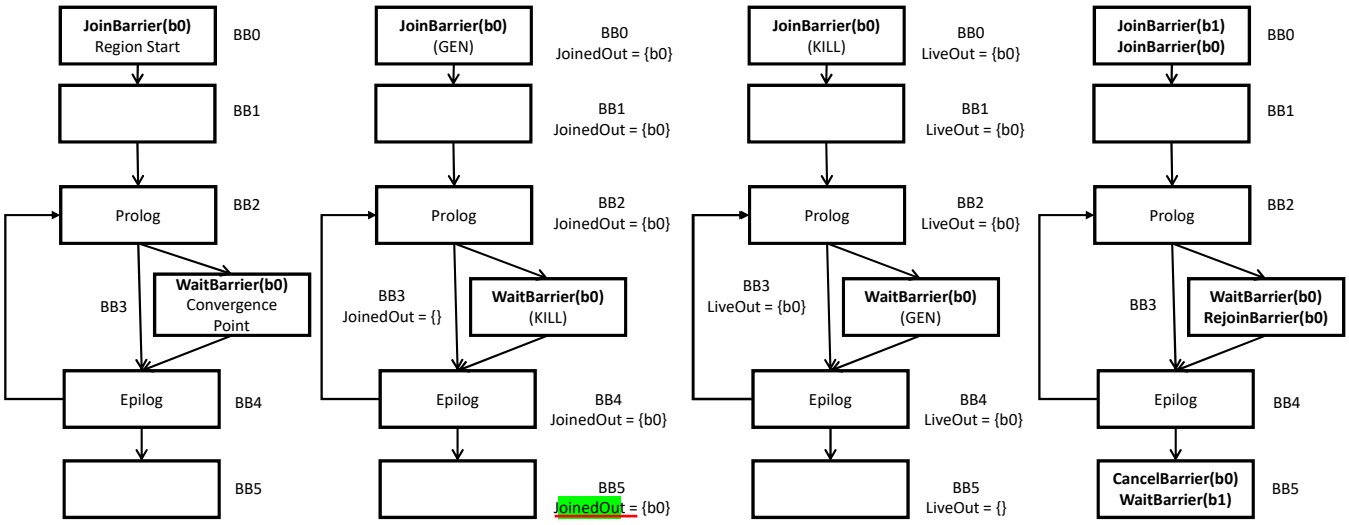
Our approach operates in two steps: (1) find the region and program point where threads should reconverge (Section 4.1); and (2) insert synchronization instructions that achieve convergence before this point and allow exiting threads to withdraw from the barrier (Section 4.2).

4.1 Reconvergence point

To specify reconvergence hints, we require two pieces of information that the compiler uses to automatically place convergence synchronization:

1. The predicted location for reconvergence.
2. The region of code where this prediction should apply.

The first element allows a potential reconvergence point to be specified. For user-directed reconvergence, this location is specified by the user via a label in the source code. The second element describes which threads are considered



(a) Synchronization Hints (b) Joined Barrier Analysis (c) Barrier Liveness Analysis (d) Synchronization Instructions

Figure 4. CFG representation for Iteration Delay style synchronization for Listing 1.

Listing 1. User Inserted Reconvergence Points.

```
1  Predict(L1)
2  for (i = 0; i < N; i++)
3  {
4      Prolog()
5      if (divergent_condition())
6      {
7        // User-specified reconvergence point
8        L1:
9          Expensive()
10     }
11     // Original reconvergence point
12     Epilog()
13 }
```

to be candidates for reconvergence at the predicted location. Threads that enter the region will attempt to honor the predicted reconvergence point, and threads that leave the region are no longer considered candidates for reconvergence. This *Prediction Region* is specified by a `Predict(<label>)` directive that associates the prediction with a specific labeled location for reconvergence. The region ends where all threads are no longer able to reach the label. In Listing 1, `Predict(L1)` marks the start of the prediction region. This information is used to disambiguate between predictions at different loop nesting levels. Label `L1` marks the user-provided reconvergence point where all threads wait before executing the code that follows.

These two pieces of information help convey the following objectives to the compiler:

- All threads within the region should reconverge, when possible, at the predicted reconvergence point.
- Threads that leave the region should withdraw from the barrier at the predicted location, so other threads do not wait forever.

Table 1. Synchronization Primitives. Each primitive references a named barrier.

Synchronization Primitive	Description
`JoinBarrier<`*barrier*`>`	Threads that enter the barrier expect to wait on the barrier at a subsequent convergence point. Implemented using BSSY.
`WaitBarrier<`*barrier*`>`	Threads wait on all participating threads to arrive before clearing the barrier and continuing execution. Implemented using BSYNC.
`CancelBarrier<`*barrier*`>`	Threads that exit the region without waiting on the barrier must clear the barrier so other threads may continue execution. Implemented using BREAK.
`RejoinBarrier<`*barrier*`>`	We introduce a new primitive to denote the location where threads that have cleared a barrier but expect to wait on the barrier again must rejoin the barrier. This is seen primarily in the case of loops. Implemented using BSSY.

- The user-specified convergence hints should receive priority over any standard GPU convergence synchronization that might conflict, such as compiler-inserted reconvergence points at branch post-dominators.

The reconvergence point and prediction region may be specified by the user or discovered automatically by the compiler or a tool. For this work, we focus primarily on user-directed convergence, but describe automatic compiler heuristics in Section 4.5.

4.2 Synchronization Algorithm

Figure 4 shows the Control Flow Graph (CFG) for Listing 1. The user-defined region start and reconvergence points are at blocks BB0 and BB3, respectively. Compiler primitives are defined in Table 1. We now describe an algorithm that inserts these synchronization primitives.

The insertion of `JoinBarrier` and `WaitBarrier` primitives is trivially achieved using user-defined or compiler-detected region start and reconvergence points described in Section 4.1. The compiler preserves this information through any control-flow altering optimizations prior to the synchronization pass. Figure 4(a) shows the insertion of `JoinBarrier` and `WaitBarrier` at BB0 and BB3 respectively.

The compiler must ensure that a thread that waits on a barrier that it has already cleared rejoins the barrier and a thread that exits the region without waiting on a barrier that it has joined withdraws from the barrier. In the next section, we detail program analyses that we use to find locations for the insertion of `RejoinBarrier` and `CancelBarrier` primitives.

4.2.1 Dataflow Analysis

We need two pieces of information to decide the placement of `CancelBarrier` and `RejoinBarrier` primitives: (1) whether a thread is part of an uncleared barrier, and (2) whether a thread expects to wait on the barrier. Figures 4(b) and (c) depict the data flow analyses the compiler uses to decide ideal placement of synchronization primitives, and Figure 4(d) shows the final synchronization. We insert a `RejoinBarrier` in BB3 where the barrier *b0* was cleared by a `WaitBarrier`, and a `CancelBarrier` in BB5 where threads that joined barrier *b0* may escape without clearing the barrier. Finally, to ensure that all threads reconverge at the region exit, we introduce an orthogonal barrier register and insert a pair of `JoinBarrier` and `WaitBarrier` primitives at the dominator (BB0) and post-dominator (BB5) of the region respectively.

Joined Barrier Analysis. A barrier register has been joined at a program point P if at least one path from program start to P contains a `JoinBarrier` primitive not followed by a `WaitBarrier` primitive. Equation 1 depicts the dataflow equations for joined barrier analysis. In Figure 4(b), the barrier at BB3 is joined at BB0 and cleared at BB3.

$$\begin{aligned} Gen(BB) &= JoinBarrier \\ Kill(BB) &= WaitBarrier \\ IN(BB) &= \cup(OUT(p) \mid \forall p \in predecessors(BB)) \\ OUT(BB) &= (IN(BB) - Kill(BB)) \cup Gen(BB) \end{aligned} \tag{1}$$

Barrier Live Range Analysis. Next, we perform standard backward liveness analysis on barrier registers to compute the set of blocks where the barrier is live. Barrier *liveness* is analogous to register liveness, i.e. a barrier is *live* at a program point P if there is a `WaitBarrier` along some path from P to the end of the program. Otherwise it is considered dead, i.e. it has no further uses. We generate the liveness of the barrier register at a use of the barrier (`WaitBarrier`), and kill the liveness of the barrier register at the `JoinBarrier`. Equation 2 depicts the dataflow equations for live barrier analysis. In Figure 4(c), the barrier *b0* is dead at BB5 and BB0.

$$\begin{aligned} Gen(BB) &= WaitBarrier \\ Kill(BB) &= JoinBarrier \\ IN(BB) &= (OUT(BB) - Kill(BB)) \cup Gen(BB) \\ OUT(BB) &= \cup(IN(s) \mid \forall s \in successors(BB)) \end{aligned} \tag{2}$$

Note that `CancelBarrier` and `RejoinBarrier` primitives can affect the dataflow analyses described above, but these primitives are not yet present in the code and are therefore ignored in our equations.

4.3 Deconfliction

Two barriers are said to be conflicting if their live ranges overlap in a non-inclusive manner, i.e. neither one is a complete subset of the other. If a region has conflicting barriers, threads may wait for each other at two different places within the region resulting in unpredictable behavior. The compiler inserts synchronization primitives that may conflict with the compiler-inserted synchronization at the postdominator of the divergent region. To break this conflict and avoid unpredictable behavior, we implement a *deconfliction pass*.

A barrier live range extends from the moment threads join the barrier until the barrier is cleared either by waiting or exiting threads. In Figure 5(a), the blue arrows represent the two live intervals for barrier b0 which was inserted by our optimization, while the red arrow represents the live interval for barrier b1, inserted by the compiler at the region postdominator. Barrier b0 and b1 conflict because their respective live intervals overlap in a non-inclusive manner.

We implemented two deconfliction strategies: *static deconfliction* and *dynamic deconfliction*. In *static deconfliction*, if two barriers are found to conflict, the compiler eliminates barrier operations corresponding to the PDOM barrier. If the newly defined convergence point is never, or rarely, entered, static deconfliction may result in poor performance. Figure 5(b) shows static deconfliction as applied to our example. The synchronization primitives for b1 are removed thereby eliminating the conflict. In *dynamic deconfliction* on the other hand, no instructions are deleted. Instead, threads waiting on one barrier exit out of the conflicting barrier, thereby removing conflicts only in cases where the convergence point is in fact executed at run time. In our example, Figure 5(c), threads that enter block BB3 exit barrier b1 before waiting on barrier b0, thus effectively eliminating the conflict at runtime.

Static deconfliction has an advantage over dynamic deconfliction in terms of number of instructions executed and barrier registers used. However, if a conditional branch is rarely executed, and the prolog/epilog sections are expensive, dynamic deconfliction performs better because it retains the original synchronization points.

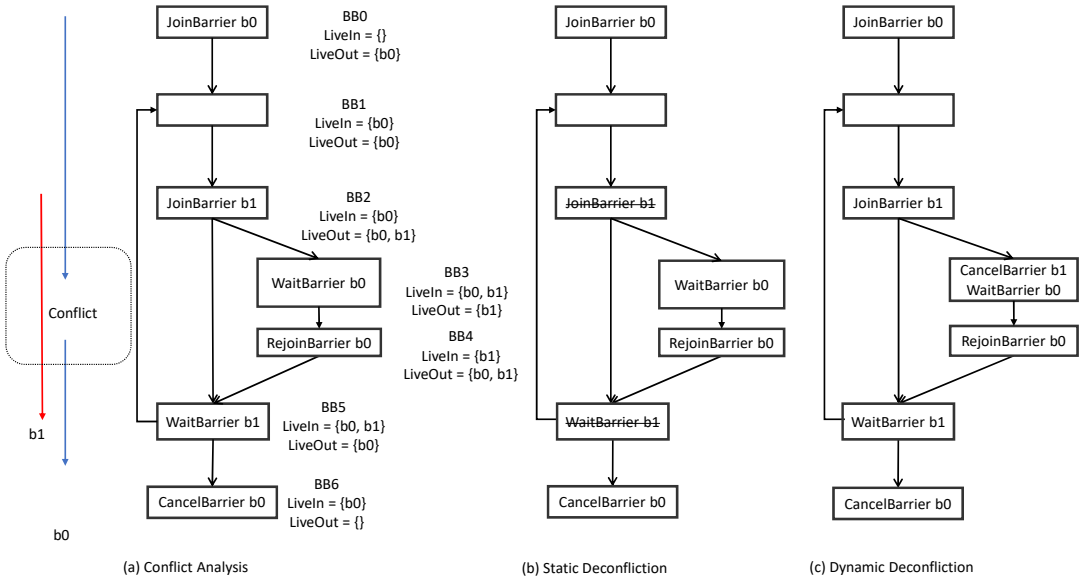


Figure 5. Deconfliction: (a) Conflict analysis; (b) Static deconfliction by deleting conflicting barrier; (c) Dynamic deconfliction by exiting conflicting barrier.

4.4 Interprocedural Analysis

As an extension to our optimization, we propose an interprocedural variant that handles a function body eventually executed by all threads in a warp. In Figure 2(c), foo() is ultimately called from both paths of the divergent branch, but serially. To indicate the desired early reconvergence point at the entry of the function, the user interface now uses the function name instead of a label name to indicate the PC at which all threads must wait before executing in parallel.

Speculatively reconverging within the divergent function call rather than at the post-dominator block of the divergent condition does not adversely affect performance because there are no prolog/epilog sections and hence no increase in divergent execution. The only cost associated with the optimization is the insertion of additional barrier instructions.

Finally, the programmer or the compiler must move calls to extern functions into a *wrapper function* body which acts as the required reconvergence point. The wrapper function may also be used for functions that are called from within multiple independent regions of the program.

To perform this optimization across functions, we introduce an interprocedural analysis that propagates barrier information upwards through the call graph from the callee to the call site. After this first stage propagation is complete, the barrier operations are inserted by the compiler as before.

4.5 Automatic Detection of Reconvergence Point

In some cases, such as programs generated by parallelizing compilers, manual modification may not be possible. In this case, automatic detection of some common code patterns may be useful. Our approach looks for opportunities within common CFG patterns in code such as divergent branches or loops with non-warp-uniform iteration counts nested inside an outer loop.

We use three metrics to approximate the cost-benefit analysis of changing convergence patterns. First, we compute *instruction count* in prolog/epilog sections, weighted by loop trip count, nest depth, and instruction latency. Static analysis is limited by its inability to predict dynamic loop counts and caching behavior, rendering it too conservative. Profile information may help improve the accuracy of our profitability tests. Next, *memory access patterns*, such as convergent memory accesses may suffer if previously convergent accesses become divergent due to our optimization. Heuristics may choose to account for the cost of making memory accesses divergent. Finally, *synchronization requirements* within the program including barriers and warp synchronous instructions may affect the correctness of modifying the convergence properties of the region.

Automatic Speculative Reconvergence is a challenging problem for the compiler to solve unaided, particularly when there are conflicting locations for Speculative Reconvergence such as in the case of triply nested loops. Further, the profitability of modifying convergence points depends on the relative cost of the common code, its divergence properties, and the prolog/epilog regions of the code. Incorrect Speculative Reconvergence may result in large performance degradations, depending on the run time behavior of the program, which is why we rely on the user's understanding of application behavior to decide the best reconvergence point and provide a compiler technique to correctly insert the necessary synchronization primitives.

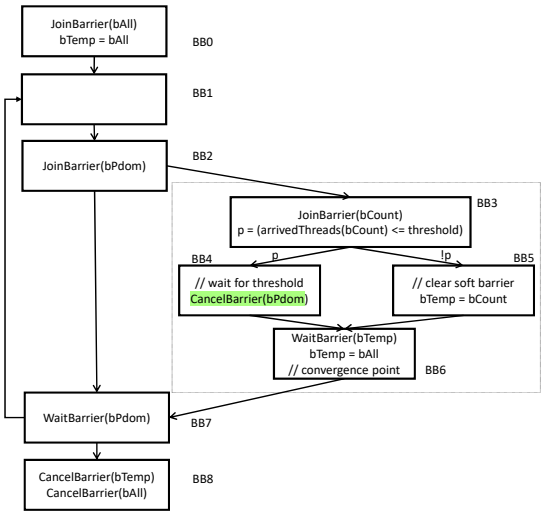


Figure 6. CFG representation for compiler-inserted synchronization primitives to enable soft barrier.

4.6 Soft Barrier

As described so far, we wait for all threads to arrive at a predicted convergence point before proceeding, thereby maximizing reconvergence at the specified location. However, maximizing convergence may not always be the best choice. As we increase convergence at the specified location, we can also increase serialized executions of other portions of code (e.g., of the prolog and epilog code sections in Figure 3). In the case of Loop Merge, depending on the relative cost of these sections with the inner loop, this alternate synchronization pattern may not be profitable. Earlier work such as [11, 23] have made similar observations about the utility of a tuning parameter to balance idle lanes versus the cost of refilling with work. To help address this tradeoff, we provide an optional threshold parameter to ensure that at least some minimum set of threads arrive at the reconvergence point before proceeding, but also allow threads to proceed without having to wait for all possible participants. This approach guarantees some minimum degree of convergence at the specified location, while ensuring that newly serialized code regions have their executions amortized across more threads.

Figure 6 shows a detailed view of our Loop Merge example with soft barriers. We use barrier `bCount` to keep a count of all threads that arrive at the user-defined convergence point. Threads that wait for the threshold to be met must exit barrier `bPdom` to allow the remaining threads to continue execution. Once the threshold is met and sufficient threads arrive at the reconvergence point, all collected threads must execute the conditional code in parallel. We use `bTemp` to synchronize within the set of threads that fulfill the threshold requirement. In case the threshold is not satisfiable, the threads waiting at BB6 can proceed once the remaining threads have opted out of barrier `bTemp` at the loop exit. `bAll` collects all threads at the loop exit so the remaining code executes convergently.

Table 2. Benchmarks.

Benchmark	Description
rsbench	A nuclear reactor simulation mini-application that optimizes Monte Carlo neutron transport [13]. The main kernel in RSBench has a loop with a divergent trip count. We apply thread coarsening to increase work per thread.
xsbench	[27] simulates a problem similar to RSBench, but is memory bound rather than compute bound. In particular, we find that the nested divergent loop in the XSBench kernel has both an expensive inner loop and an expensive epilog.
mcb	A Monte Carlo benchmark used to test performance of parallel architectures. Simulates a simplified variant of the heuristic transport equation [16].
pathtracer	A simple CUDA-based microbenchmark that renders a sample scene composed of spheres in a Cornell box. Has loop trip count divergence.
mc-gpu	A GPU-accelerated Monte Carlo simulation used to model radiation transport of x-rays for CT scans of the human anatomy [3].
mummer	A parallel sequence alignment kernel used for genome sequencing [25].
MeiyaMD5	Performs Message-Digest algorithm 5 (MD5) hash reverses [29].
Optix	NVIDIA's ray tracing engine optimized to achieve high performance for ray tracing based algorithms on parallel architectures. [23].
gpu-mcml	A benchmark that simulates photon transport [2].

5 Evaluation

We modified the NVIDIA production GPU compiler to evaluate the proposed Speculative Reconvergence technique as described in Section 4. We implemented two variants of this optimization: (1) a user-guided approach for marking reconvergence points (Section 4.1) and (2) compiler heuristics to detect divergence patterns (Section 4.5). Further, our implementation uses the *dynamic deconfliction* strategy. Our results were collected on NVIDIA's Volta V100 GPU using the nvprof profiler [22].

5.1 Benchmarks

We evaluate our proposal on a set of divergent workloads that exhibit the patterns described in Section 3. Many of these are Monte Carlo applications that model processes with variable and unpredictable length, leading to poor SIMT efficiency and reduced performance due to divergent code in loops. Table 2 describes the set of applications we evaluate. We did not find any applications that exhibit the common function call pattern described in Figure 2(c); instead, we validated this pattern using microbenchmarks.

5.2 Results: Programmer-Annotated Applications

For our evaluation, we manually insert reconvergence points and mark the prediction region as described in Section 4.1. The primary metric we influence in this work is SIMT efficiency, a measure of convergent execution of threads in a warp. Figure 7 shows SIMT efficiency before and after our transformations. Many of these applications exhibit relatively low SIMT efficiency in their default state, indicating potential for improvement. After modifying reconvergence points to user-specified alternate locations, we see significant increases in SIMT efficiency. Applications such as gpu-mcml, pathtracer, and rsbench have highly variable inner loop trip

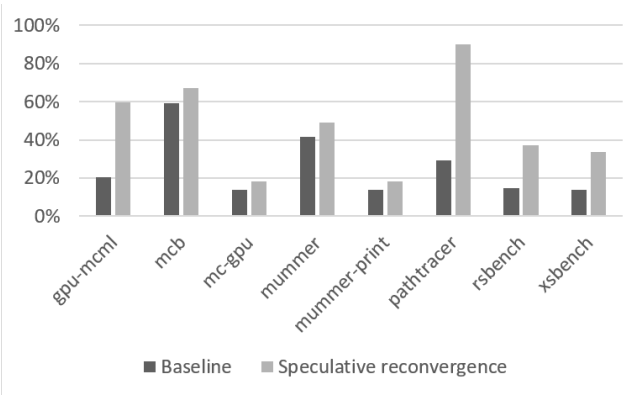


Figure 7. SIMT efficiency.

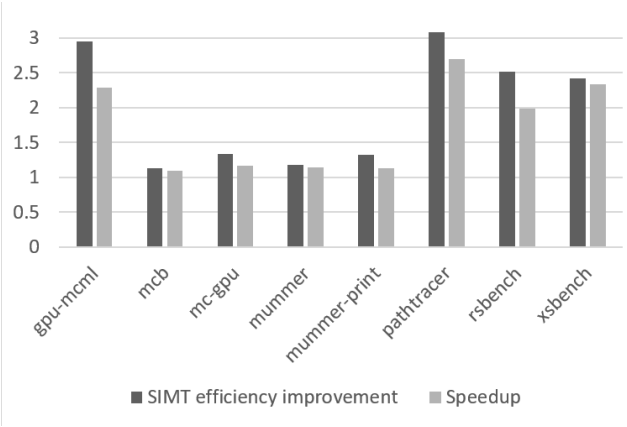


Figure 8. SIMT efficiency versus speedup.

counts, resulting in significant divergence with default synchronization; all threads within a warp must wait for any straggler threads still executing the inner loop before they can proceed. After marking a reconvergence point inside the loop instead of at the loop post-dominator, idle threads are able to diverge and acquire new work before rejoining the inner loop. SIMT efficiency is improved most when threads have a relatively high degree of compute inside their loops compared with the cost of newly-serialized code before and after the loop to refill empty threads.

While the primary metric that our transform attempts to improve is SIMT efficiency, our end goal is improved performance. Figure 8 shows our relative improvement in SIMT efficiency as well as application speedup. As expected, we see that improvement in SIMT efficiency leads to improvement in performance. However, there are many aspects that affect runtime performance beyond SIMT efficiency. While we improve overall SIMT efficiency, especially in the compute-intensive portions of code, the tradeoff is that we must now pay for any additional runtime overhead where we execute other regions of code (e.g., prolog or epilog) more divergently and more frequently. For example, each RSBench thread processes materials of different cost in nuclide count,

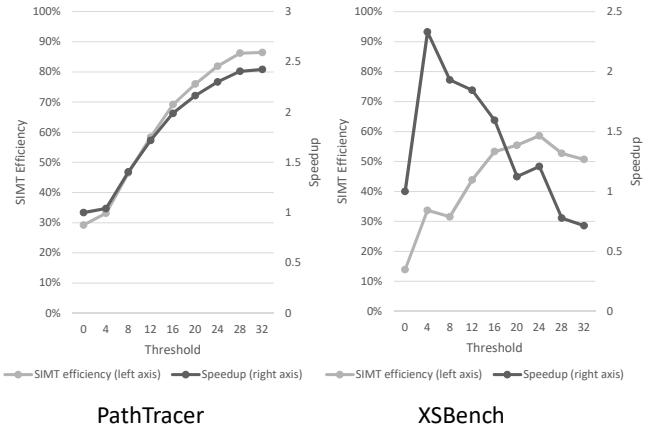


Figure 9. SIMT efficiency and speedup with soft barrier.

leading to unpredictable loop iterations, between four and 321 iterations per thread. PathTracer simulates Monte Carlo light transport using Russian Roulette to randomly terminate paths, with each sample running one or more bounces up to some maximum limit. In these cases, applying Loop Merge enables better SIMT efficiency of the inner loop, and the performance gains due to higher SIMT efficiency outweigh the cost of added serialized execution to refill empty threads.

In most cases, it is reasonable to expect that SIMT efficiency improvement serves roughly as an upper bound on speedup. Applications with smaller improvements in SIMT efficiency and lower speedups can have lower variance in iteration counts and/or higher serialization cost in the prolog and epilog sections of the code, limiting improvement.

5.3 Results: Soft Barrier

Our approach above aims to improve performance by maximizing SIMT efficiency. However, the best performance is not necessarily always achieved at maximal convergence. To balance the tradeoff between lower SIMT efficiency when threads are idle and the cost of refilling idle threads with new work, we implemented soft barrier (Section 4.6), an extension to our Speculative Reconvergence optimization which allows for partial reconvergence before execution of common code. Our implementation of the soft barrier employs a user-defined threshold and waits for *enough* threads to arrive at the reconvergence point before proceeding rather than waiting for all threads. Equivalently, it allows the program to continue execution until the number of active threads drops below some threshold and refilling idle threads becomes worth the cost.

Figure 9 shows the results of using soft barriers with varying threshold values for two applications: PathTracer and XSBench. PathTracer executes fastest when all threads reconverge before executing; the cost of filling an idle thread with new work is low enough, relative to the main computation, that it is best to immediately refill any idle thread. On the other hand, XSBench reaches peak performance when

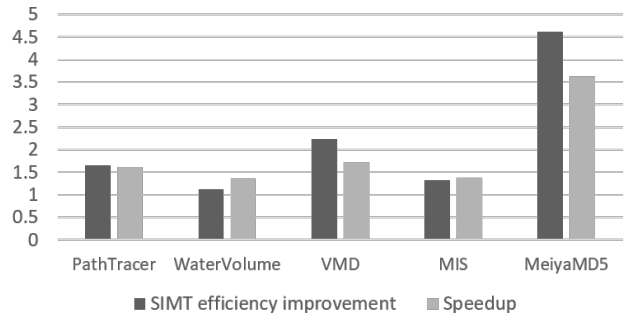


Figure 10. Automatic Speculative Reconvergence.

waiting much longer to refill empty threads. An expensive process is required when a thread wants a new task, and executing this process every time one or a few threads become idle is not profitable. For the particular configuration shown, performance is best when executing the inner loop until as few as four threads are participating. We leave the problem of automatically discovering the ideal threshold parameter for a particular problem to future work.

5.4 Results: Automatic Speculative Reconvergence

While the primary focus of this work is enabling user-directed reconvergence, we also experimented with automatic selection of alternate reconvergence points via compiler heuristics (Section 4.5). We examined a large database for opportunities to apply both Loop Merge and Iteration Delay synchronization patterns during backend compilation from PTX to machine code. As also observed in prior work [24], divergent workloads form a small fraction of GPU applications. Of the 520 CUDA applications we studied, 75 had a SIMT efficiency of less than about 80%. Our implementation detected non-trivial opportunity in 16 applications, and 5 showed significant improvement in SIMT efficiency and runtime.

Figure 10 shows upside potential for a set of applications which were automatically discovered to have opportunity for Speculative Reconvergence. Of the traces detected to have opportunity, several candidates for Loop Merge and Iteration Delay make use of the OptiX raytracing engine [23], an application space known for divergence. Another application, MeiyaMD5, contains a load-imbalanced, compute-heavy inner loop making it the ideal candidate for Loop Merge. For brevity, we restrict results here to cases with significant upside potential. However, many examples with compiler-detected opportunity see no change or even regression, limiting automatic application of this technique without better heuristics, profile guidance, or user input. Finally, automatic Speculative Reconvergence performs the same as programmer-annotated variants of the benchmarks from Table 2.

6 Discussion

This section discusses limitations of our optimization, interactions with other compiler passes, as well as future work.

Interaction with loop optimizations. If a loop is completely unrolled, Iteration Delay and Loop Merge cannot be applied. On the other hand, if the inner loop of a loop nest is partially unrolled by a factor of N, Loop Merge may be still applied. Reconvergence is needed only once per N iterations of the inner loop body, which may reduce the overhead of synchronization for reconvergence. Loop interchange may make the inner loop divergent, thereby introducing opportunity for Loop Merge. Additionally, loop fusion inside an outer loop could help improve performance of Loop Merge and reduce divergence. Loop fission may help break the outer loop into two separate loops, one with the expensive inner loop and the other with the expensive prologue/epilogue section, which could help reduce the effect of increased divergence at the prologue/epilogue regions of the outer loop. These optimizations may also inhibit Loop Merge in some cases.

Interaction with function inlining and code refactoring. If a function call that is common across divergent paths is inlined, we can no longer reconverge threads at a common PC, which inhibits the applicability of our optimization. On the other hand, common code across divergent paths may be refactored into a single method. This operation is the inverse of the function inlining optimization above and introduces opportunity for reconvergence.

Interaction with warp synchronous operations. Our technique changes the convergence properties of a program which may change behavior of warp synchronous operations. However, CUDA 9.0 introduced the requirement that instructions that require inter-thread communication cannot implicitly assume convergence and must use *warpsync* for correctness, which would inhibit automatic Speculative Reconvergence.

Interaction with scalar datapaths. Techniques such as Loop Merge change the scalar nature of loop indices across threads. For devices with scalar pipelines, the ability to exploit scalar data to reduce register count may be reduced if previously scalar data must be stored in thread-private registers. Additionally, loops with deterministic iteration count could previously be considered non-divergent. However, if we modify convergence synchronization, previously convergent regions of loops may become divergent, again limiting the use of scalar datapaths.

Multiple concurrent predictions. Speculative Reconvergence works at all levels of nesting: within a loop, within a function call, and within nested conditional statements. Our method can also support multiple concurrent predictions within a region. If these predictions are exclusive, they can be supported using deconfliction. If they are not exclusive, soft barriers may be utilized to allow for limited convergence

at each point. A detailed study of concurrent overlapping predictions is left to future work.

7 Related Work

Prior research has considered multiple techniques for increasing SIMT efficiency, including Loop Merge, Iteration Delay, and optimistic reconvergence for irregular control flow. In contrast, our work provides a single overarching infrastructure for efficiently implementing all of these optimization patterns.

Loop Merge. Von Hanxleden and Kennedy introduce *loop flattening*, which is conceptually similar to what we describe as the Loop Merge pattern [28]. Their work, while restricted to Loop Merge, provided early proof that executing SIMD threads across iterations could improve efficiency. Similarly, Han and Abdelrahman propose Loop Merge for GPUs [12], which generically merges arbitrary nested loops via a loop transformation. They show that their optimization, which developers can direct via a special pragma, can significantly improve SIMT utilization and performance. The transformation can lead to code bloat because it duplicates the epilogue and prologue of the inner loop.

Iteration Delay. Han and Abdelrahman also present Iteration Delay in separate work [11]. As with Loop Merge, their compiler applies a transformation that introduces a vote to determine which path of an *if-then-else* to execute on each iteration, which is exactly what performance-conscious programmers do for such code patterns [8], as in the case of the Optix ray tracing engine [23]. The Iteration Delay examples above inspired our soft barrier approach.

Irregular Control Flow. Diamos et al. propose the concept of a thread frontier, and show the benefits of optimistically reconverging execution before immediate post-dominator blocks [7]. Their work, which is primarily concerned with irregular control flow, relies on a mechanism for the compiler to dictate the order in which threads traverse basic blocks during execution. By programmatically allowing the insertion and deletion of threads from barriers, our work allows the architecture to use independent traversal mechanisms, and also necessitates an alternative compiler algorithm to manage barriers. Fung and Aamodt describe a mechanism for defining a likely convergence point to allow divergent threads to reconverge earlier than the immediate post-dominator [9]. Their approach precludes the possibility of Iteration Delay and Loop Merge because they do not consider eventually convergent code regions.

Our approach provides a framework to optimize SIMD utilization for a broad range of control flow patterns including irregular control flow, common function calls, Loop Merge, and Iteration Delay. In addition we performed our evaluation using real silicon (a Volta GPU [6]) in the context of a production-quality compiler.

Soft Barrier. Prior work on soft barriers includes [11], which uses a majority vote to decide which path should be taken in Iteration Delay. Our work extends this concept to nested loops. In addition, [1] discusses improving SIMD efficiency in GPU ray tracing by waiting on termination of at least m threads before fetching new data for replacement rays. This approach is similar to our concept of waiting on N threads to arrive at a PC before executing code convergently.

Hardware reconvergence techniques. Prior work on improving convergence at the hardware level involves warp resizing and reformation. Dynamic Warp Formation [10] and Thread Block Compaction [9] involve regrouping threads into warps that are known to execute convergently at runtime. Variable Warp Sizing [24] and Dynamic Warp Subdivision [19] techniques rely on smaller sized warps, effectively splitting diverged threads into separate warps that may be scheduled in parallel. Keckler et al. [15] explore *temporal SIMT*, where convergent threads execute in SIMD fashion but diverged threads may continue executing in parallel in MIMD fashion.

8 Conclusions

Standard post-dominator reconvergence techniques fail to maximize convergent execution for GPU programs with frequently executed divergent regions within imbalanced loops and conditionals. Our investigation finds significant opportunity to improve convergence using non-standard, user-guided reconvergence points, in particular, threads that execute the same code serially today but can be made convergent. We propose a programming model extension that allows a user to identify divergent regions and specify alternate reconvergence points. We have developed compiler techniques that use dataflow analysis to systematically modify synchronization to improve SIMT efficiency, resulting in speedups of up to 200%. Our experiments demonstrate the need for a user-guided approach because early reconvergence is not guaranteed to improve performance, and may in fact hurt SIMT efficiency based on the relative cost of divergence in different code regions. We describe soft barriers which allow for early reconvergence of a subset of threads in a warp to improve SIMT efficiency in multiple code regions. We also discuss an automated approach to the selection of a reconvergence point based on heuristics to calculate the cost of divergence. To our knowledge, no prior work provides such a generalized, accessible software approach for addressing these divergence problems. While our Speculative Reconvergence technique was initially motivated by performance problems due to loop iteration divergence in important Monte Carlo applications, we expect that the optimization will apply to new domains that we have not yet considered. Finally, our work can be extended using improved automated reconvergence, interprocedural analysis, loop transforms, and support for multiple predictions.

References

[1] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics (HPG)*.

[2] Erik Alerstam, William Chun Yip Lo, Tianyi David Han, Jonathan Rose, Stefan Andersson-Engels, and Lothar Lilge. 2010. Next-generation Acceleration and Code Optimization for Light Transport in Turbid Media Using GPUs. *Biomedical Optics Express* 1, 2 (2010).

[3] Andreu Badal and Aldo Badano. 2009. Monte Carlo Simulation of X-ray Imaging Using a Graphics Processing Unit. In *IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*.

[4] Phelim P. Boyle. 1977. Options: A Monte Carlo Approach. *Journal of Financial Economics* 4, 3 (1977).

[5] Judith F. Briesmeister. 2000. MCNP: A General Monte Carlo N-Particle Transport Code. (2000).

[6] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018).

[7] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD Re-convergence at Thread Frontiers. In *International Symposium on Microarchitecture (MICRO)*.

[8] Steffen Frey, Guido Reina, and Thomas Ertl. 2012. SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*.

[9] Wilson W.L. Fung and Tor M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *International Symposium on High Performance Computer Architecture (HPCA)*.

[10] Wilson W.L. Fung, Ivan Sham, George Yuan, and T. M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *International Symposium on Microarchitecture (MICRO)*.

[11] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing Branch divergence in GPU programs. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*.

[12] Tianyi David Han and Tarek S. Abdelrahman. 2013. Reducing Divergence in GPGPU Programs with Loop Merging. In *Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*.

[13] Zheming Jin and Hal Finkel. 2018. Nuclear Reactor Simulation on OpenCL FPGA: A Case Study of RSBench. In *International Workshop on OpenCL*.

[14] James T. Kajiya. 1986. The Rendering Equation. *SIGGRAPH Comput. Graph.* (1986).

[15] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011).

[16] Lawrence Livermore National Labs. 2011. Monte Carlo Benchmark (MCB). https://codesign.llnl.gov/mcb.php

[17] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008).

[18] C-M. Ma, J.S. Li, T. Pawlicki, S.B. Jiang, J. Deng, M.C. Lee, T. Koumrian, M. Luxton, and S. Brain. 2002. A Monte Carlo Dose Calculation Tool for Radiotherapy Treatment Planning. *Physics in Medicine and Biology* 47, 10 (2002).

[19] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *International Symposium on Computer Architecture (ISCA)*.

[20] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[21] NVIDIA. 2018. CUDA Binary Utilities Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#volta

[22] NVIDIA. 2019. Profiler User's Guide, CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[23] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH*.

[24] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A Variable Warp Size Architecture. In *International Symposium on Computer Architecture (ISCA)*.

[25] Michael C. Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. 2007. High-throughput Sequence Alignment Using Graphics Processing Units. *BMC Bioinformatics* 8 (2007).

[26] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *EASC 2014 - Solving Software Challenges for Exascale*.

[27] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Shulz. 2014. XSBench – the Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *The Role of Reactor Physics Toward a Sustainable Future (PHYSOR)*.

[28] Reinhard von Hanxleden and Ken Kennedy. 1992. Relaxing SIMD Control Flow Constraints Using Loop Transformations. In *Conference on Programming Language Design and Implementation (PLDI)*.

[29] Hongwei Wu, Xiangnan Liu, and Weibin Tang. 2011. A Fast GPU-Based Implementation for MD5 Hash Reverse. In *International Conference on Anti-Counterfeiting, Security, and Identification*.