GPU Subwarp Interleaving

Sana Damani[§]
Georgia Institute of Technology
Atlanta, GA, USA
sdamani@gatech.edu

Mark Stephenson
NVIDIA
Austin, TX, USA
mstephenson@nvidia.com

Ram Rangan
NVIDIA
Bangalore, India
rrangan@nvidia.com

Daniel Johnson
NVIDIA
Austin, TX, USA
djohnson@nvidia.com

Rishkul Kulkarni
NVIDIA
Austin, TX, USA
rishkulk@nvidia.com

Stephen W. Keckler
NVIDIA
Austin, TX, USA
skeckler@nvidia.com

***Abstract*—Raytracing applications have naturally high thread divergence, low warp occupancy and are limited by memory latency. In this paper, we present an architectural enhancement called Subwarp Interleaving that exploits thread divergence to hide pipeline stalls in divergent sections of low warp occupancy workloads. Subwarp Interleaving allows for fine-grained interleaved execution of diverged paths within a warp with the goal of increasing hardware utilization and reducing warp latency. However, notwithstanding the promise shown by early microbenchmark studies and an average performance upside of 6.3% (up to 20%) on a simulator across a suite of raytracing application traces, the Subwarp Interleaving design feature has shortcomings that preclude its near-term implementation. This paper introduces the reader to the challenges of raytracing and discusses a novel micro-architectural approach that, on paper, addresses many of the challenges. A thorough analysis of the idea on a production simulator reveals that the high-level motivating statistics are optimistic, and second-order effects, along with other architectural sharp edges, limit the idea's potential. We identify Subwarp Interleaving's primary limiters for an NVIDIA Turing-like architecture, and we outline the conditions under which the approach could be more effective.**

***Keywords*-GPUs, latency tolerance, warp divergence**

I. Introduction

GPUs have successfully accelerated computer graphics applications and other suitable workloads by adhering to two primary design principles. First, GPUs group threads into units, which we call *warps*, that fetch from a single program counter (PC) and execute in SIMT (single instruction, multiple thread) fashion. Second, GPUs hide stalls by concurrently scheduling among many active warps [19]. In this paper, we show that raytracing, which trends suggest is the future of real-time computer graphics, suffers because of the following corollaries to those design principles:

1) GPUs lose efficiency when threads in a warp *diverge* and therefore do not all share the same PC. GPUs serialize execution across divergent threads because SIMT execution requires that all participating threads share a PC.
2) GPUs lose efficiency when the scheduler does not have enough active warps to hide stalls.

Each thread of a traditional raytracing *megakernel* traces the path of a ray through a scene, invoking different shader programs according to whether the ray simulated in a given thread hits or misses objects in the scene [29].[1] Such raytracing kernels are latency-sensitive, divergent, and contain relatively few active warps; these inefficiencies combine to hinder performance on current GPUs. Researchers have previously introduced software solutions to mitigate the problem of divergence [38], [17]. However, in this work, we describe an architectural approach that improves latency tolerance by *leveraging* the divergence inherent in raytracing to effectively increase the scheduler's ability to hide stalls.

Figure 1 illustrates warp divergence in the control flow graph of a raytracing megakernel, where the divergent blocks labeled *Shader A* and *Shader B* contain pedagogical single-block *shader programs*. We will show later how raytracing applications produce control flow of this nature, but for now notice that after the point labeled ①, a warp of threads splits into two *subwarps*, which we define as PC-aligned subsets of a warp's threads. After this point, one subwarp, `S0`, runs "Shader `A`" and the other, `S1`, runs "Shader `B`".

Modern GPUs serialize the execution of these shader programs within a warp. A scheduling policy chooses one subwarp to execute first, and when that subwarp runs to a statically designated point of convergence, only then will the GPU execute the other subwarp. For example, if subwarp `S0` runs first, then it must run to completion before `S1` can begin execution. This approach is inefficient for workloads

[§]Sana Damani performed this work as an intern at NVIDIA, where she was co-mentored by Ram Rangan and Mark Stephenson.

[1]A megakernel's threads can potentially invoke several shader programs for a given kernel invocation, which differentiates megakernels from regular single-program compute kernels.

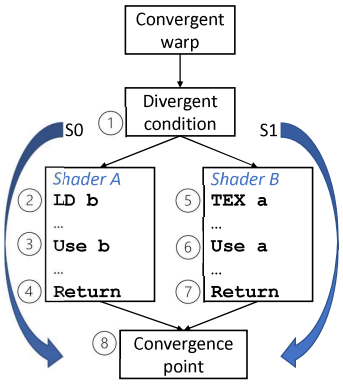
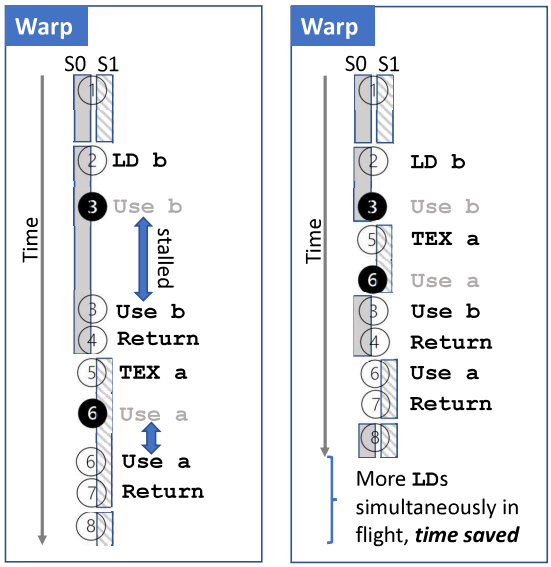


Figure 1: Divergent shader execution. Subwarps `S0` and `S1` execute example single-block shaders `A` and `B` respectively in a divergent manner.

in which there are insufficient active warps to effectively hide the workload's memory requests. We propose *Subwarp Interleaving* (SI) to allow the GPU's scheduler to interleave subwarps instead of serializing them, enabling long-latency operations from divergent code paths to overlap in time and reducing average exposed memory latency.

Figure 2 compares today's subwarp serialization (labeled "Baseline SIMT") to Subwarp Interleaving for the example in Figure 1, using only two subwarps for clarity. In the baseline case (Figure 2a), the load-to-use stalls due to memory accesses in diverged subwarps `S0` and `S1` cannot be overlapped because of the serialized execution of divergent threads. In Figure 2b, Subwarp Interleaving switches between the subwarps in a warp when the program would otherwise stall. With Subwarp Interleaving, instead of stalling at point ❸, a *subwarp scheduler* recognizes that `S1` is ready and makes `S1` the active subwarp. Subwarp `S1` executes instructions until it stalls at ❻ waiting for the result of the texture operation, at which point the subwarp scheduler can swap `S0` back in and successfully hide the long-latency load. Subwarp Interleaving increases the perceived *occupancy* of the GPU, which allows the architecture to better tolerate long-latency operations.

In this paper, we present key lessons from our next-generation GPU architecture exploration work on Subwarp Interleaving. We study its potential on real-time raytraced graphics applications, which represent an important and lucrative application category. To characterize the opportunity for better latency tolerance with Subwarp Interleaving, we define *exposed* long-latency or load-to-use stalls as cycles when *no* active warp in an SM is able to issue, and at least one active warp is stalled on an outstanding memory load operation. The lower this metric, the better the latency tolerance. Figure 3 shows total exposed load-to-use stalls and exposed load-to-use stalls in divergent code blocks, both normalized to respective kernel runtimes, across our suite of raytracing kernels. Despite the latency tolerance and latency



(a) Baseline SIMT. (b) Subwarp Interleaving.

Figure 2: Subwarp Interleaving concept. In GPUs today, the scheduler serializes the execution of subwarps `S0` and `S1`, and is unable to hide load-to-use stalls in each shader. The *subwarp scheduler* interleaves the execution of `S0` and `S1` to hide load-to-use stalls, thereby reducing overall execution time of the program.

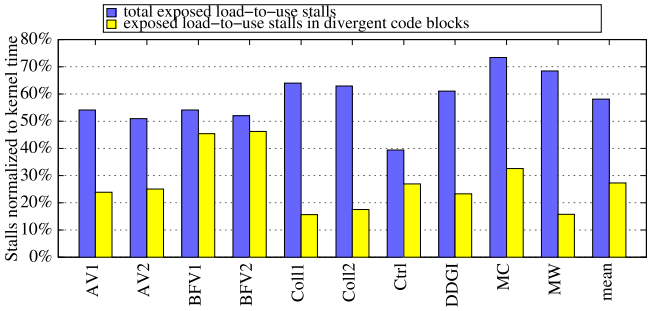


Figure 3: Characteristics favoring Subwarp Interleaving. SI targets applications with higher divergent load-to-use stalls.

reduction improvements in modern NVIDIA GPUs [31], we see that these applications are often stalled waiting for memory, and a significant percentage of those stalls are in divergent code regions. These characteristics favor Subwarp Interleaving. However, even though Subwarp Interleaving is theoretically sound and successfully reduces exposed load-to-use stalls in raytracing applications (sometimes dramatically) through effective latency tolerance, multiple practical considerations, including limited performance upside for additional area and design complexity, along with viable near-term algorithmic workarounds, make commercialization of this feature less attractive.

The contributions of this paper are:

- We propose Subwarp Interleaving to achieve effective latency tolerance in warp-starved scenarios, along with microarchitectural building blocks to support it.
- We evaluate SI using our cycle-accurate proprietary simulator and show that it successfully reduces exposed long-latency stalls, achieves near-linear speedups on micro-benchmarks, and achieves an average speedup of 6.3%, and up to 20%, over a baseline Turing-like configuration across our suite of raytracing application traces.
- Lastly, we describe practical considerations that challenge the adoption of SI and outline the conditions necessary for such a design feature to be attractive for future GPUs.

While we focus on raytracing in this paper, the concepts we present generalize to applications with similar characteristics. Namely, any divergent GPU program, including GPU computing programs, with long stalls and low occupancy might also benefit from our work. The next section describes the baseline architecture used in this study and shows how raytracing lends itself to Subwarp Interleaving.

II. Background

Video games are a staple in today's culture, and though GPUs are used for a wide variety of applications, they are still designed with graphics in mind. Gaming industry revenue is measured in hundreds of billions of dollars [1]; gaming platforms are ubiquitous, and gamers play their favorite titles on mobile devices, consoles, PCs, and in the cloud; universities now offer scholarships for Esports [14]; and millions of viewers watch the best Esports athletes compete [36].

Raytracing is the gold standard for rendering realistic images, but until recently has been too computationally demanding to meet the real-time constraints of gaming. This section provides background information on both real-time raytracing as well as the GPU architectures designed to accelerate this burgeoning workload. Our goal in this section is to illustrate the challenges we face when mapping raytracing workloads onto GPUs, and not to elaborate on the rich field of raytraced computer graphics.

A. GPU Baseline

GPU programming models allow the creation of thousands of threads that each execute the same code. NVIDIA's programming model groups threads into 32-element vectors called warps to improve efficiency. The threads in each warp execute in a SIMT (single instruction, multiple thread) fashion, all fetching from a single Program Counter (PC) in the absence of control flow. In most programs, many warps map to a single GPU core, or streaming multiprocessor (SM) in NVIDIA's terminology. A GPU consists of multiple such SM building blocks along with a memory hierarchy including SM-local scratchpad memories and L1 caches, a shared L2 cache, and multiple memory controllers.

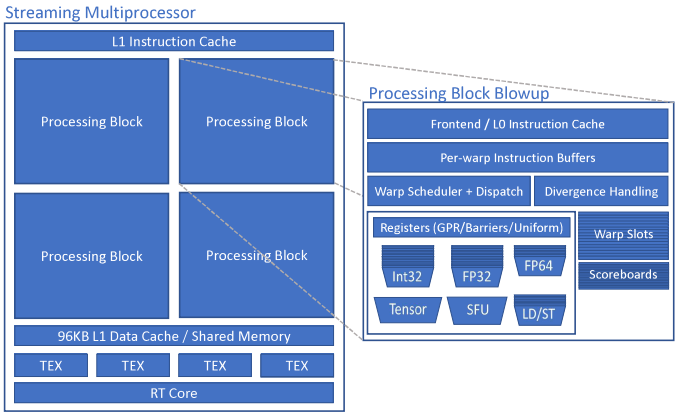


Figure 4: Turing-like SM architecture.

Figure 4 shows a diagram of an NVIDIA Turing-like SM architecture. The SM contains four *processing blocks*, each of which contains a front-end that feeds the processing block with instructions; an L0 instruction cache; 8 "warp slots" that hold warp-private scheduling data (*e.g.*, program counters, scoreboard state, etc.) for the warps that map to the processing block; a datapath for executing SIMT instructions; and finally, a warp scheduler for selecting one of the processing block's warps to schedule onto the datapath [26].

Branches and function calls can introduce *divergence* in which threads within a warp may take different control flow paths; in such cases, the SIMT datapath is underutilized because SIMT execution serializes control flow. For example, if all 32 threads in a warp execute a branch, but only four of those threads take the branch, the warp will splinter into two *subwarps*. One subwarp contains the four threads that take the branch, and the other subwarp contains the remaining 28 threads. Strict SIMT execution forces these subwarps to run serially: one subwarp will run until it reaches a compiler-defined convergence point that is common to both subwarps and wait, and then the other subwarp will run to the convergence point where they will rejoin. Divergence therefore increases the number of SIMT instructions a shader program must execute. Our Subwarp Interleaving approach builds on *independent thread scheduling*, a new divergence-handling paradigm introduced in Volta that enables flexible, fine-grain synchronization among a warp's threads [2], [25], [26].

A primary tenet of GPU execution is latency tolerance: each processing block of a Turing SM can switch between its 8 warp slots to hide long-latency instructions. A program can use at most 8 warp slots per processing block, but many other factors determine a shader program's *occupancy*, which is a measure of how full the SM's warp slots are. For example, as the number of registers used per thread

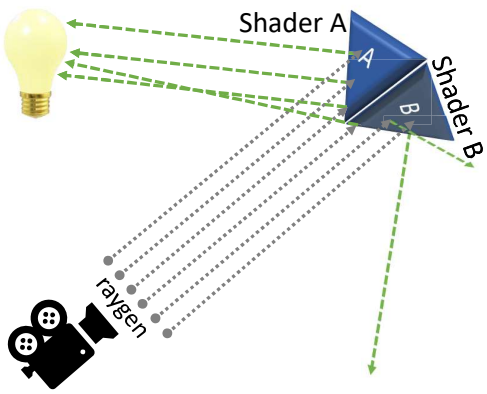


Figure 5: Raytracing example execution. Rays can quickly diverge, which hinders SIMT efficiency.

increases (which is a processing-block-shared resource), the number of warps that fit in the processing block decreases. Likewise, occupancy decreases as shared memory usage and the number of threads per block increases [28].

An SM statically distributes its warps among the warp schedulers of the processing blocks, and on every cycle, each processing block's warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp [28]. However, for memory latency limited workloads with low occupancy, there may not be a ready warp for the scheduler to select and the processing block will stall.

B. Raytracing

To satisfy the real-time constraints required of video gaming, game developers have historically used *rasterization* techniques, which are (relatively) fast and produce visually appealing, though sometimes inaccurate, scenes. Movie studios, without real-time constraints, create special effects using computationally intensive *raytracing* techniques, which recursively follow light paths backwards from the "camera" to the objects in a scene, where eventually some rays hit an emissive light source. Trends suggest raytracing will eventually become the new standard for gaming. First-class shader language support has been added to the DirectX (DXR [23]) and Vulkan APIs [16], exposing the raytracing "pipeline", which GPUs accelerate to various degrees. For example, we have added "RT-cores" to our GPUs at NVIDIA to accelerate ray traversal [26] and gaming consoles such as the Xbox Series X or the Playstation 5, based on AMD's RDNA2.0 architecture, also accelerate ray-traced games [37].

The SM can offload the traversal of a special data structure called the *Bounding Volume Hierarchy* (BVH) to an RT-core, saving thousands of software instructions per ray [26]. An RT-core's BVH traversal tests the intersection of a ray against successively smaller bounding boxes until possibly hitting a triangle, and returning either a "hit" or a "miss" to the SM for further processing. The SM can independently perform other compute or graphics work during a BVH traversal.

Notwithstanding the enormous performance gains that RT-cores provide, raytracing still poses significant optimization challenges. Conceptually all ray paths start at the "camera" and are cast out into the scene, and while the threads for each ray are initially convergent, threads tend to diverge and execute independently. Figure 5 uses a simple scene to illustrate how six of a warp's initially convergent threads diverge over time in the raytracing megakernel. Initially the warps in the megakernel convergently cast rays into the scene by issuing asynchronous *TraceRay* calls to the RT-cores [23]. The arrows in the figure correspond to processing that the RT-cores perform, namely determining whether the given rays intersect with any triangles in the scene. In Figure 5, the RT-cores determine that three rays intersect with the triangle labeled "A", and three rays intersect with "B". The RT-cores return the specifics of each ray's traversal to the issuing SM, at which point the megakernel will invoke *Shader A* or *Shader B*, as appropriate, thereby splintering the convergent threads into two independent subwarps. The shaders can recursively cast rays from a new location, perhaps using the material properties (e.g., matte, glossy) of the associated objects to stochastically scatter rays, as the green arrows in the figure indicate [30]. A subset of rays eventually reach an emissive light source, whose color the shaders may combine at the reflected point(s) and eventually at the camera.

We can refer to the megakernel's structure to explain why resource usage can be problematic for raytracing workloads. Figure 1 shows that the divergent blocks are actually wholly separate shader programs, and the caller at point (1) passes parameters to the programs via a calling convention. Readers who are familiar with the CUDA Application Binary Interface (ABI) may recognize the challenges calling conventions impose on GPU kernels: the kernel that is responsible for calling functions must ensure that it has enough resources to fulfill calls to any of its *possible* targets. For example, if *Shader A* needs 32 registers to execute and *Shader B* requires 128 registers, the entire kernel must use at least 128 registers, even if the kernel never dynamically calls *Shader B*. Figure 1 is simplified in two ways: 1) it shows two-way divergence, but in the worst case a warp can diverge into 32 ways, and 2) it might suggest that shader programs are necessarily simple, but in reality their control flow can be arbitrarily complicated, with loops, conditional statements, and recursive TraceRay calls [11]. Raytracing applications, with multiple potential shader targets, tend to stress the register file and lead to reduced occupancy. In addition, raytracing shaders are typically latency sensitive with relatively few math operations to hide outstanding loads, which further exacerbates the effects of poor SM occupancy.

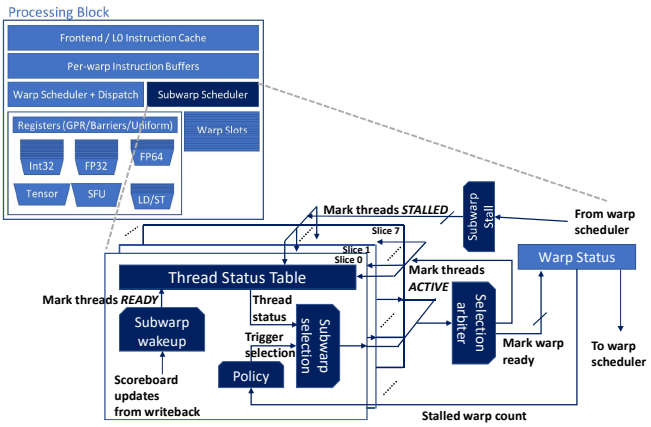


Figure 6: Subwarp Interleaving SM architecture. Our Subwarp Scheduler replaces the divergence handling mechanism and uses subwarp wakeup and selection logic described in III-B to select a new subwarp.

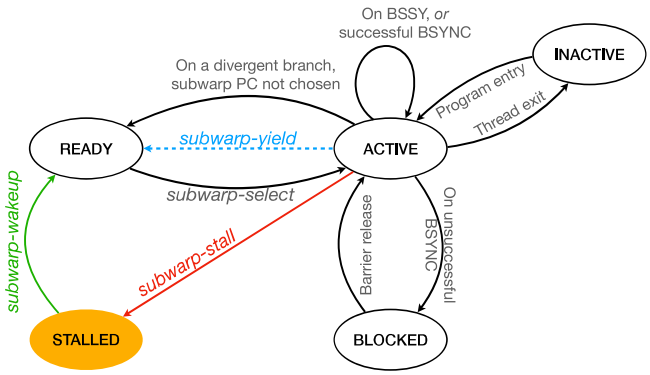


Figure 7: Thread status state machine with Subwarp Interleaving. A divergence handling unit in a Turing-like architecture maintains per-thread state to decide which threads of the warp execute at a given time. Subwarp Interleaving adds the `STALLED` state to demote subwarps that have suffered long-latency stalls. When a stalled subwarp's outstanding dependences complete, the subwarp becomes eligible for scheduling again.

Existing techniques to attack such divergent latency-sensitive codes focus on improving either SIMT convergence or latency tolerance. While we defer a detailed comparison of Subwarp Interleaving with prior work to Section VII, we note briefly that SI differs from prior approaches by opportunistically context-switching to different control paths within a warp (i.e. subwarps) to tolerate load-to-use stalls, without requiring additional warp slots or programmer intervention. We next describe the design enhancements necessary for SI.

III. Subwarp Interleaving Design

Warps are a primitive scheduling unit on modern GPUs, which causes inefficiencies when warps diverge. Our work attempts to increase the perceived occupancy of a GPU by enabling independent scheduling of subsets of a warp's divergent threads, thereby efficiently using the available warp scheduling slots. A *processing block* in an SM can schedule a warp instruction on every cycle, though instruction throughput depends on many factors, including the number of active warps and operand dependency fulfillment. GPU shader cores employ a *warp scheduler* to determine which warp, from among all active warps, gets to use the core's issue slots. At a high-level *Subwarp Interleaving* (SI) treats *subwarps* of warps, defined as a maximal group of threads at a given PC, as the primitive scheduling unit. As a result, SI exploits warp divergence to allow warps that would otherwise stall to gainfully occupy a scheduling slot. An SI architecture demotes stalled subwarps of a warp so that other subwarps of the same warp get a chance to execute.

As Figure 6 shows, we replace the baseline architecture's *divergence handling unit* with a *subwarp scheduler unit*, which contains a *thread status table* and *subwarp wakeup and selection* logic that chooses which of a warp's subwarps should execute when the warp scheduler selects the warp for execution on a core. Importantly, SI does not change the baseline architecture's warp scheduler. The rest of this section describes the components of an SI architecture and how they interact with a baseline Turing-like architecture.

A. Divergence Handling for SIMT

Subwarp Interleaving leverages divergence and must therefore closely interact with the underlying SIMT execution model, examples of which include SIMT stacks [19] and modern convergence barrier models [25], [2]. While SI can be applied to different SIMT execution models, we examine it in the context of NVIDIA's contemporary convergence barrier architectures.

When a warp's SIMT execution diverges, for example because of a conditional branch, the warp will splinter into two or more subwarps and the underlying architecture must track each subwarp's status. The architecture must provide resources to track each thread to handle the degenerate, fully divergent case where each thread operates independently of the others.

Figure 7 describes a possible state machine that tracks the status of a single thread. Baseline state transitions are in black-and-white and SI-specific additions to the state machine are highlighted in color. Every thread starts in the `INACTIVE` state. On program entry, threads transition to the `ACTIVE` state. Thereafter, when a *BSSY Bx* operation [27] executes, all active threads register themselves by setting a per-thread bit in the associated barrier register, *Bx*.

On a divergent branch a *divergence handling unit* serializes the execution of a warp's subwarps. The unit chooses one subwarp for active execution by leaving that subwarp's

threads in the ACTIVE state and transitioning the threads of the other subwarps to the READY state.

On encountering a *BSYNC Bx* operation, a thread can remain in the ACTIVE state (labeled "successful *BSYNC*" in the state transition diagram) if all threads participating in barrier register *Bx* are either in the BLOCKED or INACTIVE states. Otherwise, the thread moves to the BLOCKED state.

When an active subwarp's threads transition to the BLOCKED state, the divergence handling unit examines the status of all threads and selects, via an action we call subwarp-select, a subwarp to move from the READY state to the ACTIVE state. In a Turing-like SIMT model, the divergence handling unit triggers the subwarp-select action based on thread states and per-thread program counters (PCs) stored in dedicated registers.

B. Subwarp Interleaving SIMT Operation

Subwarp Interleaving demotes subwarps that have suffered load-to-use stalls. SI builds on the baseline architecture's thread status state machine by adding a new state, STALLED, and three new types of transitions, namely, subwarp-stall (shown in red), subwarp-wakeup (in green), and subwarp-yield (in dashed blue). While the subwarp-stall and subwarp-wakeup transitions are functionally required for long-latency tolerance, subwarp-yield is optional but may provide extra Subwarp Interleaving flexibility. The blow-up of the subwarp scheduler unit in Figure 6 shows the logical blocks responsible for subwarp-stall, subwarp-wakeup, and subwarp-select.

subwarp-stall: Subwarps that suffer load-to-use stalls on long latency operations cannot make forward progress until their memory lookups return. Rather than occupy a warp's scheduling slots, simple combinational logic in the *subwarp stall* block transitions threads of stalled subwarps to the new STALLED state. If all threads of a warp are STALLED, the warp scheduler will not select the warp for execution. However, by transitioning a subwarp that suffers a stall to STALLED, SI makes other READY subwarps of the warp eligible for scheduling.

subwarp-wakeup: A dedicated block, called *subwarp wakeup*, continually monitors specific scoreboards for successful completion of outstanding long-latency operations from threads of stalled subwarps, and transitions threads upon completion to the READY state. The scoreboards the subwarp wakeup block tracks can be textbook, per-register completion trackers or count-based scoreboards, which are low-complexity dependency trackers that leverage completion order guarantees for memory operations to infer bulk completion of one or more operations based on outstanding counts. We assume the latter in this paper.

subwarp-select: After a subwarp-stall, subwarp-yield (described below), or an unsuccessful *BSYNC*, the hardware must find a new subwarp to move to the ACTIVE state. The *subwarp selection* block consults thread status and per-thread PCs to transition subwarps in the READY state to ACTIVE.

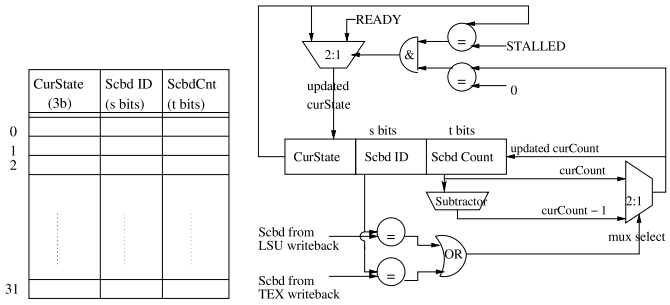
Optionally an SI architecture can support eager transitions from the ACTIVE state to READY. The subwarp-yield transition allows a subwarp to relinquish its scheduling slot to another subwarp of the same warp. The transition can be achieved either through an explicit software instruction, encoded as a scheduling hint in the instruction stream, or via fixed hardware policies. An example of a fixed hardware policy is to yield after issuing a configurable threshold of long-latency operations (such as texture lookups or global memory loads) or operations to core-level shared functional units, such as those executing transcendental operations. While subwarp-stall enables interleaving of subwarps at load-to-use stall points, subwarp-yield creates additional opportunities for interleaving, potentially allowing multiple subwarps of a warp to issue their respective memory loads even before any one subwarp of the warp suffers load-to-use stalls. Subwarps that subwarp-yield move to the READY state, allowing the subwarp selection block to select them if another READY subwarp cannot be found.

C. Subwarp Interleaving Microarchitecture

We now fill in the details of the blocks presented in Figure 6. The *subwarp scheduler* unit tracks the status of a warp's threads, and wakes up threads as and when their data dependencies are satisfied. Figure 8 shows the hardware structures SI uses for both tasks.

Prior to delving into the subwarp scheduler unit, we briefly describe salient aspects of the count-based scoreboarding mechanism used in our design. A count-based scoreboard, typically used with a variable latency operation such as a memory lookup, is *incremented* when an associated operation is issued and *decremented* when the associated operation writes back. A dependent consumer blocks until the corresponding tagged scoreboard counts down to a desired value. If there are ordering guarantees for the producer operation, a non-zero scoreboard count may be specified. If not, the dependent operation will simply wait until the scoreboard counts down to zero. This guarantees that all register writes guarded by this scoreboard have completed and the corresponding registers are safe for consumption. While the baseline architecture has a limited number of scoreboard counters per warp (N_{sb}) and the counter increments and decrements are performed on a warp-wide basis, Subwarp Interleaving replicates the same set of counters on a per thread basis (or more generally on a per subwarp basis) to avoid aliasing scoreboard updates and reads across subwarps. With this background, we present the details of the subwarp scheduler unit.

1) Thread Status Table: At the heart of the subwarp scheduler unit is a per-warp 32-entry *thread status table* (TST), which maintains information needed to perform



(a) Thread status table. (b) Wakeup logic for a single TST entry.

Figure 8: Thread status table (TST) operation. The TST holds thread active and ready states as well as scoreboard counters. On writeback, the wakeup logic updates the corresponding scoreboard for each thread and marks a STALLED thread as READY if all its scoreboard counters are `0`.

both the `subwarp-wakeup` and `subwarp-select` operations. Minimally, the TST must have dedicated storage for each thread's current state (3 bits) and pertinent information on the counted scoreboard on which a subwarp has stalled during dependency resolution, namely, the scoreboard ID (s bits wide) and current count (t bits wide). The quantity s equals $log_2(N_{SB})$, where N_{SB} is the number of counted scoreboard trackers per warp.

2) Subwarp Wakeup: A `subwarp-stall` transition occurs when instruction issue stalls due to a required scoreboard, *sbid*, not being ready. As part of the `subwarp-stall` transition, the TST records *sbid* and its current counter value in the scoreboard ID and scoreboard count fields, respectively, for all applicable threads. Thereafter, every time a subwarp writes back a scoreboard-protected operand to the vector register file, in addition to updating the main set of per-thread scoreboards located in the warp scheduler, those scoreboard IDs get broadcast to the TST as well. Each TST entry has associated wakeup logic that compares the broadcast scoreboard IDs with the recorded ID in the entry. If any of the broadcast IDs match with the recorded entry, the recorded count is decremented by one. Otherwise, the earlier value for the count field is retained. For simplicity, our design does not assume any ordering guarantees for writeback operations. Thus only when a TST entry's scoreboard count field reaches zero and its current state is `STALLED`, does the entry transition to the `READY` state.

Figure 8b shows the wakeup logic for a single entry of the TST. To handle the two writeback paths for long-latency operations in our design, one from the texture units and another from the load/store units, each TST entry has two s-wide comparators and a t-wide subtractor to decrement the scoreboard count on a successful scoreboard ID match. The logic then uses a t-wide comparator to compare the outstanding scoreboard count with 0. If the above comparison is successful and the current state is `STALLED` (a 3-bit comparator), the logic sets the current state to `READY`. For a 32-entry TST, the above logic requires 128 comparators, 64 2:1 muxes, and 32 subtractors. Supporting per-thread wakeup logic is an extreme point in the space that allows the subwarp scheduler unit to support up to 32 independent subwarps, but comes with a sizable cost in terms of extra logic and state that warps must maintain. Total additional storage per warp slot for a TST with N_{TST} entries is $N_{TST} \times (3 + s + t)$ bits. For this extreme design point, with $s = 3$, $t = 6$, $N_{TST} = 32$, and 32 warp slots per SM, additional SM-wide storage for all the TSTs comes to 1.5KB. In Section V we show that we can retain much of the benefits of per-thread tracking by binning threads of a subwarp into a limited number of TST entries. For example, a TST with two entries could efficiently support two subwarps per warp and would add a modest 96 bytes of SM-wide storage over the baseline design. Storage for per-subwarp scoreboards in the warp scheduler would reduce commensurately as well.

3) Subwarp Selection: Recall that when there is no active subwarp, the baseline relies on hardware support to identify an eligible subwarp in the `READY` state and transitions that subwarp of threads to `ACTIVE`. In our SI-enabled architecture, we implement this functionality in dedicated logic called subwarp selection, that operates on information available in the TST. The subwarp selection logic incurs a fixed cost, called the subwarp switch latency, which we model as 6 cycles.

We considered many policies to guide subwarp selection and converged to a heuristic with a single configuration knob that operates on stalled warps that contain at least one `READY` subwarp. The knob controls *when* to trigger `subwarp-select` to make a `READY` subwarp `ACTIVE`. In this paper we consider three points in this knob's space: given N, the fraction of stalled warps among currently live warps in an SM, we trigger `subwarp-select` when, 1) at least one warp (per processing block) is stalled ($N > 0$), 2) when at least half of the warps are stalled ($N \geq 0.5$), and 3) when *all* warps are stalled ($N = 1$). Intuitively, switching stalled warps when there are other active warps ($N > 0$) helps hide the subwarp switch latency under the shadow of the execution of those active warps. On the other hand, switching too aggressively can cause cache thrashing that a more conservative, demand-based policy such as $N = 1$ may avoid.

Our design then triggers `subwarp-select` on only the lowest-numbered stalled warp with a subwarp in the `READY` state. For warps with multiple `READY` subwarps, subwarp selection selects the next `READY` subwarp in a round-robin manner to transition to `ACTIVE`. If no ready subwarp is available, the current subwarp transitions back to `ACTIVE`.

```
1.      BSSY B0, syncPoint
2. @P0 BRA  Else  // P0 is 1 for t0, 0 for t1
3.      TLD R2, R0, R1; &wr=sb5     // incr. scoreboard 5
4.      FMUL R10, R5, c[1][16];
5.      FMUL R2, R2, R10; &req=sb5 // load-to-use stall
6.      BRA syncPoint;
Else:
7.      TEX R1, R8, R9; &wr=sb2     // incr. scoreboard 2
8.      FADD R1, R1, R3; &req=sb2  // load-to-use stall
9.      BRA syncPoint;
syncPoint:
10.     BSYNC B0;
```

Figure 9: A simple code example that shows a divergent if-then-else branch with load-to-use stalls along both paths.

D. Operation

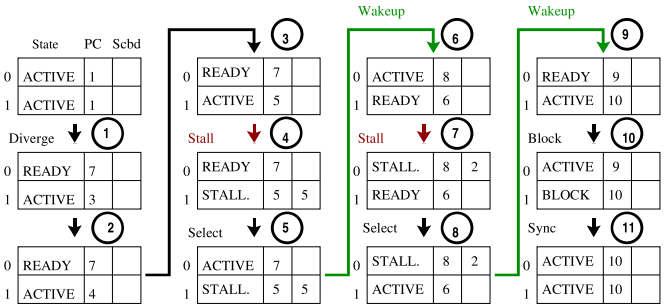
The Subwarp Interleaving components work together as follows to better tolerate long latency stalls. The currently active subwarp transitions to `STALLED` upon suffering a long-latency stall or moves to `BLOCKED` upon hitting a *BSYNC* operation.

The subwarp selection heuristic then selects and transitions a ready subwarp, if available, to the `ACTIVE` state. The newly-active subwarp might issue long latency operations from its code region, which could cause the subwarp to transition to the `STALLED` state. Even if the code region does not stall, the subwarp will eventually transition to the `BLOCKED` state.

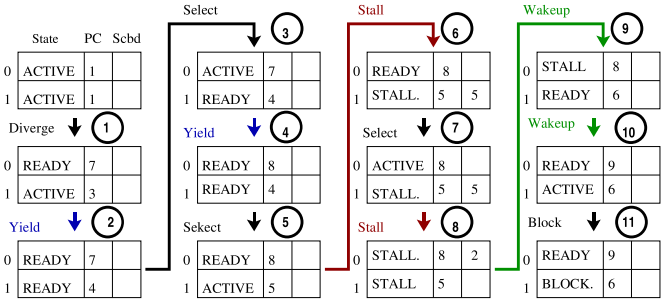
At the point where all subwarps are in the `STALLED` or `BLOCKED` states, memory lookups from all applicable subwarps of a warp will successfully overlap in time, thus achieving improved latency tolerance compared to a baseline execution that serializes execution of divergent subwarps. Eventually, as memory lookups return to the SM, `STALLED` subwarps move to the `READY` state, thus making them eligible for selection again.

On a system that supports the `subwarp-yield` transition, after issuing a group of independent long latency texture lookups or global memory loads, the issuing subwarp may optionally attempt to yield its scheduling slot (through the use of explicit software instructions or through hardware based programmable thresholds) and eagerly move from `ACTIVE` to the `READY` state. Theoretically, `subwarp-yield` provides more flexibility to Subwarp Interleaving and enables maximal memory latency overlap. However, eager approaches like `subwarp-yield` lead to more switching on average in a subwarp's lifetime. There are two reasons why frequently switching between subwarps can hurt performance: First, subwarp selection incurs a fixed six cycle overhead, and second, ping-ponging between subwarps can thrash a processing block's L0 instruction cache.

Figure 10a illustrates the operation of the TST and the state machine described earlier on the toy example from Figure 9. We trace the execution of the toy example in steps. Each step may take one or more cycles. Though we show at most one state change per step (across subwarps) for clarity, in practice, since the `subwarp-wakeup` transition happens



(a) Without `subwarp-yield`.



(b) With `subwarp-yield`.

Figure 10: TST operation with two 1-thread subwarps for the example code in Figure 9. Initially, all threads are active. When threads arrive at a divergent branch, the election logic selects one thread to execute (ACTIVE) and moves the other to a READY state. In (a), when the ACTIVE thread encounters a load-to-use stall at the use instruction, it moves to a STALLED state and the scheduler activates the READY thread. In (b), an ACTIVE thread yields execution to the READY thread as soon as it issues a long latency instruction and does not wait for a stall in the pipeline.

asynchronously as a result of register writeback, `STALLED`-to-`READY` state changes can happen concurrently with other state changes brought about by the baseline SIMT model itself. In step 1, control flow diverges at the branch at PC 2. Prior to this, not shown in the figure, the *BSSY B0, syncPoint* operation is executed by both threads, t0 and t1, and the convergence barrier mask in *B0* gets set to `0x0b11`. After control flow diverges, thread 0 (t0) goes down the "else" path and its TST entry moves to the `READY` state. At step 2, thread 1 (t1) issues its long latency texture operation at PC 3 and remains in the `ACTIVE` state. At step 3, t1 successfully executes an independent math operation at PC 4. At step 4, t1 suffers a load-to-use stall and moves to the `STALLED` state. Thread 0 (t0) is selected in step 6 and issues a long-latency operation (PC 7) at step 5. It suffers a load-to-use stall in step 7. Meanwhile, in the background, t1's scoreboard stall condition cleared and it is now eligible for selection. Thread t1 becomes `ACTIVE` in step 8, attempts to execute the *BSYNC* at step 9, and moves to the `BLOCKED` state in step 10.

Table I: Architecture simulation parameters.

# Streaming Multiprocessors	2
Processing blocks per SM	4
Warp slots per processing block	$\{2, 4, 8\}$
Warp slots per SM	$\{8, 16, 32\}$
Warp size	32
L1 data cache size	128KB
L1 instruction cache size	$\{64KB\}$
L0 instruction cache size	$\{16KB\}$
L1 miss latency	$\{300, 600, 900\}$ cycles
Subwarp switch latency	6 cycles

In a similar vein, Figure 10b illustrates Subwarp Interleaving on a system supporting the `subwarp-yield` transition. They key difference is that the `subwarp-yield` transition at step 2, after t1 has issued its long-latency texture operation, enables t0 to become `ACTIVE` and issue its long-latency texture operation much earlier in the overall schedule (at step 4) compared to Figure 10a, where t1 is able to issue its long-latency lookup only in step 6. Thus, `subwarp-yield` allows a system to maximize memory level parallelism, which is key to lowering average memory access latencies.

IV. Methodology

A. Simulator

To collect the results in this paper, we extended an execution-driven, proprietary simulator that guides NVIDIA GPU product designs. This bare metal simulator (i.e. not full-stack) initializes data and instruction memory from traces. We configure the simulator for a Turing-like SM in a cycle-accurate fashion, including processing blocks, L1 caches, texture units, and the RT-core raytracing accelerator. Because our target applications are not memory bandwidth limited, which we verified by examining performance counters from silicon runs on an NVIDIA GeForce Titan RTX, we do not model a complete GPU memory system, choosing instead to model memory with a simple fixed-latency stub model. Our stub model allows us to flexibly sweep through a range of memory latency values relatively quickly, often 100× faster than a full GPU simulator with a memory system.

Table I lists the simulation parameters used to collect the results in this paper. We test the sensitivity of SI to simulation parameters, such as the number of warp slots per processing block and SM. As mentioned above, we do not model the memory system beyond the SM, but instead study sensitivity to a variety of memory latencies, ranging from optimistically fast to pessimistically slow. To evaluate Subwarp Interleaving we modified our simulator to execute the policies described in Section III-D.

B. Application Traces

Our work focuses on raytracing applications because they tend to be highly divergent with unpredictable memory behavior. The raytracing algorithms in all our workloads, which are based on the DXR API [23], use Bounded Volume Hierarchy (BVH) data structures as configured by their respective developers. SI leverages RT-core accelerated BVH traversals to see past traversal bottlenecks and optimize execution of shading-heavy RT algorithms with high divergence and memory load-to-use stalls. Shadow raytracing regimes tend to be traversal-heavy and their performance is primarily limited by the RT-cores. Such regimes have minimal shading and are unlikely to benefit from SI. On the other hand, techniques like global illumination and reflection raytracing tend to be shading-heavy and benefit nicely from SI. We draw our suite of application traces from this latter category. All of our traces are based on megakernels as our driver currently prefers a megakernel-based implementation for raytracing calls.

Table II: Real-time graphics applications. GI-D stands for Global Illumination - Diffuse, AO for Ambient Occlusion, R for Reflection, and M for multiple raytracing effects represented in a given trace.

Application	Trace Name	RT effect	Description
ArchViz Interior	AV1	GI-D	Architectural rendering [7]
ArchViz Interior	AV2	AO	Architectural rendering
Battlefield V scene 1	BFV1	R	Game [13]
Battlefield V scene 2	BFV2	R	Game
Control	Ctrl	M	Game
RTX Collage	Coll1	AO	Internal demo
RTX Collage	Coll2	R	Internal demo
DDGI Villa	DDGI	GI-D	Greek Villa demo for [20]
Mechwarrior 5	MW	R	Game
Minecraft	MC	M	Game

Table II provides details of the application traces we evaluate in Section V. The ArchViz Interior and Mechwarrior 5 applications are based on Unreal Engine 4 [6], Control is based on Northlight Engine [32], and Battlefield V on Frostbite 3 [4]. The other applications use custom development frameworks. Since full-frame, trace-based execution is intractable, we restricted ourselves to simulating only key raytracing kernel calls. We collected traces of these kernels from recent builds of the associated DXR applications as well as a recent NVIDIA GeForce driver. Our simulator performs systematic sampling of cooperative thread arrays (CTAs) from these traces, captured at either 1080p or 1440p resolution, to obtain representative coverage of the entire screen space for all our workloads at reasonable simulation times (akin to similar sampling techniques for CPU simulations [40]). While our kernel-level results do not reflect current frame-level opportunity, we predict that raytracing will become the de facto real-time rendering standard. Thus we use the performance of these kernels as a proxy for the future opportunity of SI.

V. Results

Before considering raytracing kernels from graphics applications, we first evaluate SI using a microbenchmark.

```
__global__ void subwarps(int *_data, int *_result)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int warp_tid = tid % WARP_SIZE;
    int subwarpid = warp_tid / SUBWARP_SIZE;
    int subwarp_offset = subwarpid *
         NUM_ACCESSES_PER_SUBWARP;

    for (int it = 0; it < ITERATIONS; it++) {
        switch (subwarpid) {
        case 0:
          _result[tid] = gen_ld_to_use_stalls(_data,
                subwarp_offset, subwarpid);
          break;
        case 1:
          _result[tid] = gen_ld_to_use_stalls(_data,
                subwarp_offset, subwarpid);
          break;
        ...
        case 31:
          _result[tid] = gen_ld_to_use_stalls(_data,
                subwarp_offset, subwarpid);
          break;
        }
        __syncwarp();
        subwarp_offset += L2_CACHE_LINE;
    }
}
```

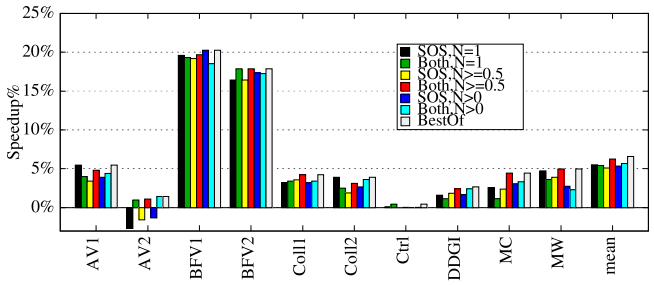
Figure 11: A CUDA microbenchmark that mimics a RT megakernel's structure but allows us to selectively generate up to 32 subwarps with guaranteed exposed load-to-use stalls.

Table III: Subwarp Interleaving performance on the microbenchmark with an L1 miss latency of 600 cycles.

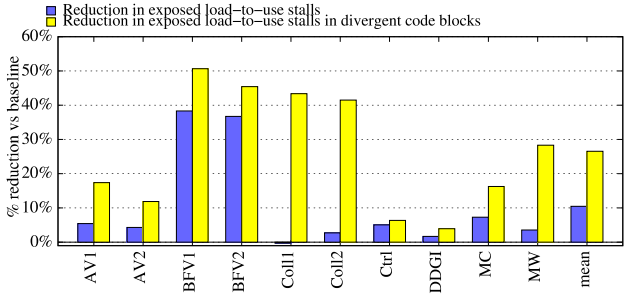
SUBWARP_SIZE	16	8	4	2	1
Divergence factor	2	4	8	16	32
Speedup(×)	1.98	3.95	7.84	15.22	12.66

A. CUDA Microbenchmark

Figure 11 shows the core of a simple CUDA microbenchmark that we designed to mimic a megakernel's structure while allowing us to test the upper bounds of SI's performance. We can configure the benchmark to splinter a warp from two to 32 subwarps (via `SUBWARP_SIZE`). Each subwarp calls the `gen_ld_to_use_stalls` function, which performs a reduction of a slice of `_data`. We ensure that each subwarp will suffer compulsory data cache misses when executed. Table III shows the performance potential of Subwarp Interleaving on the above microbenchmark. We vary the divergence factor as 2, 4, 8, 16, and 32, by varying `SUBWARP_SIZE` as 16, 8, 4, 2, and 1, respectively. SI delivers almost linear speedups until about 16-way divergence before tapering off. Performance plateaus at high divergence factors (above 16) because with increasing active subwarps, the number of active instruction fetch streams increases as well, leading to L0 and L1 instruction cache thrashing. With 32-way divergence, we see load-to-use stalls decrease to 0 with Subwarp Interleaving, but instruction fetch stalls rise sharply, leading to diminishing returns.



(a) Speedup.



(b) Reduction in exposed load-to-use stalls.

Figure 12: Subwarp Interleaving versus baseline at a fixed L1 miss latency of 600 cycles.

B. Raytracing Kernels

We next describe the performance of Subwarp Interleaving on real-time raytracing kernels. Unlike the microbenchmark scaling study, our goal here is to evaluate the effectiveness of SI in exploiting these kernels' natural divergence to effectively tolerate memory stalls.

We study the performance of our technique with an exhaustive sweep over three different parameters: subwarp selection trigger policy, L1 miss latency, and Subwarp Interleaving with and without the `subwarp-yield` transition in our state machine. As mentioned in Section III-C, we trigger subwarp selection based on how many warps are currently stalled in a given processing block. We pick three points on that axis: $N = 1$ to represent all active warps having stalled, $N \geq 0.5$ to represent at least half of the active warps having stalled, and $N > 0$ to represent any active warp having stalled. For L1 miss latency, we sweep across values of 300, 600, and 900 cycles. For the last parameter, we report just the basic Subwarp Interleaving design as *SOS* (for switch-on-stall) and *SOS* plus `subwarp-yield` after long-latency instructions as *Both*.

Figure 12a shows the performance of subwarp interleaving at a fixed L1 miss latency of 600 cycles. In addition to showing the performance of individual settings, the graph shows a *BestOf* bar that captures the best performance for each application across all SI configurations. The single best performing setting is *Both*, $N \geq 0.5$, achieving an average speedup of 6.3%. Average *BestOf* speedup across all settings is 6.6%.

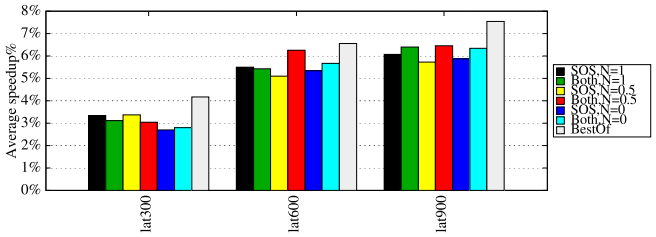


Figure 13: Average speedups of Subwarp Interleaving over baseline across different L1 miss latency settings.

Figure 12b shows the reduction in total exposed load-to-use stalls (total stalls) and exposed load-to-use stalls in divergent code blocks (divergent stalls) from Subwarp Interleaving with respect to the corresponding baseline metrics. Divergent stalls dropped by 26.5% on average (yellow bars). However, more than half of our traces see small reductions for divergent stalls. There are two primary reasons that warps may be divergent yet present few interleaving opportunities. First, SI cannot generate two independent subwarps for "if-then" divergence because the subwarp that executes the *then* code must finish before the two subwarps rejoin and continue executing together. SI can only exploit divergence that leads to multiple, independent subwarps and is thus applicable only to divergent if-then-else statements, switch statements, and divergent function calls. Second, even for branches that generate two or more independent subwarps, which we expect the megakernel's primary branch to do, SI is still at the mercy of the subwarp selection order. Ideally subwarps with loads execute before compute-heavy subwarps that will never trigger a `subwarp-stall` or `subwarp-yield`. Thus, while the divergent stalls metric is *indicative* of opportunity, it provides only a loose approximation. For applications with significant load-to-use stalls where most of the stalls are in divergent code blocks, SI is likely to help (*BFV1*, *BFV2*). For applications where most load-to-use stalls are in *convergent* code, performance is unlikely to improve commensurately with the reduction in divergent stalls (*Coll1*, *Coll2*).

C. Sensitivity Studies

1) L1 miss latency: We summarize the sensitivity to latency by reporting the average performance of the various configurations in Figure 13. As expected, Subwarp Interleaving performs better with increasing L1 miss latencies, demonstrating its ability to effectively tolerate memory latency stalls. The *BestOf* speedups across L1 miss latency settings of 300, 600, and 900 cycles are 4.2%, 6.6%, and 7.6%, respectively.

2) Peak warp slots per SM: As applications increasingly look to exploit task-level parallelism with modern graphics API support for Asynchronous Compute queues ("Async" queue) [5], [35], there will be contention for limited available warp slots since tasks from multiple queues will overlap

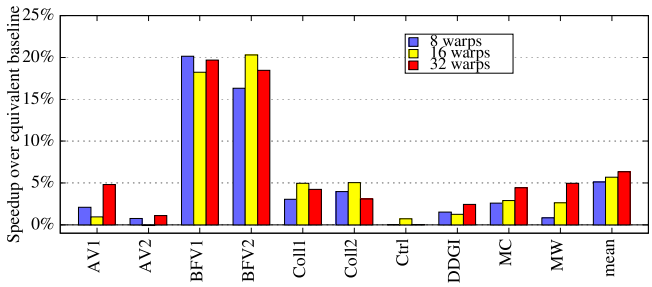


Figure 14: Sensitivity to number of warp slots.

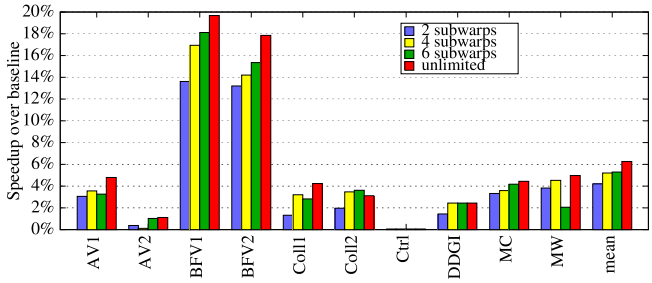


Figure 15: Sensitivity to subwarps count (32 peak warps).

in time. This trend may prevent an approach like Dynamic Warp Subdivision [22], which relies on forking new warps at divergence points, from finding enough free warp slots to achieve maximal latency tolerance. To study the effect of limited warp slots on the performance of SI, we sweep through different maximum warp count settings and compare Subwarp Interleaving with identically warp-throttled baseline configurations in Figure 14. SI sees a small impact from throttling active warps, delivering 5.1%, 5.7%, and 6.3% average speedups when peak warp count is restricted to 8, 16, and 32 warps over *equivalent* baselines. Warp throttling leads to reduced latency tolerance overall, including in non-divergent code regions, which leads to reduced overall efficiency. This reduction in latency tolerance is responsible for SI's reduced impact with warp throttling.

3) Subwarps per warp: We study the sensitivity of Subwarp Interleaving to the number of subwarps per warp supported in the SI design. The sensitivity we present allows an architect to choose an appropriately sized thread status table (TST) to balance performance with power and area concerns. Figure 15 presents speedup data from sweeping subwarp count from 2, 4, 6, to unlimited (max 32). Even with support for as little as 2 subwarps per warp, Subwarp Interleaving is able to achieve an average speedup of 4.2%, with speedups increasing sub-linearly with more subwarps per warp. The 4 subwarps per warp configuration only requires a 4-entry TST and uses one eighth the TST and subwarp wakeup logic of the unlimited configuration. Yet the 4 subwarp configuration achieves a 5.2% speedup, capturing 82% of the unlimited configuration's average upside.

4) Instruction cache sizing: Finally, our baseline configuration upsizes the L0 and L1 instruction caches (16KB and 64KB respectively) to better cater to the needs of Subwarp Interleaving. An experiment with $4\times$ smaller L0 and L1 instruction caches (to mimic shipping GPUs) yielded a 4.5% average speedup, which is about 70% of our single best configuration speedup of 6.3%.

VI. Discussion

Our primary objective with SI was to reduce exposed memory load-to-use stalls. Our evaluation on a cycle-level simulator shows that Subwarp Interleaving does reduce exposed long-latency stalls, sometimes significantly. However, productization of SI is challenged by several factors.

First, the reduction in exposed long-latency stalls due to SI often comes with an increase in instruction fetch stalls due to frequent switching among instruction streams. These stalls can be mitigated with larger caches, albeit at an area cost. In an area-limited chip design, spending additional area on one feature typically involves reducing area for or eliminating another feature. These decisions require broad performance per mm^2 analyses across many candidate features in a product design, with the ones producing the biggest bang for the buck winning out.

Second, Amdahl's Law limits speedups for some of the raytracing kernels. Recall that the invocation of various hit or miss shaders in the raytracing megakernel depends on the outcome of a ray traversal operation, which the RT-core unit performs. While Subwarp Interleaving reduces exposed load-to-use stalls in divergent hit/miss shader execution for all of our raytracing kernels, the latency of ray traversal operations is often the dominant factor.

Third, the order in which a processing block encounters subwarps is important. For example, in a warp with two subwarps A and B, if only subwarp B contains load-to-use stalls, execution order matters. Unless B executes first, SI will be of no value since A will never switch. When A completes and B begins executing, no other subwarp is available to switch to on its memory stalls. Future work could explore the use of software hints to convey load stall probabilities in each divergent path so that hardware can prefer the higher load stall probability path first and use the other path for latency tolerance. Or, in the absence of such hints, the hardware could randomize the execution order of divergent paths, and thus improve the odds of creating a profitable dynamic subwarp scheduling order.

Fourth, while Subwarp Interleaving broadly applies to raytracing kernels, and we argue that raytracing is an extremely important application category, SI currently applies to a limited set of code. SI can only improve application performance when there are long stalls within divergent code, and too few active warps to hide the latency. As skilled SIMT programmers strive to remove divergence from code, we did not expect SI to be universally applicable. We profiled a broad suite of more than 400 non-raytracing CUDA and Direct3D compute kernels and found only 11 that feature long stalls in divergent code, and none benefited beyond the margin of noise from SI.

In addition, current RT game titles are not fully raytraced, but also include general compute and traditional rasterization-based graphics, which dilute SI's gains at the frame level. While Subwarp Interleaving is not readily applicable to today's workloads, we remain optimistic that as GPU applications and technologies evolve, it could be a boon for a future GPU design. As researchers explore novel uses of NVIDIA's raytracing cores to accelerate complex tree data structure traversals [39], [34], [24], [41], we expect that more applications with unpredictable control flow and dependent memory fetches will begin running efficiently on modern GPUs, and a first class hardware feature like Subwarp Interleaving that exploits divergence to achieve better latency tolerance will find broader benefit. Though sophisticated software techniques can help improve convergence rates and memory level parallelism [12], [38], [17], a well-designed hardware technique will avoid the runtime overheads that software techniques impose and will also relieve application developers of the burden of devising strategies to overcome performance problems and allow them to focus on delivering richer functionality.

VII. Related Work

A. Increasing Convergence

Multiple software techniques exist to improve convergence. OptiX is a raytracing library for SIMT architectures that relies on an iterative, warp-level scheduler to reduce divergence [29]. On any given iteration the scheduler chooses the predominant shader (*i.e*, the largest subwarp) and pushes the execution of the other subwarps to future scheduling iterations where they might combine with new work to become part of a predominant subwarp [29]. *Speculative reconvergence* similarly aligns subwarps across loop iterations to increase convergence [3], and *SIMT microscheduling* performs "task fetching" and "task context switching" to ensure that threads in a warp execute convergently and do not remain idle [8]. Hoberock et al. proposed *stream compaction* to sort shaders, triggered by divergent ray intersections, prior to scheduling them in a highly convergent fashion [12]. Laine et al's *wavefront* technique uses global queues to achieve implicit compaction, which then enables convergent material shading with material-specific kernels [17]. Wald's *active thread compaction* technique improves path-tracing performance by compacting active threads globally prior to path-extension [38]. *Dynamic warp formation* [10] and *thread block compaction* [9] are hardware techniques that regroup threads across warps to increase convergence.

While these approaches would likely reduce divergence and serialization penalties in raytracing applications, the cost of performing thread grouping and state migration across

warp contexts is significant and may impose non-trivial costs on interactive applications. In contrast to the above techniques, our Subwarp Interleaving approach is lighter weight. It leverages divergence to interleave subwarps to effectively tolerance long latency stalls, a primary performance limiter in raytracing shaders.

B. Increasing Scheduling Opportunities

Subwarps, not warps, are the basic scheduling unit of our approach. Architectures that support narrow SIMT widths similarly increase scheduling opportunities when control flow diverges [18]. However, architects must carefully balance the ramifications of reducing SIMT width because many important application classes (*e.g.*, deep neural networks and legacy computer graphics) are highly convergent.

Some proposed SIMT and SIMD architectures support flexible warp sizing. *Variable warp sizing* uses a smaller base warp size of four threads and groups together convergent subwarps while allowing diverged subwarps to execute independently [33]. Variable warp sizing reduces the penalty of serialization due to divergence and helps interleave execution of instructions across subwarps, but the technique requires complex hardware to handle warp grouping and it can exacerbate divergence in cases where a subwarp's threads are randomly scattered across lanes (and therefore base warps are still likely to contain threads from more than one subwarp). *Robust SIMD* likewise adjusts SIMD width based on performance feedback [21].

Keckler et al. propose an architecture where each lane is an independent multiple instruction, multiple-data (MIMD), multi-threaded processor. The lanes run most efficiently when all threads are processing the same instruction in SIMT fashion, but diverged threads can still continue to execute in MIMD fashion [15].

Dynamic Warp Subdivision (DWS), which also dynamically subdivides warps, inspired our approach. Meng et al. proposed DWS before real-time raytracing was tractable, and GPUs have changed substantially in that time. Whereas DWS introduced bespoke architectural features [22] to support subwarp-level scheduling, NVIDIA's modern threading strategy provides a natural mechanism for Subwarp Interleaving. Other key differences between our approaches are: a) our approach does not rely on programmer intervention, whereas DWS does; b) our approach interleaves subwarps at stall points, not at branches; and c) our approach allows for unlimited subwarp creation, whereas DWS is limited by availability of unused warp slots.

Graphics APIs and GPUs have recently added support for *asynchronous compute* queues [5], [35], which allows the warp scheduler to overlap work from multiple queues that contain disparate work. Asynchronous compute can improve SM occupancy, but it requires a developer to carefully orchestrate their computation and resource usage. We believe that our approach will perform better than DWS, *especially* when there are few unused warp slots as is likely to be the case with effective asynchronous compute use.

VIII. Conclusion

While real-time raytracing produces more realistic and immersive images than rasterization techniques, the added realism comes with a cost. Raytracing kernels stress GPUs in three fundamental ways: they are highly divergent, they suffer from poor occupancy, and warps routinely stall waiting for long latency operations to complete. Ordinarily, GPU schedulers can hide long latency stalls by switching to other ready warps, but raytracing kernels often have insufficient active warps to hide latency. Subwarp Interleaving is a new technique that aims to reduce pipeline bubbles in raytracing kernels. When a long latency operation stalls a warp and the GPU's warp scheduler cannot find an active warp to switch to, a subwarp scheduler can instead switch execution to another divergent subwarp of the current warp. We present architectural extensions to an NVIDIA Turing-like GPU, which leverages many of the features inherent to the baseline architecture for supporting independent thread scheduling. SI substantially reduces the exposed load-to-use stalls by 10.5%. Our evaluation shows that secondary performance limiters cap the potential of SI in current applications and architectures. While Subwarp Interleaving shows some compelling performance gains (6.3% average, 20% maximum) across a suite of raytracing application traces, its narrow usage and design complexity limit its attractiveness to current GPU architectures. However, the evolution of application demands and their control behavior may motivate future examination of latency tolerance and divergence mitigation approaches such as Subwarp Interleaving.

Acknowledgments

Feedback from the anonymous reviewers greatly improved the quality of this paper. We thank Emmett Kilgariff, Shirish Gadre, and Olivier Giroux for their feedback on draft versions, Sriharsha Niverty for simulation and raytracing workload assistance, Bob Li, Roc Yuan, Colin Liu, and Alex Joly for their help in capturing traces, Ankita Upreti and Samba Wen for simulator help, Micheal Haidl for megakernel support, and John Spitzer for helping curate the final set of applications evaluated in this paper.

References

[1] K. Anderton, "The Business Of Video Games: Market Share For Gaming Platforms in 2019," *Forbes*, June 26, 2019.

[2] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[3] S. Damani, D. R. Johnson, M. Stephenson, S. W. Keckler, E. Yan, M. McKeown, and O. Giroux, "Speculative Reconvergence for Improved SIMT Efficiency," in *International Symposium on Code Generation and Optimization (CGO)*, 2020.

[4] *Frostbite Engine*, DICE. [Online]. Available: https://www.ea.com/frostbite/engine

[5] A. Dunn and S. Hodes, "Asynchronous Compute - Deep Dive," in *Game Developers Conference*, 2017.

[6] *Unreal Engine 4 Documentation*, EPIC Games. [Online]. Available: https://docs.unrealengine.com/4.27/en-US/

[7] *Archviz Interior Rendering*, Epic Games Inc. [Online]. Available: https://www.unrealengine.com/en-US/blog/new-archviz-interior-rendering-sample-project-now-available

[8] S. Frey, G. Reina, and T. Ertl, "SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2012.

[9] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *International Symposium on High Performance Computer Architecture (ISCA)*, 2011.

[10] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[11] E. Haines and T. Akenine-Möller, Eds., *Ray Tracing Gems*. Apress, 2019, http://raytracinggems.com.

[12] J. Hoberock, V. Lu, Y. Jia, and J. C. Hart, "Stream Compaction for Deferred Shading," in *Proceedings of the Conference on High Performance Graphics*, 2009.

[13] C. Holmquist, *Battlefield V: Real-Time Ray Tracing*, DICE, November 2018. [Online]. Available: https://www.ea.com/news/battlefield-5-real-time-ray-tracing

[14] A. Juhasz, "As Esports Take Off, High School Leagues Get In The Game," *National Public Radio*, January 24, 2020.

[15] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, 2011.

[16] *Ray Tracing in Vulkan*, The Khronos Group Inc., 2020, https://www.khronos.org/blog/ray-tracing-in-vulkan.

[17] S. Laine, T. Karras, and T. Aila, "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs," in *Proceedings of the Conference on High Performance Graphics*, 2013.

[18] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators," *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013.

[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, 2008.

[20] Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai, and M. McGuire, "Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields," *Journal of Computer Graphics Techniques (JCGT)*, vol. 8, 2019.

[21] J. Meng, J. W. Sheaffer, and K. Skadron, "Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[22] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[23] *DirectX Raytracing (DXR) Functional Spec*, Microsoft Corporation, 2020, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html.

[24] N. Morrical, W. Usher, I. Wald, and V. Pascucci, "Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing," in *2019 IEEE Visualization Conference (VIS)*, 2019.

[25] *NVIDIA Tesla V100 GPU Architecture*, NVIDIA, Santa Clara, CA, 2017.

[26] *NVIDIA Turing GPU Architecture*, NVIDIA, Santa Clara, CA, 2018.

[27] *CUDA Binary Utilities*, NVIDIA, 2020, https://docs.nvidia.com/cuda/cuda-binary-utilities/.

[28] *CUDA C++ Programming Guide*, NVIDIA, 2020, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[29] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: A General Purpose Ray Tracing Engine," *ACM Transactions on Graphics*, vol. 29, no. 4, 2010.

[30] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory To Implementation*. Morgan Kauffman, 2019.

[31] R. Rangan, N. Turakhia, and A. Joly, "Countering Load-to-Use Stalls in the NVIDIA Turing GPU," *IEEE Micro*, vol. 40, no. 6, 2020.

[32] *Northlight Storytelling Engine*, Remedy Entertainment Plc. [Online]. Available: https://www.remedygames.com/northlight/

[33] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A Variable Warp Size Architecture," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[34] J. Salmon and S. McIntosh-Smith, "Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019.

[35] M. Satran, S. White, and J. Natalie, *Multi-Engine Synchronization*, Microsoft Corporation, 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d12/user-mode-heap-synchronization

[36] N. Smith, "While the Sports World Falls Silent, Esports and Streamers Fill the Void," *Washington Post*, March 17, 2020.

[37] *AMD RDNA2.0 GPUs*, TechPowerUp. [Online]. Available: https://www.techpowerup.com/gpu-specs/?architecture=RDNA+2.0&sort=generation

[38] I. Wald, "Active Thread Compaction for GPU Path Tracing," in *Proceedings of the Conference on High Performance Graphics*, 2011.

[39] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location," in *High-Performance Graphics - Short Papers*, M. Steinberger and T. Foley, Eds. The Eurographics Association, 2019.

[40] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the International Symposium on Computer Architecture*, 2003.

[41] S. Zellmann, M. Weier, and I. Wald, "Accelerating Force-Directed Graph Drawing with RT Cores," in *IEEE Visualization Conference (VIS)*, 2020.