A Variable Warp Size Architecture

Timothy G. Rogers[†] Daniel R. Johnson[‡] Mike O'Connor[‡*] Stephen W. Keckler[‡*]
[†]University of British Columbia [‡]NVIDIA [*]The University of Texas at Austin
tgrogers@ece.ubc.ca {djohnson,moconnor,skeckler}@nvidia.com

Abstract

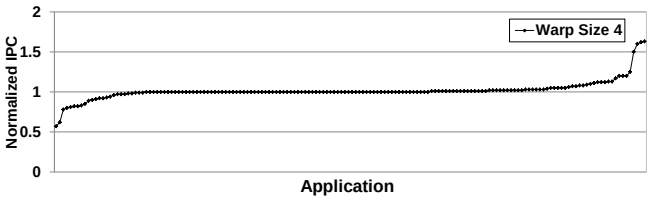
This paper studies the effect of warp sizing and scheduling on performance and efficiency in GPUs. We propose Variable Warp Sizing (VWS) which improves the performance of divergent applications by using a small base warp size in the presence of control flow and memory divergence. When appropriate, our proposed technique groups sets of these smaller warps together by ganging their execution in the warp scheduler, improving performance and energy efficiency for regular applications. Warp ganging is necessary to prevent performance degradation on regular workloads due to memory convergence slip, which results from the inability of smaller warps to exploit the same intra-warp memory locality as larger warps. This paper explores the effect of warp sizing on control flow divergence, memory divergence, and locality. For an estimated 5% area cost, our ganged scheduling microarchitecture results in a simulated 35% performance improvement on divergent workloads by allowing smaller groups of threads to proceed independently, and eliminates the performance degradation due to memory convergence slip that is observed when convergent applications are executed with smaller warp sizes.

1. Introduction

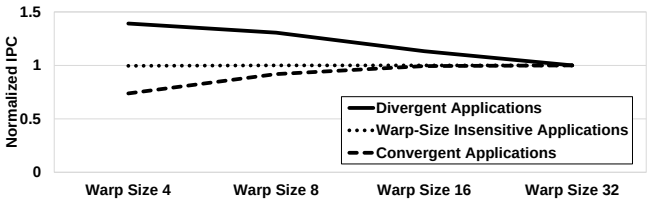
Contemporary *Graphics Processing Units* (GPUs) group collections of scalar threads into warps [19] or wavefronts [15] and execute them in *Single Instruction Multiple Thread* (SIMT) fashion. The number of threads in a warp is defined by the machine architecture. Grouping threads into warps amortizes the fetch, decode, and scheduling overhead associated with managing thousands of threads on a single *Streaming Multiprocessor* (SM) core. However, grouping threads into warps creates performance challenges when threads within the same warp execute different control flow paths (control flow divergence) or access non-contiguous regions in memory (memory divergence).

ISCA'15, June 13-17, 2015, Portland, OR USA

DOI: http://dx.doi.org/10.1145/2749469.2750410



(a) Performance of 165 real world applications using a warp size of 4, normalized to a warp size of 32.



(b) Performance versus warp size using a representative subset of applications presented in 1a. These applications are described in more detail in Section 5.

Figure 1: A survey of performance versus warp size.

Figure 1 plots the *Instructions Per Cycle* (IPC) resulting from shrinking warp size from 32 threads to 4 threads while keeping the machine's total thread-issue throughput and memory bandwidth constant. Figure 1a shows the effect of shrinking the warp size on a large suite of real world applications, while Figure 1b plots the harmonic mean performance of 15 applications which are selected to represent the 3 classes of workloads we study throughout this paper. We classify a workload as being *divergent* when performance increases as the warp size decreases, *convergent* when performance decreases as the warp size decreases, and *warp-size insensitive* when performance is independent of warp size. Figure 1 demonstrates that application performance is not universally improved when the warp size is decreased. This data indicates that imposing a constant machine-dependent warp size for the varied workloads running on GPUs can degrade performance on divergent applications, convergent applications, or both.

A large set of existing, highly regular GPU applications do not see any performance improvement at a smaller warp size. However, the divergent applications which do see a performance improvement represent a class of workloads that are important for future GPUs. Prior work such as [6, 7, 22, 25] has shown great potential for increasing the performance and energy efficiency of these types of workloads by accelerating them on a GPU. These applications include future rendering algorithms such as raytracing, molecular dynamics simulations,

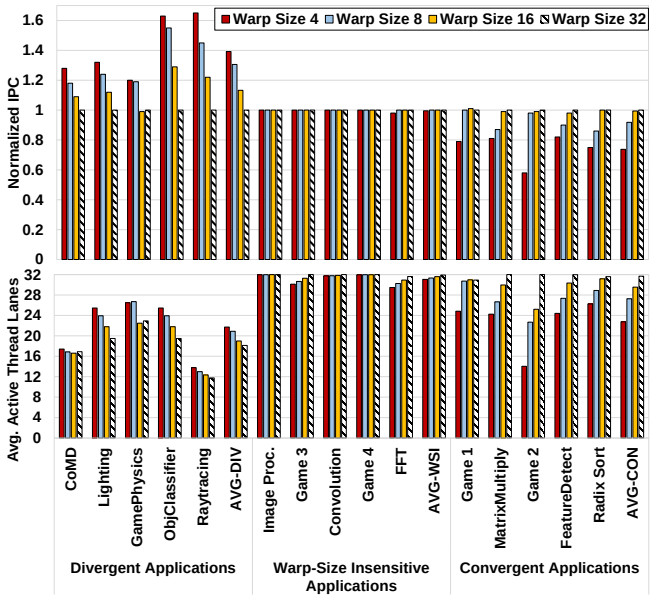


Figure 2: Normalized IPC (top) and the average number of active thread lanes on cycles when an instruction is issued (bottom). All configurations can issue 32 thread instructions per cycle.

advanced game physics simulations, and graph processing algorithms among many others. The goal of our proposed architecture is to evolve GPUs into a more approachable target for these parallel, but irregular, applications while maintaining the efficiency advantages of GPUs for existing codes.

Figure 2 plots the performance and resulting SIMT lane utilization of different warp sizes for each of the applications we study. Control-divergent applications have a low utilization rate at a warp size of 32 and see utilization increase as the warp size is decreased. These workloads are able to take advantage of executing different control flow paths simultaneously by using a smaller warp size. Convergent applications have a high lane utilization rate at a warp size of 32 and see their utilization decrease as the warp size is reduced. This reduction in utilization occurs because of increased pressure on the memory system caused by destroying horizontal locality across a larger warp. Horizontal locality occurs when threads within a warp or thread block access similar memory locations. Modern GPUs coalesce memory requests from the same warp instruction that access the same cache line. By allowing smaller groups of threads to proceed at different rates, the locality that existed across the same static instruction is spread over multiple cycles, causing additional contention for memory resources. We call this effect *memory convergence slip*.

In addition to the performance benefit convergent applications experience with larger warps, convergent and warp-size insensitive applications gain energy efficiency from executing with larger warps. A larger warp size amortizes the energy consumed by fetch, decode, and warp scheduling across more threads. When there is no performance benefit to executing with smaller warps, the most energy-efficient solution is to execute with as large a warp size as possible.

Our paper first examines the effect of providing a variable warp size, which can be statically adjusted to meet the performance and energy efficiency demands of the workload. We then propose Variable Warp Sizing, which gangs groups of small warps together to create a wider warp and dynamically adjusts the size of each gang running in the machine based on the observed divergence characteristics of the workload.

Prior work such as [11, 12, 5, 29, 30, 26, 31, 9, 24] proposes various techniques to improve *Single Instruction Multiple Data* (SIMD) efficiency or increase thread level parallelism for divergent applications on GPUs. However, the use of small warps is the only way to improve both SIMD efficiency and thread level parallelism in divergent code. These prior works focus on repacking, splitting, and scheduling warps under the constraint of a fixed-size warp. Our work approaches the problem from the other direction. We simplify the acceleration of divergent workloads by starting with a small warp size and propose a straightforward ganging architecture to regain the efficiencies of a larger warp. Prior work can improve the performance of divergent applications when the total number of unique control paths is limited and the number of threads traversing each respective path is large. Starting with smaller warps allows our microarchitecture to natively execute many more concurrent control flow paths, removing this restriction. Section 7 presents a more detailed quantitative and qualitative comparison to prior work.

In this paper, we make the following contributions:

- We characterize the performance, control flow/memory divergence, and fetch/decode effects of different warp sizes on a large number of graphics and compute GPU workloads.
- We demonstrate that reducing the warp size of modern GPUs does not provide a universal performance advantage due to interference in the memory system and an increase in detrimental scheduling effects.
- We explore the design space uncovered by enabling a dynamic, variable warp size. We quantify the effects of scheduling and gang combination techniques when the machine has the flexibility to issue from multiple control flow paths concurrently.
- We propose a novel warp ganging microarchitecture that makes use of a hierarchical warp scheduler, enabling divergent applications to execute multiple control flow paths while forcing convergent ones to operate in lock-step.

2. Baseline Architecture

Figure 3 depicts our model of a modern GPU, consisting of several streaming multiprocessor cores (SMs) connected to the main memory system via an interconnection network. This paper studies the detailed design of an SM.

Our pipeline decouples the fetch/decode stages from the issue logic and execution stage by storing decoded instructions in per-warp instruction buffers (similar to the pipeline model

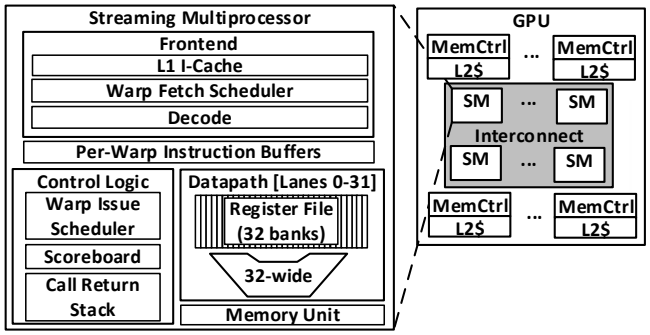


Figure 3: Baseline GPU architecture.

in GPGPU-Sim [1]). Each warp stores one decoded instruction in the instruction buffer. Each instruction entry in the buffer also contains a valid bit, which is set when an instruction is filled into the buffer, and a ready bit, which is set when the in-order scoreboard indicates the instruction is able to execute.

The front-end of each SM includes an L1 instruction cache which is probed once per cycle by the warp fetch scheduler. The warp fetch scheduler determines which empty entry in the instruction buffer is to be filled. On the execution side, a warp issue scheduler selects one decoded, ready instruction to issue each cycle. The register file consists of one bank per lane and the datapath executes 32 threads per cycle in SIMT fashion.

Our memory system is similar to that used in modern Kepler [28] GPUs. Each SM has a software-managed scratchpad (known as shared memory in CUDA [27]), an L1 data cache, and a texture unit. Access to the memory unit is shared by all lanes. To reduce the number of memory accesses generated from each warp, GPUs coalesce memory requests into cache line sized chunks when there is spatial locality across the warp. A single instruction that touches only one cache line will generate one transaction that services all 32 lanes. Our main memory system includes a variable latency, fixed bandwidth DRAM model.

Program Counters (PCs) and control flow divergence information for each warp are stored on a compiler-managed call return stack (CRS). The stack's operation is similar to the post dominator reconvergence stack presented by Fung et al. [11]. Our baseline SM is heavily multithreaded and is able to schedule up to 32 warps (1024 threads).

3. Trade-offs of Warp Sizing

This section details the effect of warp size on both the memory system and SM front-end. This data motivates an architecture that is able to dynamically vary warp size.

3.1. Warp Size and Memory Locality

Figure 4 shows the effect warp size has on L1 data cache locality in terms of hits, misses, and *Miss Status Holding Register* (MSHR) merges *Per Thousand Instructions* (PKI) for the applications we study. As the warp size is decreased, some applications see an increase in the number of L1 data cache accesses. This phenomenon occurs when memory accesses that were coalesced using a larger warp size become distributed over multiple cycles when smaller warps are used, an effect we term *memory convergence slip*.

In the divergent applications, memory convergence slip does not significantly degrade performance for two reasons. First, an application that is control flow diverged has less opportunity for converged accesses because fewer threads are participating in each memory instruction. Second, even when convergence slip occurs on a divergent application, as it does in CoMD, ObjClassifier, and Raytracing, it also often results in more cache hits, mitigating the effect on performance. While the control-divergent Raytracing application also sees an increase in misses, the performance cost of these misses is offset by the increased lane utilization observed with smaller warps.

In the convergent applications, memory convergence slip has a greater effect on performance. All of these applications see both an increase in the total number of memory accesses and cache misses at smaller warp sizes. Radix Sort and Game 2 also see an increase in MSHR merges. The loss in throughput caused by additional traffic to the L2 data cache and DRAM in these applications is not offset by any increase in lane utilization, as these applications already have high SIMT utilization at larger warp sizes. Perhaps not surprisingly, L1 locality for the warp-size insensitive applications is insensitive to the warp size.

3.2. Warp Size and SM Front-end Pressure

Figure 5 plots the average number of instructions fetched per cycle at various warp sizes. Decreasing the warp size places increasing pressure on the SM's front-end. Convergent and warp-size insensitive applications see a nearly linear increase in fetch requests as the warp size is reduced. This data indicates that a fixed 4-wide warp architecture increases front-end energy consumption for non-divergent applications, even if the performance does not suffer. While divergent applications also see increased front-end activity, the ability of the architecture to exploit many more independent control paths is fundamental to increasing the performance of these applications. Our design focuses on creating a flexible machine that is able to expand and contract the size of warps executing in the system. The best warp size for a given application balances the demands for independent control flow with the limitations due to memory convergence slip.

4. Variable Warp Sizing

This section describes the high level operation of Variable Warp Sizing, discusses the key design decisions, and details the operation of each architectural component. We selected four threads as the minimum warp size for three reasons: (1) graphics workloads commonly process threads in groups of four known as quads, (2) the performance opportunity for the compute workloads we examined reaches diminishing returns at warp sizes smaller than four, and (3) the area overhead rises

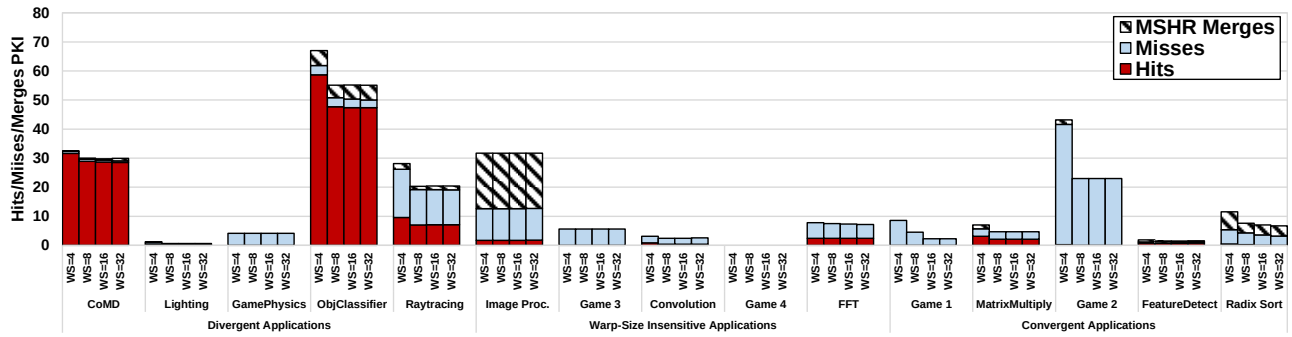


Figure 4: L1 data cache hits, misses, and MSHR merges *per thousand instructions* (PKI) at different warp sizes.

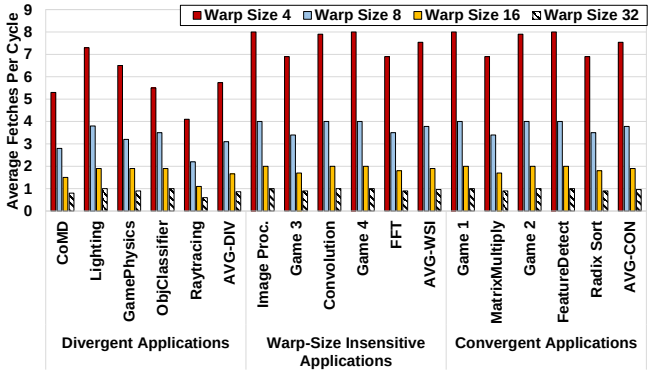


Figure 5: Average instructions fetched per cycle. Fetch bandwidth is scaled to match issue bandwidth for each warp size.

notably at warp sizes smaller than four. We discuss the area trade-offs of different warp sizes in Section 6.7.

4.1. High-level Operation

The goal of VWS is to create a machine that is able to dynamically trade off MIMD-like performance with SIMD-like efficiencies depending on the application. Our proposed variable warp sized machine shrinks the minimum warp size to four threads by splitting the traditional GPU datapath into eight unique slices. Each slice can fetch, decode, and issue instructions independent of other slices. Figure 6 presents the microarchitecture of our proposed design. Each slice is statically assigned threads in a linear fashion: threads 0-3 are assigned to slice 0, 4-7 to slice 1, and so on. Threads cannot migrate between slices.

VWS does not change the number of register file banks in the SM or impose any additional communication between them. As in our baseline, each four-lane slice of the datapath receives its own set of four register file banks (❶ in Figure 6). VWS requires no changes to the memory unit (❷), which includes the shared scratchpad memory, L1 data cache, and texture cache. All memory requests generated by any slices in the same cycle are presented to the memory unit as a single 32-thread access in the same manner as when executing 32-wide warps. The coalescing unit also operates in the same fashion as the baseline; multiple threads accessing the same cache line in the same cycle generate only one memory request.

To facilitate warp sizes greater than four, we introduce the warp ganging unit (❸), which is able to override local per-slice fetch/decode (❹) and issue (❺) decisions. The gang front-end (❻) performs instruction fetch and decode once for all small warps participating in a gang. The gang issue scheduler enforces lock-step execution of all slices participating in a given gang. The warp ganging unit is discussed in more detail in Section 4.2.

When VWS is operating in ganged-only mode, the per-slice front-end logic (❼) and warp issue scheduler (❽) are disabled to save energy. When operating in slice-only mode, each SM slice uses its independent front-end to fetch and decode instructions. When both gangs and independent warps are present in the system at the same time, gangs are given both fetch and issue priority. This policy ensures that gangs remain in lock-step as long as possible. When possible, independent warps are used to fill in holes in the executing gang. Each slice front-end includes an L0 I-cache (❾) to reduce pressure on the larger L1 I-cache (❿) which is shared by all slices in the SM. Without L0 I-caches, providing peak throughput in slice-only mode would require 8× the L1 I-cache bandwidth. Our microarchitecture allows 9 separate fetch schedulers (one for each of eight slices and one for gangs) to request instructions from the L1 I-cache. We study the effects of scaling L1 I-cache bandwidth in Section 6. Arbitration to determine which scheduler is granted L1 access is done by the L1 fetch arbiter (⓫), described in more detail in Section 4.6

This microarchitecture can be run in gang-only or slice-only mode (effectively locking the warp size at 32 or 4 respectively). However, our proposed solutions evaluated in Section 6 and described in the remainder of Section 4 operate by beginning execution in ganged mode. Sections 4.4 and 4.5 describe how gangs can be split and reformed on a per-gang basis.

4.2. Warp Ganging Unit

The goal of the *warp ganging unit* is to force independent slices to fetch, decode, and execute instructions in lock-step *gangs* when threads across multiple slices are control-convergent. Several factors motivate such ganging. First, issuing memory accesses from convergent applications without lock-step execution places significantly more pressure on the memory system and degrades performance. Second, the

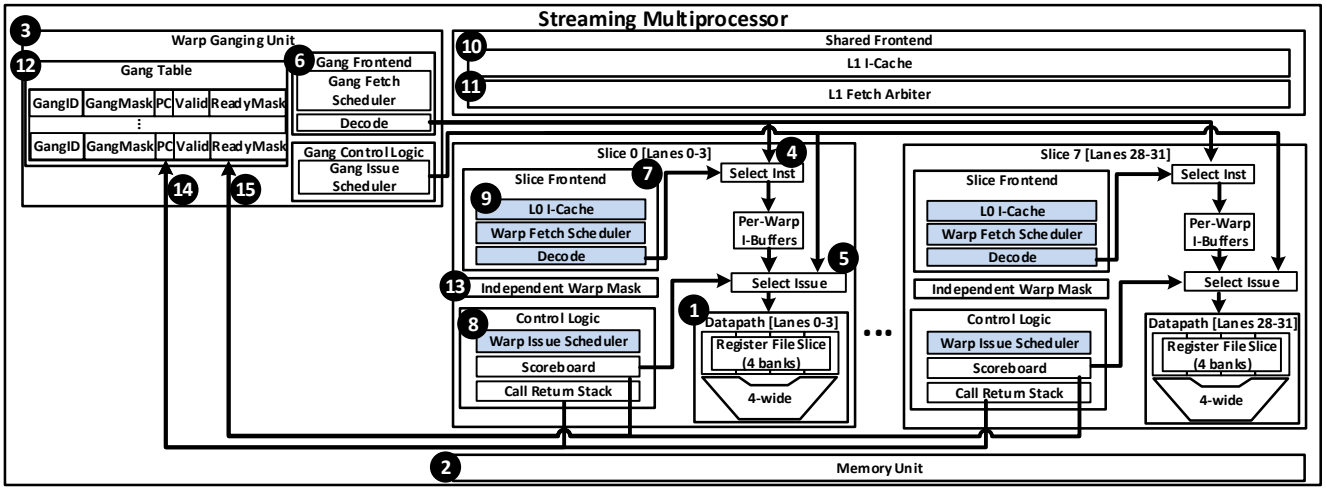


Figure 6: Variable Warp Sizing SM microarchitecture. Shaded units are disabled when operating in ganged mode to save energy.

system can amortize front-end energy consumption across more threads when some or all small warps across the slices in an SM are executing the same instruction.

When a kernel begins and thread blocks are assigned to an SM, gangs are created from the thread blocks in the same fashion as 32-wide warps are created in our baseline system. Each gang is statically assigned eight 4-wide warps, one from each slice. Information about which warps are participating in which gang is stored in the gang table (12). Each entry in the gang table contains a GangID, an 8-bit GangMask (indicating which slices are participating in the gang), the current PC of the gang, a valid bit (which is cleared when the gang's instruction buffer entries are empty), and a ReadyMask which indicates which warps in the gang can issue. To simplify the design of the gang unit, warps are not allowed to migrate between gangs. We implemented more complex gang forming and reforming schemes, but saw no significant performance or energy advantage for our workloads. All warps not participating in a gang (*unganged warps*) are managed independently by their respective slice. Each slice stores an independent warp mask (13) indicating which of its warps are managed independent of the warp ganging unit.

4.3. Gang Table

The *gang table* tracks all information necessary for scheduling gangs as well as for managing gang splitting and reformation. The baseline SM described in Section 2 has a capacity of 1024 schedulable threads organized into 32 warps of 32 threads each. The VWS SM has the same total thread capacity, but organized into a total of 256 warps of 4-threads each, or 32 4-thread warps per slice. At kernel launch, the threads are aggregated into maximally-sized gangs of eight 4-wide warps, or 32 threads per gang to match the baseline architecture. The term *original gang* is used throughout this paper to describe a gang of warps that is created when a thread block is initially assigned to an SM.

When a gang splits, more entries in the gang table become necessary. Because individual warps are not managed by the warp ganging unit, a gang of 8 warps can split into at most 4 gangs, with a minimum of two warps per gang. Further subdivision yields singleton warps which are managed within each slice. Thus the maximum number of entries needed in the gang table to track the smallest gangs is 128 (32 original gangs $\times$ 4). These 128 entries can be organized in a set-associative manner with 32 sets, one set per original gang and four entries representing up to 4 different gang splits.

Each entry in the gang table contains a unique *GangID* identifier and *GangMask* that indicates which slices are participating in this gang. Since warps can only be ganged with other members of their original gang, all warps from the same original gang access the same set in the gang table and must have GangIDs that are in the same group. For example, warp 0 in each slice can only be a member of gangs 0–3. With this organization, each warp's index in the GangMask is simply the warp's slice number.

To perform fetch and issue scheduling, the warp ganging unit requires information from the slices. Specifically, the gang front-end must know the next PC for each gang, and the gang issue scheduler must know when all warps in a gang have cleared the scoreboard. Per warp call return stack (or reconvergence stack) tracking is done locally in each slice. To track per-gang PCs and handle gang splitting when control flow divergence is encountered, each slice signals the warp ganging unit when the PC at the top of a warp's stack changes (14). Instruction dependence tracking is also done in each slice, even when operating in gang-only mode. Keeping the dependence information local to each slice makes transferring warps from ganged to unganged simpler and decreases the distance scoreboard control signals must travel. The warp ganging unit tracks dependencies for an entire gang in a ReadyMask by receiving scoreboard ready signals from each slice (15).

The gang table also contains a per-entry valid bit to track instruction buffer (I-Buffer) status. The warp gang unit is responsible for both fetching and issuing of gangs. The gang

unit front-end stores decoded instructions in each member warp's per-slice I-Buffer. The valid bit is set by the gang fetch scheduler when a gang's per-slice I-Buffer entries are filled and is cleared by the gang issue scheduler when the associated instruction has been issued. All member warps in a gang issue their instructions in lockstep from their respective slice-local I-Buffers. This bit is managed internally by the warp ganging unit and does not require any input from the slices.

4.4. Gang Splitting

The warp ganging unit decides when gangs are split and reformed based on a set of heuristics evaluated in Section 6. To make splitting decisions, the warp gang unit observes when control flow and memory divergence occurs. Control flow divergence is detected by observing the PCs sent to the ganging unit by each slice. PCs from the slices undergo a coalescing process similar to global memory accesses. If all warps in a gang access the same PC, no splitting is done. If any warp in the gang accesses a different PC, the gang is split. If more than one warp accesses a common PC, a new gang is formed for these warps. If only one warp accesses a given PC, that warp is removed from the control of the ganging unit and a signal is sent to that warp's slice, transferring scheduling to the local slice. All VWS configurations explored in this work split gangs whenever control flow divergence is detected.

In addition to control flow divergence, memory latency divergence is another motivation for gang splitting. Memory latency divergence can occur when some threads in a warp hit in the data cache while other threads must wait for a long-latency memory operation to complete. Prior work such as Dynamic Warp Subdivision [24] has suggested warp subdivision to tolerate memory latency divergence.

Section 6 evaluates VWS architecture configurations that can split gangs when memory latency divergence is observed among member warps. Memory latency divergence is detected when scoreboard ready bits for different warps in a gang are set at different times when completing memory instructions. Tracking which warps in a gang are ready is done through the ReadyMask. We evaluate VWS with two different types of gang splitting on memory divergence. *Impatient Splitting* is the simplest form of gang splitting on memory divergence. If any warp in a gang sets its ready bit before any other member warps, the gang is completely split; all members participating in the gang become independent warps. Impatient splitting simplifies the splitting process and allows highly memory divergent workloads to begin independent execution as quickly as possible. *Group Splitting* enables warps that depend on the same memory access to proceed together as a new gang. When more than one warp in a gang has its ready bit set in the same cycle, a new gang is created from those warps. Any singleton warps that result from this process are placed under independent, per-slice control.

4.5. Gang Reformation

In addition to splitting gangs, VWS supports the reformation of gangs that have been split. The warp ganging unit decides if warps or gangs from the same original gang should be re-ganged. While we explored numerous policies, two simple but effective choices emerged: (1) opportunistic reformation and (2) no reformation. To simplify the re-ganging hardware, only one gang can be reformed each cycle. To perform opportunistic gang reformation, one original gang is selected each cycle, in round-robin order. The hardware compares the PCs from each of the original gang's new gangs or independent warps, with a worst-case 8-way comparison if the gang has completely split apart. If any groups of two or more of these warps or gangs have the same PC, they are merged. Section 6 describes policies to promote more gang reformation by forcing gangs and warps to wait at common control flow post dominator points in the code.

4.6. Instruction Supply

To avoid building a machine with 8× the global fetch bandwidth when VWS is operating in completely independent slice mode, the fetch bandwidth of the L1 instruction cache is limited. We evaluated several different L1-I cache bandwidths and determined that with modestly sized L0 I-caches, L1 I-cache bandwidth can be scaled back to two fetches per cycle and achieve most of the performance of allowing 8 fetches per cycle. The *global fetch arbiter* determines which fetch schedulers access the L1 I-cache's 2 ports on any given cycle. The gang fetch scheduler is always given priority to maximize the number of lanes serviced. The remaining fetch bandwidth is divided among the per-slice warp fetch schedulers. Individual warps are distributed to the slices in round-robin fashion (warp 0 is assigned to slice 0, warp 1 to slice 1, and so on). An arbitration scheme prioritizes slice requests to ensure that each slice gets fair access to the L1 I-cache.

5. Experimental Methodology

The results in this paper are collected using a proprietary, cycle-level timing simulator that models a modern GPU streaming multiprocessor (SM) and memory hierarchy similar to that presented in Section 2. The simulator is derived from a product development simulator used to architect contemporary GPUs. Table 1 describes the key simulation parameters. The simulator processes instruction traces encoded in NVIDIA's native ISA and generated by a modern NVIDIA compiler. Traces were generated using an execution-driven, functional simulator and include dynamic information such as memory addresses and control flow behavior. We simulate a single SM with 32 SIMT execution lanes that execute 32-wide warps as our baseline, similar to that described in [13]. For warps smaller than 32, we use the same memory system but maintain a fixed count of 32 execution lanes sliced into the appropriate number of groups. We model a cache hierarchy and memory system similar to

Table 1: Baseline simulator configuration.

# Streaming Multiprocessors	1
Execution Model	In-order
Warp Size	32
SIMD Pipeline Width	32
Shared Memory / SM	48KB
L1 Data Cache	64KB, 128B line, 8-way LRU
L2 Unified Cache	128KB, 128B line, 8-way LRU
DRAM Bandwidth	32 bytes / core cycle
Branch Divergence Method	ISA Controlled Call Return Stack
Warp Issue Scheduler	Greedy-Then-Oldest (GTO) [32]
Warp Fetch Scheduler	Loose Round-Robin (LRR)
ALU Latency	10 cycles

contemporary GPUs, with capacity and bandwidth scaled to match the portion available to a single SM.

The trace set presented was selected to encompass a wide variety of behaviors. Traces are drawn from a variety of categories, including High Performance Computing (HPC), games, and professional/consumer compute application domains such as computer vision. A third of the selected traces belong to each of the three categories described in Section 1: (1) divergent codes that prefer narrow warps, (2) convergent codes that prefer wider warps, and (3) codes that are mostly insensitive to warp size.

6. Experimental Results

This section details experimental results for the Variable Warp Sizing microarchitecture. First, we quantify performance for several configurations of VWS and then characterize instruction fetch and decode overhead and the effectiveness of mitigation techniques. We perform several sensitivity studies exploring various design decisions for gang scheduling, splitting, and reforming. We demonstrate how gang membership evolves over time for some sample workloads. Finally, we examine area overheads for the proposed design.

6.1. Performance

Figure 7 plots the performance of multiple warp sizes and VWS, using different warp ganging techniques. All techniques can issue 32 thread instructions per cycle. Fetch and decode rates are scaled with the base warp size; WS4 and WS32 can fetch and decode eight instructions per cycle and one instruction per cycle, respectively. The VWS configurations use a base warp size of 4 and can fetch up to 8 instructions per cycle from the L1 I-cache. Simulating our ganging techniques with 8 times the L1 I-cache fetch throughput allows us to explore the maximum pressure placed on the global fetch unit without artificially constraining it. Section 6.2 demonstrates that VWS using the L0 I-caches described in Section 4 and an L1 I-cache with only 2× the bandwidth achieves 95% of the performance of using 8× the L1 I-cache bandwidth on divergent applications. Warp-size insensitive and convergent applications are insensitive to L1 I-cache bandwidth. We chose the following VWS configurations based on an exploration of the design space detailed in the rest of this section.

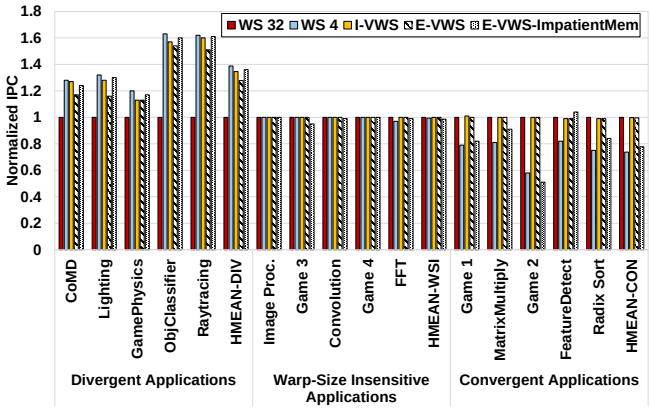


Figure 7: Performance (normalized to WS 32) of large warps, small warps, and different warp ganging techniques.

WS 32: The baseline architecture described in Section 2 with a warp size of 32.

WS 4: The baseline architecture described in Section 2 with a warp size of 4.

I-VWS: Inelastic Variable Warp Sizing with a base warp size of 4, where gangs are split only on control flow divergence. Warps are initially grouped together into gangs of 8 warps (32 threads total). Upon control flow divergence, gangs are split based on each warp's control flow path. Once split, they are never recombined. The ganging unit selects up to two gangs to issue each cycle. Slices that do not receive a ganged instruction pick the next available warp from their pool of unganged warps. The ganged scheduler uses a Big-Gang-Then-Oldest (BGTO) scheduling algorithm, where gangs with the most warps are selected first. Gangs with the same number of warps are prioritized in a Greedy-Then-Oldest (GTO) fashion. Per-slice schedulers manage independent warps using a GTO scheduling mechanism.

E-VWS: Elastic Variable Warp Sizing. Warps are split on control flow divergence and combined in an opportunistic fashion when multiple gangs or singleton warps arrive at the same PC on the same cycle. Gangs can only be created from members of an original gang. A maximum of 2 gangs or warps can be combined per cycle.

E-VWS-ImpatientMem: Warp ganging similar to E-VWS, except that gangs are also split when memory divergence occurs across warps in the same gang. Whenever any memory divergence occurs in a gang, the entire gang is split. Gangs are recombined in the same opportunistic fashion as E-VWS.

Figure 7 shows that the I-VWS warp ganging microarchitecture is able to achieve a 35% performance improvement on divergent applications over a static warp size of 32. This improvement is within 3% of using a warp size of 4 on divergent applications and it results in no performance loss on convergent applications where simply using a warp size of 4 results in a 27% slowdown. This data also demonstrates that splitting gangs on control flow divergence without performing any gang recombining, the simplest solution, provides the best overall

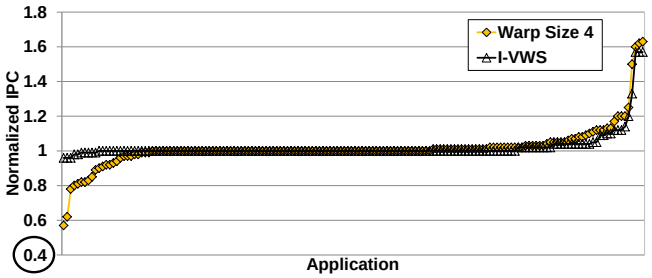


Figure 8: Performance (normalized to WS 32) of I-VWS and WS 4 on all the applications from in Figure 1a.

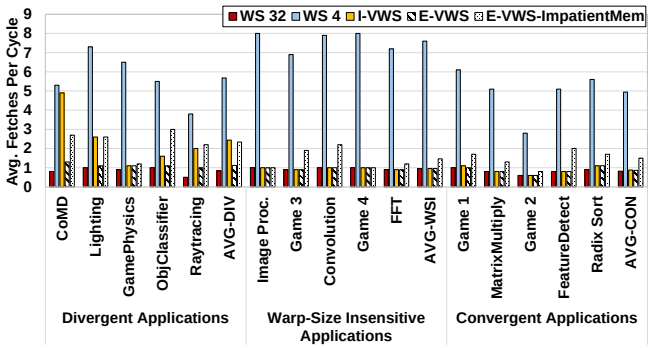


Figure 9: Average fetches per cycle with different warp sizes and ganging techniques.

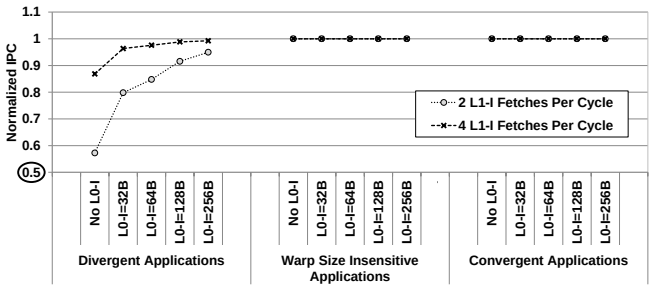


Figure 10: Average performance of I-VWS at different L1 I-cache bandwidths and L0 I-cache sizes. Normalized to I-VWS with 8x L1-I cache bandwidth.

performance for these workloads. Adding opportunistic gang recombining (E-VWS in Figure 7) actually results in a small performance decrease on divergent applications. This decrease is caused by scheduling and packing problems associated with attempting to issue across more slices at once. When gangs are larger, there is a greater likelihood that multiple gangs need to issue to the same slice on the same cycle.

Elastically splitting and regrouping makes no performance difference on convergent and warp-size insensitive applications because these applications experience little or no control flow divergence. Recombining gangs for the divergent workloads makes little performance difference when the hardware has the ability to issue many smaller gangs (or single 4-sized warps) because remaining unganged is unlikely to result in a loss of utilization. Having the ability to concurrently issue multiple paths at the slice granularity makes control flow reconvergence less performance critical than when only one path can be executed concurrently.

Figure 7 also quantifies the effect of splitting gangs on memory divergence (E-VWS-ImpatientMem). Reducing the effect of memory divergence helps some of the divergent applications like Lighting, ObjClassifier, and Raytracing and provides a modest 2% performance increase over I-VWS on the divergent applications. However, allowing gangs to split based on memory divergence results in significant performance degradation on Game 1, Game 2, and Radix Sort in the convergent application suite, resulting in an average slowdown of 22% on the convergent applications. Like 4-sized warps, this loss in performance can be attributed to memory convergence slip. Formerly coalesced accesses become uncoalesced and create excessive pressure on the memory system causing unnecessary stalls.

Figure 8 shows the performance of all 165 applications. The figure demonstrates that the ganging techniques used in I-VWS are effective for all the applications studied. I-VWS tracks warp size 4 performance on the divergent applications and eliminates warp size 4 slowdown on the convergent applications at the left side of the graph.

6.2. Front-end Pressure

Figure 9 plots the fetch pressure of several warp sizes and ganging configurations. For the divergent applications, I-VWS results in 57% fewer fetch/decode operations required each cycle versus a warp size of 4. This reduction in fetch/decode represents a significant energy savings while providing almost all of the performance of 4-sized warps. By opportunistically recombining gangs for divergent applications, E-VWS requires a further 55% less fetch/decode bandwidth than I-VWS, at the cost of some performance. On divergent applications, E-VWS-ImpatientMem increases fetch/decode pressure versus E-VWS but not more than I-VWS.

On the convergent and warp-size insensitive applications, the ganging configurations that do not split on memory divergence show fetch pressure equal to that of warp size 32. Because these applications lack control flow divergence, gangs rarely split and I-VWS operates exclusively in ganged mode. However, when gangs are split on memory divergence, the skewing of memory access returns causes a significant increase in the number of fetch/decodes per cycle.

Figure 10 plots the performance of I-VWS at different L1 I-cache bandwidths and L0 I-cache sizes. Because the divergent applications traverse multiple independent control flow paths, restricting L1 I-cache bandwidth results in a significant performance loss. However, the inclusion of per-slice L0 I-caches, which are probed first when independent warps fetch instructions, vastly decreases the performance loss. With only 2× the L1 I-cache bandwidth of the baseline architecture, the addition of small 256B L0s are able to cover most of the bandwidth deficiency at the L1. Since they remain in ganged operation, the warp-size insensitive and convergent applications are insensitive to L1 I-cache fetch bandwidth.

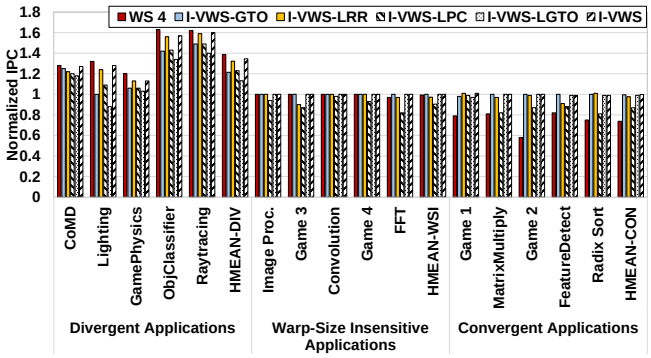


Figure 11: Performance (normalized to WS 32) of warp ganging with different schedulers.

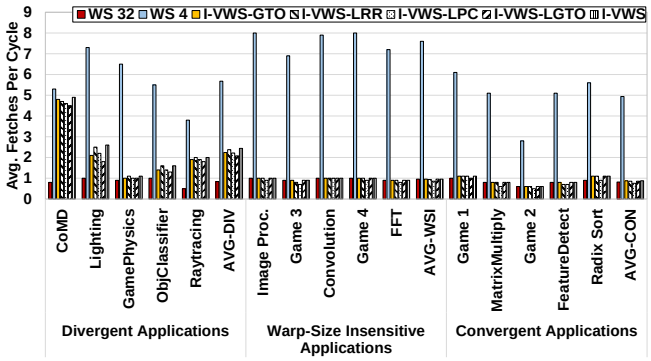


Figure 12: Averages fetches per cycle with different schedulers.

6.3. Gang Scheduling Policies

We measured the sensitivity of performance and instruction fetch bandwidth to several different gang scheduling policies. All gang schedulers attempt to issue up to two gangs per cycle, and local per-slice schedulers attempt to issue on any remaining idle slices. We examine the following policies:

I-VWS: As described in Section 6.1, the gang issue scheduler prioritizes the largest gangs first Big-Gangs-Then-Oldest (BGTO) and per-slice schedulers are Greedy-Then-Oldest (GTO).

I-VWS-GTO: Similar to I-VWS, except the gang issue scheduler uses a greedy-then-oldest policy.

I-VWS-LRR: Similar to I-VWS, except both the gang issue scheduler and per-slice schedulers use a Loose-Round-Robin (LRR) scheduling policy.

I-VWS-LPC: Similar to I-VWS, except both the gang issue scheduler and per-slice schedulers prioritize gangs/warps with the lowest PC first.

I-VWS-LGTO: Similar to I-VWS, except the gang issue scheduler prioritizes gangs with the fewest warps first Little-Gangs-Then-Oldest (LGTO). Per-slice schedulers use a GTO policy.

Figure 11 shows that the performance of the divergent applications is sensitive to the gang scheduler choice. The lowest-PC-first configuration results in a universal performance reduction across all the applications. Little-Gangs-Then-Oldest

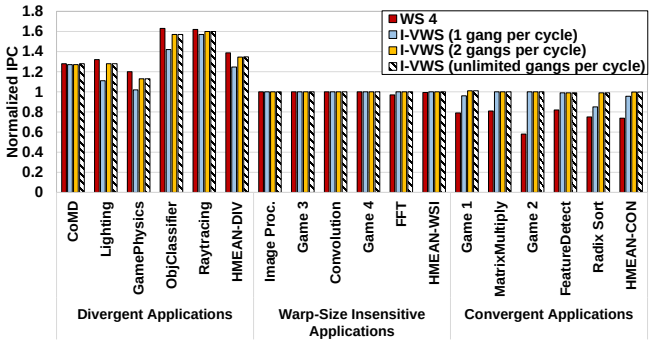


Figure 13: Performance (normalized to WS 32) when the number of gangs able to issue each cycle is changed.

(I-VWS-LGTO) creates a scheduling pathology on the divergent applications. Prioritizing the smallest gangs first is bad for performance because the gang issue scheduler can only select a maximum of 2 gangs for execution each cycle; giving the smallest ones priority can limit utilization by delaying the execution of gangs with many warps. We also observed that prioritizing little gangs was detrimental even when more than two gangs could be scheduled per cycle because little gangs block the execution of larger gangs. Figure 12 shows the resulting fetch and decode requirements for different gang scheduling policies. Although the choice of gang scheduler has a significant effect on performance, it has little effect on fetch/decode bandwidth. This insensitivity occurs because gang scheduling has nothing to do with gang splitting when gangs are split only for control flow divergence and are not recombined. When splitting gangs on memory divergence is enabled, the effect of scheduling on the fetch rate is much greater.

Figure 13 plots performance when the number of gangs selectable per cycle by the gang issue scheduler is set to one, two, or unlimited (up to four). This data shows that limiting the gang scheduler to a single gang per cycle reduces the performance of the divergent applications by 10% versus the baseline of two gangs per cycle. Allowing the gang scheduler to pick unlimited gangs per cycle results in performance that is within 1% of two gangs per cycle. Any slices not consumed by the gang scheduler may be used whenever possible by any singleton warps managed by local slice schedulers. We choose to limit the gang scheduler to two gangs per cycle to balance performance and scheduler complexity.

6.4. Gang Reformation Policies

Figures 14 and 15 plot performance and instruction fetches per cycle when the following policies are used to reform gangs after they have been split:

E-VWS: As described in Section 6.1, gangs are reformed on an opportunistic basis only.

E-VWS-Sync<xx>: Similar to E-VWS, except that when warps reach a compiler-inserted call-return stack sync instruction, they wait for recombination. These instructions are

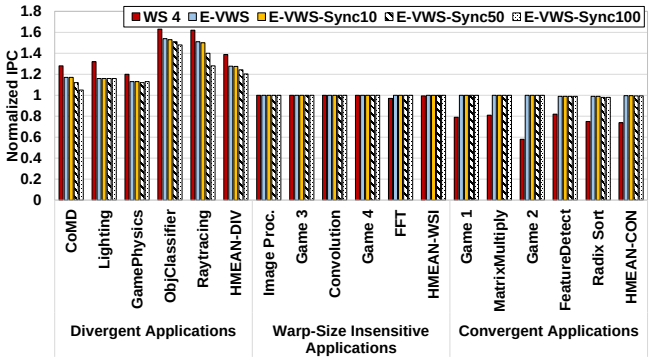


Figure 14: Performance (normalized to WS 32) of elastic gang reformation techniques.

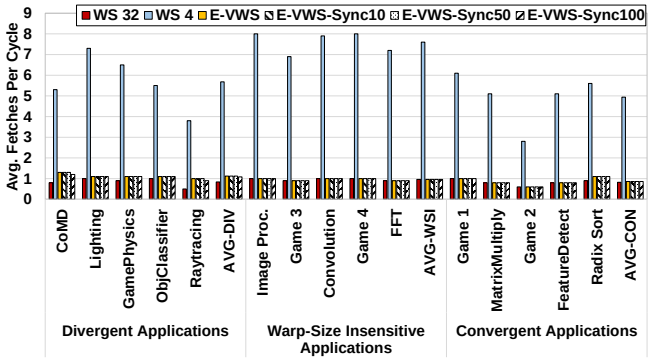


Figure 15: Average fetches per cycle with different gang reformation techniques.

already inserted, typically at basic block post-dominators, to enforce the NVIDIA call-return stack architecture. <XX> indicates how many cycles a warp will wait at the sync point for potential reganging.

Forcing warps to wait at control flow post-dominator points can potentially improve gang reformation, leading to more or larger gangs and reduced front-end energy while hopefully resulting in minimal performance degradation. Figures 14 and 15 demonstrate that waiting at sync points results in a performance loss on our divergent applications. We see minimal decrease in the number of fetches per cycle as waiting time is increased, and any energy efficiency gained from this reduction would be more than offset by the loss in performance. The warp-size insensitive and convergent applications contain fewer compiler-inserted sync points, experience little or no control flow divergence, and may spend much or all of their execution time fully ganged. As a result, their performance is largely unaffected by wait time at infrequent sync points. Thus we conclude that forcing warps to wait at post-dominators provides little to no benefit; most of the reduction in fetch pressure is captured by opportunistic reganging in E-VWS.

6.5. Gang Splitting Policies

Figure 16 explores the use of the following gang splitting policies without any gang reformation:

I-VWS: As described in Section 6.1. Warps are split only on control flow divergence.

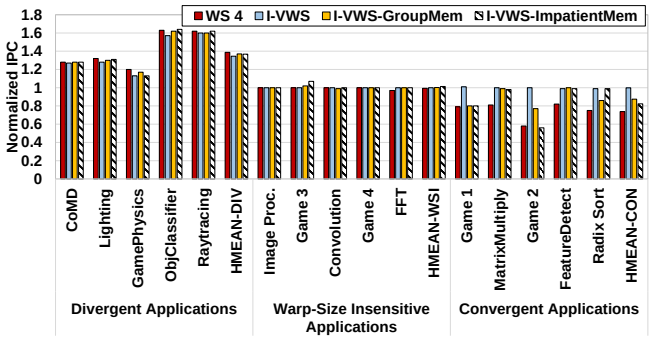


Figure 16: Performance (normalized to WS 32) of different gang splitting policies.

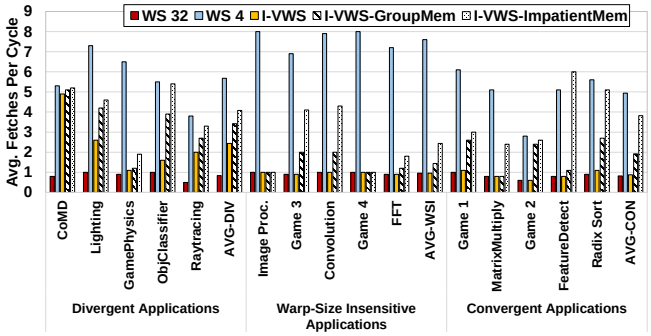


Figure 17: Average fetches per cycle using different gang splitting policies.

I-VWS-GroupMem: Warp ganging similar to I-VWS except gangs are also split on memory divergence. As memory results return for a gang, all warps in a gang that are able to proceed based on the newly returned value form a new gang. Gangs are never recombined.

I-VWS-ImpatientMem: Warp ganging similar to I-VWS-GroupMem except gangs that experience any memory divergence are completely subdivided into individual warps.

As in Section 6.1, Figure 16 demonstrates that splitting on memory latency divergence can have a small performance advantage on some divergent applications, but has a large performance cost on convergent ones. Minimizing the amount of splitting that occurs on memory divergence (I-VWS-GroupMem) gains back some of the performance lost for Game 2 but creates problems in Radix Sort. Overall, splitting on memory divergence is a net performance loss due to its negative effect on convergent applications.

Figure 17 plots the resulting number of instructions fetched per cycle when different gang splitting policies are used. This data demonstrates that even though splitting on memory divergence may be a small performance win for divergent applications, the number of instructions fetched increases greatly as a result, by 41% and 69% for I-VWS-GroupMem and I-VWS-ImpatientMem, respectively.

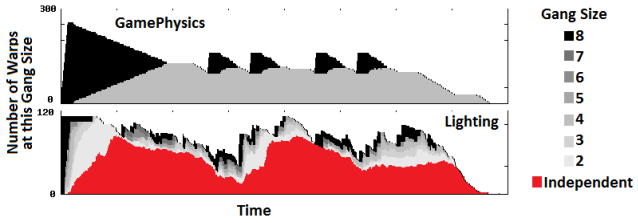


Figure 18: Gang sizes versus time for I-VWS.

6.6. Gang Size Distribution

Figure 18 visualizes how gang sizes change over time for two example divergent workloads, GamePhysics and Lighting. Each warp assigned to the SM on any given cycle is classified according to the size of the gang to which it belongs. For example in the Lighting application, execution begins with 120 4-wide warps assigned to the SM. The black bar at cycle 0 indicates that all warps start out in their original gangs of size 8. As time progresses, the original gangs split apart into smaller gangs until eventually most warps in the SM are executing independently. In contrast, GamePhysics exhibits much more structured divergence. The SM begins execution with 300 warps all in their original gangs. Over time, the warps split in two (the grey color in the GamePhysics graph represents warps participating in a gang of 4). One half of the gang exits, while the other half continues executing in lock step. These two plots illustrate how I-VWS reacts to different kinds divergence. Most of the divergent workloads studied react similar to Lighting. Similar plots for E-VWS show gangs splitting and reforming as time progresses. The plots collected for the convergent applications show that warps stay in their original gang throughout execution.

6.7. Area Overheads

Table 2 presents an estimate of the area required to implement I-VWS in a 40nm process. Column two presents the raw area estimate for each I-VWS component, while columns three and four present a rolled-up incremental SM area increase for 4-wide and 8-wide warps, respectively. We model the L1 I-cache using CACTI [36] at 40nm. The L0 I-cache, decoded I-Buffers, and the gang table are small but dominated by the storage cells required to implement them. We estimate the area of these structures by using the area of a latch cell from the NanGate 45nm Open Cell library and scaling it to 40nm. We multiply the resulting cell area ($2.1\mu m^2$) by the number of bits and a factor of 1.5 to account for area overheads including control logic. For the per-slice scoreboards, we use a larger FlipFlop cell ($3.6\mu m^2$ scaled to 40nm) from the NanGate library and 3× area overhead factor to account for the comparators necessary for an associative lookup. Compared to the scoreboard described in [9], ours has fewer bits and noticeably less area. Finally, to estimate the area cost of the additional control logic required for slicing the SIMD datapath, we examine published literature on the percentage of total

Table 2: Area overhead estimates.

Component	Component Area	Additional SM Area 4-wide warps	Additional SM Area 8-wide warps
Single-ported L1 I-cache (64KB)	0.087mm^2		
Dual-ported L1 I-cache (64KB)	0.194mm^2	0.108mm^2	0.108mm^2
L0 I-cache (256B)	0.006mm^2	0.052mm^2	0.026mm^2
Decoded I-Buffers (4Kbits)	0.013mm^2	0.103mm^2	0.052mm^2
Gang Table (128 entries)	0.026mm^2	0.026mm^2	0.026mm^2
Scoreboard (1800 bits)	0.019mm^2	0.154mm^2	0.077mm^2
Additional control	0.160mm^2	1.280mm^2	0.640mm^2
Total SM area increase		1.7mm^2	0.93mm^2
Percent SM area increase		11%	6%
Total GPU area increase		25.8mm^2	13.9mm^2
Percent GPU area increase		5%	2.5%

core area other processors devote to control [34, 4, 21, 20]. Based on these studies and the high datapath densities found in GPUs, we estimate that 1% of the Fermi SM area (16mm^2) is devoted to the datapath control logic that needs to be replicated in each slice. In total, we estimate that I-VWS adds approximately 11% and 6% to the area of an SM for 4-wide and 8-wide warps, respectively. For a Fermi-sized 15 SM GPU (529mm^2), I-VWS adds approximately 5% and 2.5% more area for 4-wide and 8-wide warps, respectively.

7. Related Work

This section first presents a quantitative comparison of I-VWS against two previously proposed techniques which address the effect of branch divergence on GPUs. It then presents a qualitative characterization of our work and prior art in the branch and memory divergence mitigation design space.

7.1. Quantitative Comparison

We compare I-VWS to two previously proposed divergence mitigation techniques: *Thread Block Compaction* (TBC) [12] and *Dynamic Warp Formation* (DWF) [11]. This data was collected using TBC's published simulation infrastructure [10], which is based on GPGPU-Sim 2.x [3]. We run TBC and DWF with the exact configuration specified in [10] and implement I-VWS with a warp size of 4 on top of their infrastructure. Figure 19 plots the result of this study on 5 divergent applications taken from the TBC simulation studies: raytracing (NVRT) [2], face detection (FCDT) [21], breadth first search (BFS) [8], merge sort (MGST) [8], and nearest neighbor (NNC) [8]. NVRT does not run when using DWF [12]. We chose these applications because their divergence behavior highlights the advantages of I-VWS. We also ran I-VWS on the rest of the applications in the TBC paper and observed little performance difference. On the five applications, I-VWS achieves an average 34% and 15% performance improvement over the baseline 32-wide warp and TBC respectively. The increased issue bandwidth and scheduling flexibility of I-VWS enables divergent applications to better utilize the GPU.

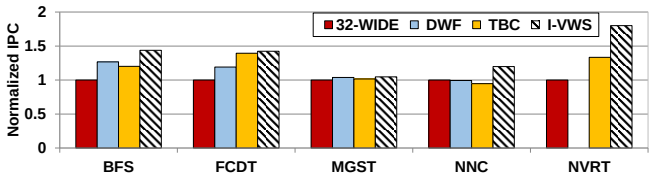


Figure 19: Performance (normalized to the 32-wide warp baseline) using the released TBC infrastructure [10].

7.2. Qualitative Comparison

Table 3 presents a qualitative characterization of previously proposed divergence mitigation techniques versus small warps and I-VWS. We classify previous work into three categories: (1) techniques that dynamically form or compact warps to regain SIMD efficiency [11, 12, 26, 29, 30]; (2) techniques that subdivide warps in the presence of memory and control flow divergence to expose more thread level parallelism [24, 33]; and (3) techniques that allow the interleaved execution of multiple branch paths within a warp by scheduling multiple paths from the control flow stack [31, 9].

The fundamental characteristic that sets I-VWS and the use of small warps apart from prior work is the ability to concurrently issue many more unique PCs by scaling and distributing instruction fetch bandwidth. Additionally, I-VWS can seamlessly adapt to memory latency divergence by breaking gangs. *Dynamic Warp Subdivision* (DWS) is also able to break lock-step warp execution on memory divergence [24]. However, DWS does this with a loss of SIMD efficiency and requires additional entries to be added to a centralized warp scheduler for each subdivided warp. In contrast, I-VWS allows smaller warps to continue independently, without losing SIMD efficiency; management of these smaller warps is distributed among many smaller scheduling entities.

While formation and compaction techniques increase SIMD efficiency, they pay for this increase with a reduction in available thread level parallelism, since forming and compacting warps decreases the number of schedulable entities. This decrease in TLP degrades the SM's ability to tolerate latencies. Conversely, subdivision and multipath stack techniques can increase latency tolerance but are limited by the number of scheduling entities in a monolithic warp scheduler and the number of concurrent entries on the call return stack.

All of the techniques perform well on convergent applications with the exception of small warps which suffer from memory convergence slip. We consider two classes of divergent applications: those that have a limited number of unique control flow (CF) paths and those that have many unique control flow paths. All of the proposed techniques perform well on applications that have a limited number of control flow paths. However, only smaller warps and I-VWS can maintain good performance when the number of unique control flow paths is high. Compaction and formation techniques require candidate threads to be executing the same instruction (PC). Subdivision and multipath stack techniques increase the number of schedulable entities in the SM, but do not improve lane utilization and become limited by the number of entries in a large, centralized structure when the number of unique control flow paths is large. Energy efficiency largely follows performance. On diverged applications with limited control flow paths, small warps and I-VWS lose some energy efficiency by fetching and decoding multiple times from smaller distributed fetch/decode structures, while prior work fetches one instruction from a larger structure. However, on divergent applications with many control flow paths, prior work inefficiently fetches one instruction repeatedly from a larger structure, while small warps and I-VWS distribute this repeated fetching over smaller structures. The smaller warps in I-VWS make it the only technique which improves both SIMD efficiency and thread level parallelism, while still exploiting the performance and energy efficiencies presented by convergent and warp-size insensitive code.

Table 3: Characterization of divergence mitigation techniques.

Characteristic	Form/ Compact	Subdivide	Multipath	Small Warps	I-VWS
# PCs per Cycle	1	1	1	Many	Many
Mem. Divergence Adaptive	No	Yes	No	Yes	Yes
Latency Tolerance	Low	Limited	Limited	High	High
Performance					
Convergent apps	High	High	High	Low	High
Divergent apps, limited CF	High	High	High	High	High
Divergent apps, many CF	Limited	Limited	Limited	High	High
Energy Efficiency					
Convergent apps	High	High	High	Low	High
Divergent apps, limited CF	High	High	High	Medium	Medium
Divergent apps, many CF	Low	Low	Low	High	High

7.3. Further Related Work

Other architectures have been proposed which support narrow SIMT execution, but lack the ganging features of I-VWS [14, 16, 18]. Simultaneous Branch and Warp Interleaving explores warp reconvergence and scheduling techniques using a modified SIMD pipeline that is able to execute two different instructions on the same cycle without scaling the fetch/decode bandwidth [5]. Meng et al. describe an approach called Robust SIMD that determines whether an application would be best served with a given SIMD width [23]. Wang et al. describe a "Multiple-SIMD, Multiple Data (MSMD)" architecture that supports flexible-sized warps with multiple simultaneous issue from different control-flow paths [35]. Lashgar et al. perform an investigation on the effects of warp size on performance [17].

8. Conclusion

This paper explores the design space of a GPU SM with the capability to natively issue from multiple execution paths in a single cycle. Our exploration concludes that convergent applications require threads to issue in lock-step to avoid detrimental memory system effects. We also find that the ability to execute many control flow paths at once vastly decreases a divergent application's sensitivity to reconvergence techniques.

We propose *Variable Warp Sizing* (VWS) which takes advantage of the many control flow paths in divergent applications to improve performance by 35% over a 32-wide machine at an estimated 5% area cost when using 4-wide warps. An 8-wide design point provides most of that performance benefit, while increasing area by only 2.5%. VWS evolves GPUs into a more approachable target for irregular applications by providing the TLP and SIMD efficiency benefits of small warps, while exploiting the regularity in many existing GPU applications to improve performance and energy efficiency.

Acknowledgements

The authors thank the anonymous reviewers for their insightful feedback. This work was supported by US Department of Energy contracts LLNS B599861 and LLNS B609478.

References

[1] T. M. Aamodt, "GPGPU-Sim 3.x Manual," http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual, University of British Columbia, 2012.

[2] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics (HPG)*, June 2009, pp. 145–149.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009, pp. 163–174.

[4] J. Baxter, "Open Source Hardware Development and the OpenRISC Project," Ph.D. dissertation, KTH Computer Science and Communication, 2011.

[5] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 49–60.

[6] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, November 2012, pp. 141–151.

[7] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-Based Barnes Hut N-Body Algorithm," in *GPU Computing Gems, Emerald Edition*, W. Hwu, Ed. Elsevier, 2011, pp. 75–92.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, October 2009, pp. 44–54.

[9] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2014, pp. 248–259.

[10] W. W. L. Fung, "Thread Block Compaction Simulation Infrastructure," http://www.ece.ubc.ca/~wwlfung/code/tbc-gpgpusim.tgz, University of British Columbia, 2012.

[11] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2007, pp. 407–420.

[12] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2011, pp. 25–36.

[13] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 235–246.

[14] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, September/October 2011.

[15] Khronos Group, "OpenCL," http://www.khronos.org/opencl/.

[16] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović, "The Vector-Thread Architecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 52–63.

[17] A. Lashgar, A. Baniasadi, and A. Khonsari, "Towards Green GPUs: Warp Size Impact Analysis," in *International Green Computing Conference (IGCC)*, June 2013, pp. 1–6.

[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," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 129–140.

[19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March/April 2008.

[20] A. Mahesri, "Tradeoffs in Designing Massively Parallel Accelerator Architectures," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2009.

[21] A. Mahesri, D. Johnson, N. Crago, and S. J. Patel, "Tradeoffs in Designing Accelerator Architectures for Visual Computing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 2008, pp. 164–175.

[22] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, August 2012, pp. 107–116.

[23] J. Meng, J. Sheaffer, and K. Skadron, "Robust SIMD: Dynamically Adapted SIMD Width and Multi-Threading Depth," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May 2012, pp. 107–118.

[24] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2010, pp. 235–246.

[25] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, August 2012, pp. 117–128.

[26] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2011, pp. 308–317.

[27] "NVIDIA CUDA C Programming Guide v4.2," NVIDIA, 2012.

[28] "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK-110," http://www.nvidia.ca/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, NVIDIA, 2012.

[29] M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-adequacy for Handling Control-divergence in GPGPU Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 61–71.

[30] M. Rhu and M. Erez, "Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2013, pp. 356–367.

[31] M. Rhu and M. Erez, "The Dual-Path Execution Model for Efficient GPU Control Flow," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2013, pp. 235–246.

[32] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2012, pp. 72–83.

[33] D. Tarjan, J. Meng, and K. Skadron, "Increasing Memory Miss Tolerance for SIMD Cores," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2009.

[34] "Diamond Standard 108Mini Controller: A Small, Low-Power, Cacheless RISC CPU," http://ip.cadence.com/uploads/pdf/108Mini.pdf, Tensilica.

[35] Y. Wang, S. Chen, J. Wan, J. Meng, K. Zhang, W. Liu, and X. Ning, "A Multiple SIMD, Multiple Data (MSMD) Architecture: Parallel Execution of Dynamic and Static SIMD Fragments," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, February 2013, pp. 603–614.

[36] S. Wilton and N. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.