Improving GPU Performance via Large Warps and Two-Level Warp Scheduling

Veynu Narasiman† Michael Shebanow‡ Chang Joo Lee¶
Rustam Miftakhutdinov† Onur Mutlu§ Yale N. Patt†

†The University of Texas at Austin
{narasima, rustam, patt}@hps.utexas.edu

‡Nvidia Corporation
mshebanow@nvidia.com

¶Intel Corporation
chang.joo.lee@intel.com

§Carnegie Mellon University
onur@cmu.edu

ABSTRACT

Due to their massive computational power, graphics processing units (GPUs) have become a popular platform for executing general purpose parallel applications. GPU programming models allow the programmer to create thousands of threads, each executing the same computing kernel. GPUs exploit this parallelism in two ways. First, threads are grouped into fixed-size SIMD batches known as *warps*, and second, many such warps are concurrently executed on a single GPU core. Despite these techniques, the computational resources on GPU cores are still underutilized, resulting in performance far short of what could be delivered. Two reasons for this are conditional branch instructions and stalls due to long latency operations.

To improve GPU performance, computational resources must be more effectively utilized. To accomplish this, we propose two independent ideas: the large warp microarchitecture and two-level warp scheduling. We show that when combined, our mechanisms improve performance by 19.1% over traditional GPU cores for a wide variety of general purpose parallel applications that heretofore have not been able to fully exploit the available resources of the GPU chip.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)
General Terms: Design, Performance.
Keywords: GPGPU, SIMD, Divergence, Warp Scheduling

1. INTRODUCTION

Graphics processing units (GPUs) have recently become a popular platform for executing general purpose parallel applications. Programming systems such as CUDA [20], ATI Stream Technology [1], and OpenCL [13] allow programmers to parallelize an application into thousands of threads each of which executes the same code. Previous work [23, 9] has shown that some applications experience an order of magnitude speedup when run on a GPU instead of a CPU. GPUs achieve such speedups because of the sheer amount of computational power they possess in relation to CPUs. They exploit this power by utilizing the *thread-level parallelism* (TLP) exposed by the programmer.

GPUs exploit TLP in two major ways. First, threads executing the same code are grouped into fixed sized batches known as *warps*.[1] These warps are executed on a processing core that employs a scalar front end (fetch and decode) and a SIMD (*single instruction, multiple data*) backend. The number of threads in a warp is usually equal to the SIMD width of the core so that a warp can execute an instruction for all its threads across the SIMD resources in parallel.

Second, GPUs concurrently execute many warps on a single core. For example, 32 warps, each with 32 threads, can all be assigned to execute on the same core. When one warp is stalled, other warps can execute which helps tolerate data dependencies, branch penalties, and long latency operations.

The Problem: Despite these techniques, the computational resources on a GPU core are still underutilized. For example, grouping threads into warps is efficient if those threads remain on the same dynamic execution path (i.e., same PC) throughout their execution. Although this holds true for graphics applications, many general purpose parallel applications exhibit more complex control flow behavior due to frequent conditional branches in the code. Conditional branch instructions can cause threads in a warp to take different dynamic execution paths, or *diverge*. Since existing GPU implementations allow a warp to have only one active PC at any given time, these implementations must execute each path sequentially. This leads to lower utilization of SIMD resources while warps are on divergent control-flow paths because the warp must execute with a fewer number of active threads than the SIMD width of the core.

Another example of unused computational resources occurs when a GPU core is unable to effectively hide the latency of long latency operations. The warp instruction fetch scheduling policy employed on a GPU core can significantly affect the core's ability to hide such latencies. For example, commonly-employed scheduling policies that give equal priority to each warp (i.e., round-robin scheduling) tend to result in all warps arriving at the same long latency operation at roughly the same time. Therefore, there are no other warps to execute to hide the latency. On the other hand, allowing warps to progress at very different rates can result in starvation and destroy the data locality among the warps. For example, data brought into the cache and row buffers

[1] Warp sizes for current NVIDIA [20] and ATI [1] GPUs are 32 and 64 respectively.

MICRO'11 December 3-7, 2011, Porto Alegre, Brazil
Copyright 2011 ACM 978-1-4503-1053-6/11/12 ...$10.00.

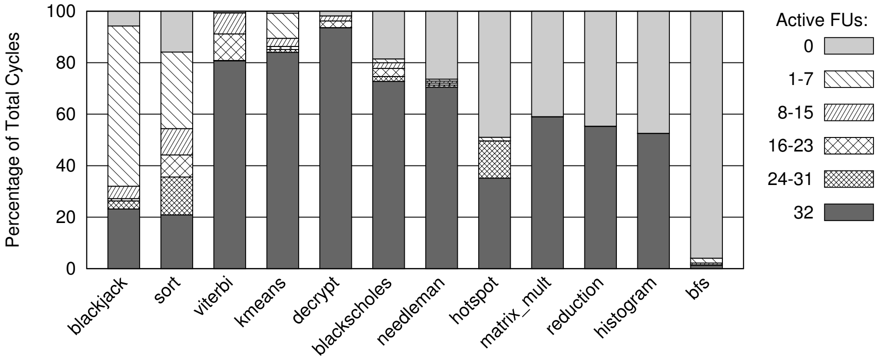


Figure 1: Computational resource utilization

opened by one warp are likely to be accessed again by other warps. However, allowing warps to progress very unevenly may destroy this locality.

Figure 1 illustrates the unused computational resources for a set of general purpose parallel benchmarks. Each benchmark is represented by a stacked bar indicating the percentage of cycles a certain number of the functional units (FUs) are active. In this experiment, SIMD width and warp size is 32, and 32 warps are concurrently executing on a single GPU core using round-robin scheduling. Branch divergence results in a reduction of the number of active threads in a warp which leads to underutilization of the computational resources. The leftmost benchmarks suffer from this problem as indicated by the large percentage of cycles where only a fraction of the FUs are active. On the other hand, the rightmost benchmarks suffer less from branch divergence but rather experience a significant fraction of cycles where none of the FUs are active (idle FU cycles). The main reason for these idle cycles is that all (or most) warps are stalled waiting on a long latency operation (e.g., a cache miss). Even with so many warps concurrently executing, several benchmarks show a significant fraction of idle cycles. For example, *bfs* spends approximately 95% of its execution time stalling.

Our goal is to improve GPU performance by better utilizing computational resources. To alleviate the performance penalty due to branch divergence, we propose the *large warp microarchitecture* (LWM). Existing GPU cores statically create many warps each with a modest number of threads (usually equal or close to the SIMD width of the core). Instead, we propose creating fewer but correspondingly larger warps (that have a significantly larger number of threads than the SIMD width of the core), and dynamically creating SIMD width sized sub-warps from the active threads in a large warp. The key insight is that even in the presence of branch divergence, there will likely be a large number of active threads in the large warp. These active threads can be dynamically grouped together into fully populated sub-warps that better utilize the SIMD resources on the core.

To reduce the number of idle FU cycles, we propose a novel two-level round-robin warp instruction fetch scheduling policy which can be applied on top of conventional GPU cores as well as the LWM. This policy splits all concurrently executing warps into fetch groups (e.g., 32 warps could be split up into 4 fetch groups of 8 warps each) and prioritizes warps from a single fetch group until they reach a stalling point (i.e., long latency operation). Then, the next fetch group is selected and the policy repeats. The scheduling policy within a fetch group is round-robin, and switching from one fetch group to another is also done in a round-robin fashion (hence two-level round-robin). The key insight is that each fetch group reaches a long latency instruction at different points in time; as such, when the warps in one fetch group are stalled, warps from another fetch group can be executing thereby effectively tolerating the latency. Since a fair round-robin policy is used at each level of scheduling, our two-level policy is still able to exploit the data locality among warps (which the conventional round-robin scheduling policy does very well). The overall result is reduced idle FU cycles leading to performance improvement.

We show that when combined, our mechanisms significantly improve computational resource utilization, resulting in 19.1% performance improvement over traditional GPU cores on a set of general purpose parallel applications.

2. BACKGROUND

We first describe the microarchitecture of a single GPU core.[2] Although we focus on a single GPU core, it should be noted that many such cores are replicated on the GPU.

2.1 GPU Core Pipeline

Figure 2 illustrates the baseline architecture of a single GPU core composed of a scalar front end (fetch, decode) and a SIMD backend. GPU programming models allow the programmer to create thousands of threads, each executing the same code. Before execution, threads are grouped into fixed size SIMD batches called warps. A warp contains threads with consecutive thread IDs and the number of threads is equal to the SIMD width of the core (N in Figure 2). Many warps (M in Figure 2 for a total of M × N threads) are assigned to execute concurrently on a single GPU core.

In the fetch stage, the scheduler selects a warp from the list of ready warps using a round-robin policy that gives equal priority to each warp [7, 16]. Associated with each warp is a warp ID, a bit vector called the active mask, and a single Program Counter (PC). Each bit in the active mask indicates whether the corresponding thread is active. When a warp is first created, all threads are active.[3]

Our baseline GPU core employs barrel processing [26, 24] such that once a warp is selected in the fetch stage, it cannot be selected again until that warp completes execution. After a warp is selected by the scheduler, the instruction cache

[2] Our term "GPU core" corresponds to a single Streaming Multiprocessor (SM) in NVIDIA's terminology [20].

[3] If the total number of threads is not a multiple of the warp size, one warp may be created without all threads active.

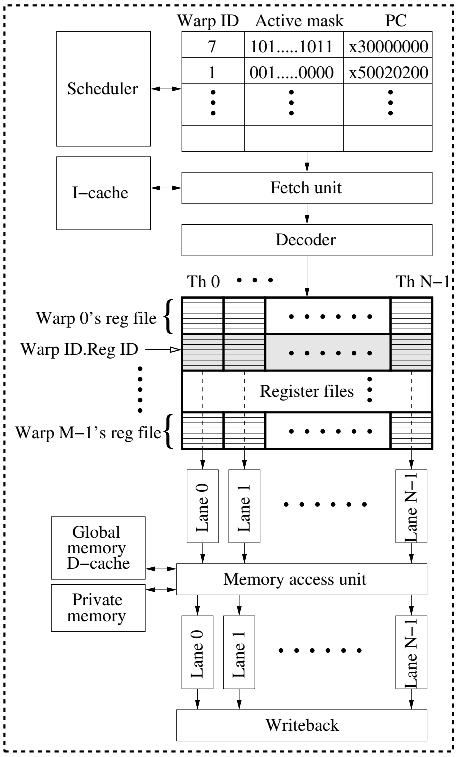


Figure 2: GPU core pipeline

is accessed at the PC of the warp and the fetched instruction is decoded, thereby completing the scalar portion of the pipeline. Next, register values for all threads in the warp are read in parallel from the register file indexed by warp ID and register ID as shown in Figure 2. These register values are then fed into the SIMD backend of the pipeline (i.e., the computational resources) and are processed in parallel across multiple SIMD lanes. Once a warp reaches the final stage of the pipeline, its PC and active mask are updated and the warp is again considered for scheduling.

2.2 Memory Model

Figure 2 also illustrates the memory model for the baseline GPU core. Data from global memory is cached on chip in the data cache. An entire cache line can be read (or written) in parallel in a single transaction. Therefore, the data accesses of all threads in a warp can be completed in a single transaction if all accesses map to the same line. If threads within a warp access different cache lines, the accesses will be serialized resulting in stalls in the pipeline. If one or more threads in the warp access a line not present in the cache, the entire warp stalls and is put aside, allowing other warps to flow through the pipeline.

Each thread also has access to a small amount of on-chip private memory which stores private data of each thread (i.e., local variables). This helps avoid costly accesses to global memory for applications where each thread's private data is too large for the register file. This on-chip memory is highly banked (one bank per SIMD lane) so that threads in a warp can read private data efficiently in parallel. This memory corresponds to private memory in OpenCL [13].

2.3 Conditional Branch Handling

Figure 3 illustrates the baseline branch handling mechanism currently employed by GPUs. In this example, there is only a single warp consisting of four threads, each of which is executing the same code whose control flow graph is shown in Figure 3(a). Since a warp can only have a single active PC at any given time, when branch divergence occurs, one path must be chosen first and the other is pushed on a divergence stack associated with the warp so that it can be executed later. The divergence stack is also used to bring the warp back together once the divergent paths have been executed and all threads have reached a control flow merge (CFM) point. A divergence stack entry consists of three fields: a re-convergence PC, an active mask, and an execute PC. Executing divergent paths serially but then re-converging at the CFM point is accomplished as follows:

1) When a warp encounters a divergent branch, push a *join* entry onto the divergence stack. This entry has both the re-convergence and execute PCs equal to the compiler identified control flow merge (CFM) point of the branch. The active mask field is set to the current active mask (i.e., the active mask when the branch instruction was executed). Next, one of the two divergent paths is selected to execute first and the current PC and active mask of the warp are updated accordingly. Lastly, another entry, the *divergent* entry, is pushed on the divergence stack. The execute PC and active mask of this entry correspond to the divergent path that was not selected to be executed first. The re-convergence PC is set equal to the CFM point of the divergent branch.

2) Each time a warp reaches the last pipeline stage, the warp's next PC is compared to the re-convergence PC at the top of the stack. If equal, the stack is popped, and the active mask and PC of the warp are updated with the active mask and execute PC fields of the popped entry.

Figures 3(b) through (e) show the state of a warp's PC, active mask, and divergence stack at relevant points in time as it executes the control flow graph of Figure 3(a). Inside each basic block of Figure 3(a) is a bit vector indicating whether the corresponding thread in the warp needs to execute the instructions in that basic block, i.e., the current active mask of the warp. The SIMD lanes are fully utilized as the instructions in block A execute but are underutilized when the divergent paths (blocks B and C) execute. Once all threads reach block D, the warp is restored to four active threads and execution once again proceeds efficiently. However, the under-utilization of SIMD resources before re-convergence results in performance degradation.

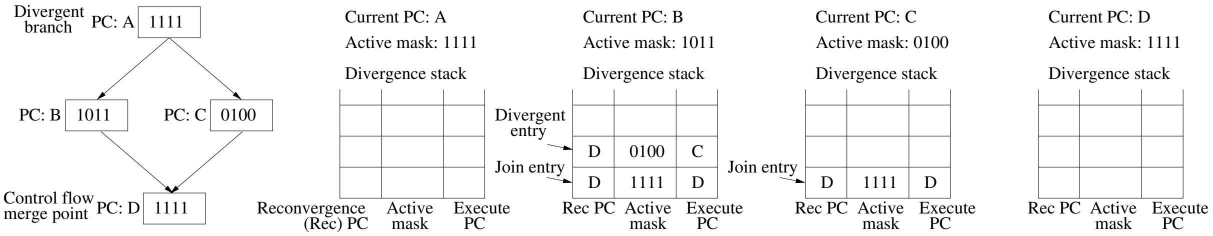
3. MECHANISM

In this section we describe our two new mechanisms: the Large Warp Microarchitecture, and the two-level round-robin warp scheduling policy. We first describe each mechanism, then discuss how the two are combined.

3.1 The Large Warp Microarchitecture

To alleviate the performance penalty due to branch divergence, we propose the *large warp microarchitecture* (LWM). While existing GPUs assign several warps to concurrently execute on the same GPU core, we propose having fewer but correspondingly larger warps. The total number of threads and the SIMD width of the core stay the same. The key benefit of having large warps is that fully populated sub-warps can be formed from the active threads in a large warp even in the presence of branch divergence.

A large warp is statically composed of consecutive threads and has a single PC. It also has an active mask organized as



(a) Control flow graph (b) Initial state (c) After executing A (d) After executing B (e) After executing C

Figure 3: Stack based re-convergence for baseline GPU cores

a two dimensional structure where the number of columns is equivalent to the SIMD width of the core. Figure 4 shows the organization of the active mask of a large warp of size K × N threads executing on a core with a SIMD width of N. Each cell in Figure 4 is a single bit indicating whether or not the corresponding thread is currently active. Notice that the actual storage cost does not change compared to the baseline. The baseline processor would have K separate N-bit wide active masks instead (i.e., K separate warps).

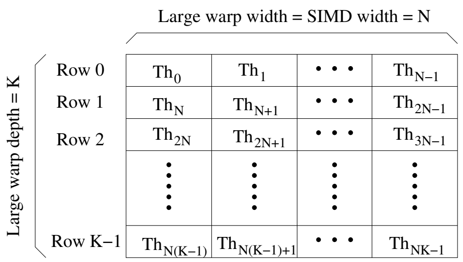


Figure 4: Large warp active mask

Once a large warp is selected in the fetch stage, the instruction cache is accessed at the PC of the large warp and the instruction is decoded in the following cycle just as in the baseline processor. In parallel with decode, SIMD-width sized sub-warps are created which then flow through the rest of the pipeline. When forming sub-warps, the goal is to pack as many active threads as possible into a sub-warp so as to best utilize the SIMD resources further down the pipeline. To accomplish this, specialized sub-warping logic examines the two dimensional active mask of the large warp and aims to pick one active thread from each column.

Sub-warp Creation: When determining how to pack active threads into a sub-warp, the design of the register file must be taken into consideration since it is imperative that the register values for a sub-warp can be sourced in parallel. Figure 5(a) shows the design of the register file for the baseline microarchitecture (no large warps). Since consecutive threads are statically grouped into warps and this assignment never changes, the register file can be conceptually designed as a very wide single banked structure indexed by warp ID concatenated with the register ID as shown in Figure 5(a).[4] However, having a single address decoder does not give enough flexibility for the LWM to pack threads into sub-warps. Ideally, we want to allow any set of active threads to be packed into a sub-warp. This would require the register file to have a number of ports equivalent to the SIMD width of the core. Such a design would require considerable increase in area and power. Therefore, we use a register file design similar to the one used by Jayasena et al. [10] and Fung et al. [7, 8] and shown in Figure 5(b). The register file is split up into separately indexable banks, one bank per SIMD lane. This design is cost-effective and provides much more flexibility in grouping active threads into sub-warps than the baseline register file. Using this design, we can now group threads into a sub-warp as long as they come from different columns in the large warp's active mask.

Figure 6 illustrates the dynamic creation of sub-warps from a large warp of 32 threads executing on a core with a SIMD width of four. Due to branch divergence, the large warp is shown with only a subset of its threads active. Each cycle, the hardware searches each column of the active mask in parallel for an active thread. The selected threads are grouped together into a sub-warp. Once an active thread is selected, the corresponding bit in the active mask is cleared. If there are still active threads remaining, a stall signal is sent to the fetch stage of the pipeline since the large warp has not yet been completely broken down into sub-warps. Once all bits in the active mask have been cleared, sub-warping for the current warp is complete and sub-warping for the next large warp (selected in the fetch stage) begins. Figure 6 illustrates how a large warp is dynamically broken down into four sub-warps over four successive cycles.

Note that the baseline processor would form eight different warps of four threads each rather than grouping all 32 threads into a large warp. Therefore, while the divergent code executes, SIMD resources will be underutilized since each warp contains fewer active threads than the SIMD width. However, with large warps, the inactive slots are filled with active threads during sub-warp creation. Therefore, only four efficiently packed sub-warps are created and SIMD resources are better utilized.

Barrel Processing: In the baseline GPU core, multiple warps are executed in a barrel processing fashion such that once a warp is selected by the scheduler in the fetch stage, it is not considered again for scheduling until the warp completes execution. For the large warp microarchitecture, we impose a similar, but slightly relaxed model. Once a large warp is selected, it is not reconsidered for scheduling until the first sub-warp completes execution. This re-fetching policy requires hardware interlocks to ensure that register dependencies are not violated. Rather than having full register dependency checking hardware for each thread, we employ a single bit per thread to ensure that if a thread has been packed into a sub-warp that has not completed execution, it will not be packed into another sub-warp (corresponding to the next instruction) until the previous sub-warp completes.

We use this relaxed re-fetching policy with one exception: conditional branch instructions. When a large warp executes a conditional branch, it is not refetched until *all* sub-warps complete. We do this since it is not known whether or not a

[4] Such large SRAMs cannot be built for timing/energy reasons [3], so even the baseline register file is slightly banked.

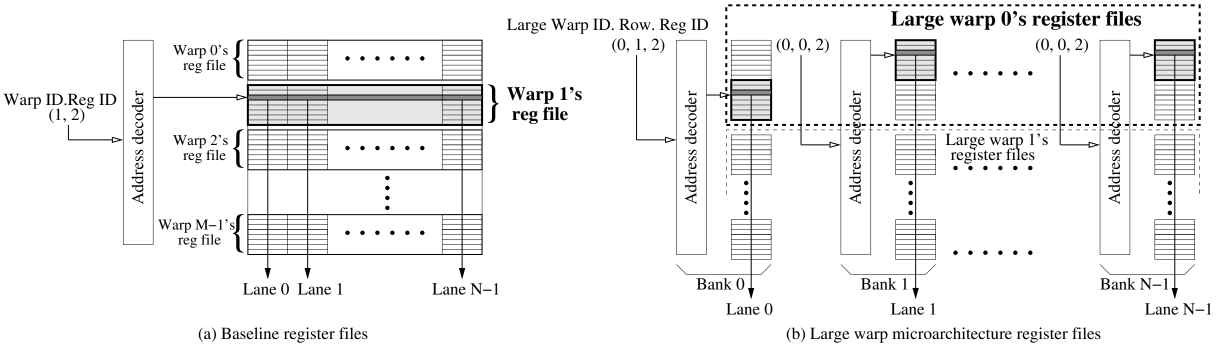
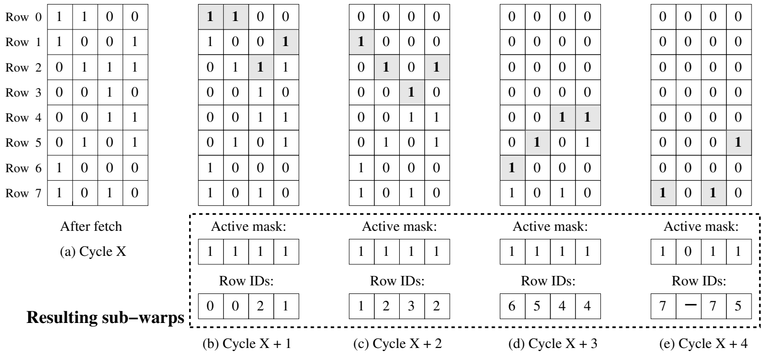


Figure 5: Large warp vs. baseline register file design



	(a) Cycle X	(b) Cycle X + 1	(c) Cycle X + 2	(d) Cycle X + 3	(e) Cycle X + 4
Row 0	1 1 0 0	1 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0
Row 1	1 0 0 1	1 0 0 1	1 0 0 0	0 0 0 0	0 0 0 0
Row 2	0 1 1 1	0 1 1 1	0 1 0 1	0 0 0 0	0 0 0 0
Row 3	0 0 1 0	0 0 1 0	0 0 1 0	0 0 0 0	0 0 0 0
Row 4	0 0 1 1	0 0 1 1	0 0 1 1	0 0 1 1	0 0 0 0
Row 5	0 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1	0 0 0 1
Row 6	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	0 0 0 0
Row 7	1 0 1 0	1 0 1 0	1 0 1 0	1 0 1 0	1 0 1 0
Active mask:		1 1 1 1	1 1 1 1	1 1 1 1	1 0 1 1
Row IDs:		0 0 2 1	1 2 3 2	6 5 4 4	7 − 7 5

Figure 6: Dynamic creation of sub-warps

large warp diverged until *all* sub-warps complete. As such, re-fetching a large warp after the first sub-warp completes requires speculation and complicates the implementation.

Divergence and Reconvergence: Large warps handle divergence and reconvergence much the same way that baseline warps do. However, as mentioned before, when a large warp executes a branch instruction, it is not known for sure whether or not the large warp diverged until the last sub-warp completes execution. Therefore, the new active mask and the active masks to be pushed on the divergence stack are buffered in temporary active mask buffers. Once all sub-warps complete execution, the current active mask and PC of the large warp are updated and divergence stack entries are pushed on the large warp's divergence stack (if in fact the large warp diverged). The divergence stack is popped just as described in the baseline processor in Section 2.3.

Unconditional Branch Optimization: When a warp in the baseline processor executes an unconditional branch instruction (i.e., a jump), only a single PC update is needed. The same is true for large warps, therefore there is no need to create multiple sub-warps when a large warp executes a jump. Thus, sub-warping for a large warp executing a jump completes in just one cycle, allowing sub-warping for the next large warp to begin sooner. Note that for large warps of 256 threads and a SIMD width of 32, this optimization saves up to 7 cycles by creating one sub-warp instead of 8.

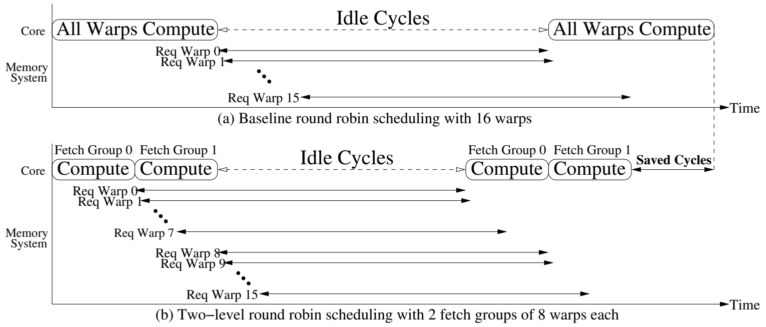
3.2 Two-level Warp Scheduling

GPU cores concurrently execute many warps on the same core which helps avoid stalls due to long latency operations. However, the warp instruction fetch scheduling policy employed on the GPU core can considerably affect the core's ability to hide long latencies. In this section, we propose a new two-level round-robin scheduling policy which more effectively hides long latencies and therefore reduces idle functional unit (FU) cycles. We first describe our new scheduling policy in the context of the baseline processor (not the LWM) and later describe how the two can be combined.

The baseline processor uses a round-robin warp instruction fetch policy giving equal priority to all concurrently executing warps [16, 7]. This policy results in warps progressing through the program at approximately the same rate which can be beneficial since warps tend to have a lot of data locality among them.[5] When one warp generates a memory request, other warps are likely to produce memory requests that map to that same row buffer. This row buffer locality can be exploited as long as the requests are generated close enough to each other in time. A fair round-robin policy allows this to happen whereas a scheduling policy that results in very uneven warp progression could destroy such locality. However, a pure round-robin scheduling policy also tends to make all warps arrive at the same long latency operation at roughly the same time. Since all (or most) of the warps are stalled, there are not enough active warps to hide the long latency resulting in several idle FU cycles.

To this end, we propose a two-level round-robin scheduling policy. The policy groups all concurrently executing warps into fixed size fetch groups. For example, 32 warps could be grouped into 4 fetch groups (with fetch group IDs 0-3) each with 8 warps. The scheduling policy selects a single fetch group to prioritize (let's say fetch group 0) and warps in that fetch group are given priority over warps in other fetch groups. More specifically, fetch group 0 is given the highest priority, fetch group 1 the next highest, and so on.

[5] GPU programmers are encouraged to make consecutive threads access consecutive memory locations so that memory accesses can be coalesced [20, 14], implying requests from different warps have significant spatial locality.



(a) Baseline round robin scheduling with 16 warps

(b) Two–level round robin scheduling with 2 fetch groups of 8 warps each

Figure 7: Baseline round-robin vs two-level round-robin scheduling

Warps within the same fetch group have equal priority and are scheduled in a round-robin fashion amongst each other. Once *all* warps in the highest prioritized fetch group are stalled on a long latency operation, a fetch group switch occurs giving fetch group 1 the highest priority, fetch group 2 the next highest, and so on. Fetch group 0, which used to have the highest priority, now has the least.

Note that the scheduling policy within a fetch group is round-robin, and switching fetch group priorities is also round-robin (hence two-level round-robin). Prioritizing fetch groups prevents all warps from stalling together. Instead, a smaller subset of warps (i.e., a fetch group) arrives at the stall together leaving other warps to execute while warps in one fetch group are stalled. Since both levels of scheduling are round-robin, row-buffer locality among warps remains high just as in conventional round-robin scheduling.

Figure 7 shows execution on a GPU core (a) with round-robin scheduling, and (b) with two-level scheduling. In this example, there are 16 total warps. With round-robin, all warps progress evenly through the compute phase of the program but then all stall waiting on data from memory. However, two-level scheduling with 2 fetch groups of 8 warps each reduces the number of idle cycles as shown in Figure 7(b). With two-level scheduling, warps in fetch group 0 proceed through the computation in half the time it took all 16 warps and therefore reach the stalling point sooner. Since all warps in fetch group 0 are stalled, a fetch group switch occurs and warps in fetch group 1 begin to execute the compute phase while requests created by fetch group 0 are serviced.

For two-level scheduling to be effective, the fetch group size must be set judiciously. The fetch group size should have enough warps to keep the pipeline busy in the absence of long latency operations. Recall from Section 2 that the baseline GPU core uses a barrel processing model where once a warp is selected in the fetch stage, it is not reconsidered for scheduling until it completes execution. Therefore, the minimum fetch group size is equal to the number of pipeline stages. Having too large a fetch group size limits the effectiveness of two-level scheduling for two reasons: 1) A larger fetch group takes longer to progress through computation than a smaller fetch group and therefore will not reach the stalling point as soon, and 2) A larger fetch group implies a greater number of warps stalling at the same time, leaving fewer warps to hide the latency. We evaluate the effect of fetch group size in our results section.

3.3 The LWM and Two-Level Scheduling

The LWM and two-level scheduling can be combined. In our results section, we show that when the LWM is evaluated independently, the best performing large warp size is 256 threads. Likewise, the best performing fetch group size for two-level scheduling is 8 regular-sized warps (i.e., 256 total threads since there are 32 threads per warp in the baseline). One would think that when the two mechanisms are combined, we should use a large warp size of 256 threads, and the fetch group size should be a single large warp.

However, the combination of the large warp re-fetch policy for branch instructions (i.e., waiting for all sub-warps to complete) and two-level scheduling with a fetch group size of one can be problematic and limit the effectiveness of the combination. For example, consider an application that has no stalls after a short warmup period. With no stalls, two-level scheduling continues prioritizing a single large warp until the entire program completes. Only then will a fetch group switch occur. This will ultimately result in a single large warp having to execute the entire program with no other large warps active (since they all finished executing the program). Having only one large warp active for such a long time is problematic since every time a conditional branch instruction is fetched, the large warp must wait until all sub-warps complete before being re-fetched thereby introducing several bubbles in the pipeline.

To alleviate this, we implement a two-level scheduling timeout rule whereby if a single large warp is prioritized for more than *timeout* instructions, we preemptively invoke a fetch group switch. Note that for most applications, the timeout is never invoked. However, for branch-intensive applications with few long latency stalls (e.g., the *blackjack* benchmark used in our evaluation), this rule helps significantly by bounding the time only a single large warp is active. We determine empirically that 32K instructions works well for the timeout period.

3.4 Hardware Cost

Most of the hardware cost for the LWM comes from restructuring the register file. As shown in Section 3.1, instead of a single address decoder, our mechanism requires a separate address decoder per SIMD lane. Previous work [10, 7, 8] estimates that such a design results in little die area increase. Jayasena et al. [10] propose stream register files with indexed access which require dedicated row decoders for each bank instead of a single shared decoder. They show that this results in an 11% to 18% increase in register file area which corresponds to a 1.5% to 3% increase in chip area of the Imagine processor [12]. Fung et al. [8] show similar results with an estimated 18.7% increase in register file area, corresponding to 2.5% of GPU area.

In addition to the register file overhead, there are a few

Scalar front end	1-wide fetch and decode stages, round-robin warp scheduling 4KB single-ported instruction cache
SIMD back end	In order, 5 stages, 32 parallel SIMD lanes
Register file and on-chip memories	64KB register file (16 32-bit registers per thread, 1024 concurrent threads) 128KB, 4-way, single cycle data cache, 1 read, 1 write port, 128-byte line size 128KB, 32-banked private memory (128 bytes per thread)
Memory system	Open-row, first-come first-serve scheduling policy, 8 banks, 4KB row buffer 100-cycle row-hit, 300-cycle row-conflict latency, 32 GB/s memory bandwidth

Table 1: Baseline GPU core and memory configuration

Benchmark	Description	Input Set
blackjack	Simulation of blackjack card game to compute house edge	Standard 52-card deck per thread
sort	Parallel bucket sort of a list of integers	1M random integers
viterbi	Viterbi algorithm for decoding convolutional codes	4M convolutionally encoded bits
kmeans	Partitioning based clustering algorithm	16K 1-dimensional 8-bit data points
decrypt	Advanced Encryption Standard decryption algorithm	256K AES encrypted bytes
blackscholes	Call/put option pricing using blackscholes equation	Initial values for 512K options
needleman	Calculate optimal alignment for 2 DNA sequences	2 DNA Sequences of length 2048
hotspot	Processor temperature simulation	512x512 grid of initial values
matrix_mult	Classic matrix multiplication kernel	2 256x256 integer matrices
reduction	Reduce input values into single sum	32M random boolean values
histogram	Binning ASCII characters into a histogram	16M ASCII characters
bfs	Breadth first search graph traversal	1 million node arbitrary graph

Table 2: Benchmarks

extra storage structures required by the LWM not present in the baseline GPU core. As explained in Section 3.1, sub-warp creation is done in parallel with decode by searching the columns of the two dimensional active mask of a large warp and clearing the bits corresponding to the selected threads. We cannot clear the bits in the large warp's actual active mask and therefore a copy must be made before the large warp can be broken down into sub-warps. For large warps of size 256 threads, this corresponds to 256 bits of storage. In addition, as explained in Section 3.1, the large warp microarchitecture uses temporary active mask buffers while executing branch instructions. Since a temporary buffer is required for each path of the divergent branch, this corresponds to 512 bits of storage. Lastly, the LWM requires extra bits and some simple logic for dependency handling as explained in Section 3.1. This additional storage amounts to just a single bit per thread. The total additional storage cost is 256+512+1024 bits (i.e., 224 bytes).

Two-level warp scheduling does not require any additional storage cost. The only change is a simple logic block in the fetch stage implementing two-level round-robin scheduling.

4. METHODOLOGY

We use a cycle accurate simulator that simulates parallel x86 threads, each executing the same compute kernel. In our results, we simulate a single GPU core concurrently executing 1024 threads. Table 1 presents the system parameters used in our simulations for the baseline processor.

Since x86 does not have instructions to aid with branch divergence/re-convergence of parallel threads like GPU ISAs do [21], we created instrumentation tools to identify conditional branch instructions and their control flow merge points. We used a similar procedure to identify barrier synchronization points since x86 does not support single instruction barrier synchronization present in GPU ISAs [21].

We created parallel applications adapted from existing benchmark suites including Rodinia [5], MineBench [18], and NVIDIA's CUDA SDK [19] in addition to creating one of our own (blackjack). Each benchmark was parallelized using POSIX threads (Pthreads) and compiled with Intel's ICC compiler. We optimized each benchmark for GPU execution using principles found in [23] and [14]. Each benchmark runs to completion and consists of 100-200 million dynamic instructions across all 1024 threads. Table 2 lists the benchmarks (along with input set) used in this study.

The metric we use to compare performance is retired *instructions per cycle* (IPC). Note that when a warp (or a sub-warp) executes an instruction, we treat each active thread in the warp as executing a single instruction. Therefore, if the warp (or sub-warp) size is 32, the maximum IPC is 32.

5. RESULTS

In this section we provide the overall results of the large warp microarchitecture and two-level scheduling. We also compare our work to recent state of the art work in divergent control flow handling on GPUs, Thread Block Compaction [6] (TBC). TBC groups multiple regular-sized warps into a block and synchronizes them at every branch instruction. Special hardware is used to dynamically create new warps (i.e., compact into fewer warps) based on the active mask of all warps in the block after the divergent branch.

5.1 Overall IPC Results

Figures 8 and 9 show IPC and functional unit utilization for the baseline (32 warps of 32 threads each, round-robin scheduling), Thread Block Compaction (TBC), large warp microarchitecture (LWM), two-level scheduling (2Lev), and large warp microarchitecture combined with two-level scheduling (LWM+2Lev). Note that the SIMD width (32) and total thread count (1024) is the same for each configuration. For TBC, we used a block size of 256 threads and sticky round robin (SRR) scheduling, which performed slightly better than round-robin in our evaluation. For the LWM, we created 4 large warps of 256 threads each. For two-level scheduling only, we set the fetch group size to 8 (i.e., 4 fetch groups, each consisting of 8 warps). For the combination of LWM and two-level scheduling, we again formed 4 large warps of 256 threads each and set the fetch group size to 1 (i.e., 4 fetch groups, each consisting of 1 large warp).

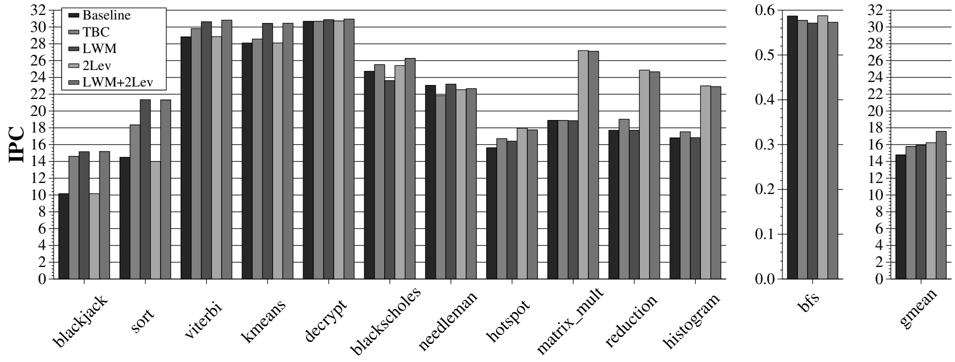


Figure 8: Performance

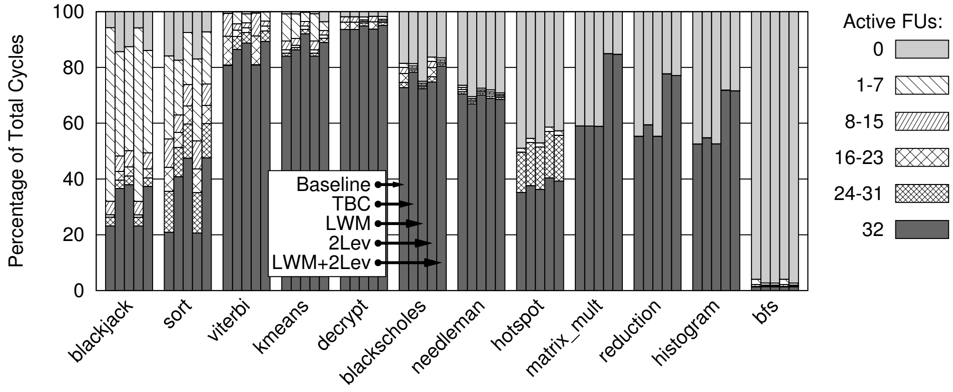


Figure 9: Functional unit utilization

As expected, the LWM (third bar) significantly improves performance for branch-intensive applications (the leftmost 4 benchmarks), whereas two-level scheduling (fourth bar) does not provide much benefit compared to the baseline for these applications. The reason for this is that these benchmarks make very good use of the on chip data cache and private memory and therefore do not suffer much from long latencies. However, they do contain frequent divergent branches which is the main reason for performance degradation for these applications. This is justified by looking at the computational resource utilization for these applications in Figure 9. There are relatively few idle cycles (0 active FUs) for these benchmarks even in the baseline architecture, however they do have a significant number of cycles where only a small portion of the FUs are active. The LWM improves this by efficiently packing active threads into sub-warps, thereby increasing SIMD utilization and improving performance. TBC also improves performance for these applications but not as much as the LWM does. There are two main reasons for LWM's edge. First, the LWM benefits from the optimization for unconditional branch instructions discussed in Section 3.1. TBC cannot benefit from this optimization since it does not keep a large number of threads together between branch instructions. After compacting threads after a branch, warps are treated as individual scheduling units and therefore this optimization does not apply. Second, when a large warp executes a predicated instruction (e.g., cmov in x86), it creates efficiently packed sub-warps based on the predicated active mask, whereas TBC does not efficiently execute predicated code since it only performs compaction after divergent branch instructions.

The rightmost benchmarks show the opposite behavior. These benchmarks suffer from long latency stalls and therefore the LWM provides no benefit but two-level scheduling effectively reduces idle FU cycles as shown in Figure 9. Two-level scheduling results in only a subset of warps (i.e., fetch group) stalling at the same time, allowing warps in different fetch groups to effectively hide the latency. In addition, row buffer locality remains high (hit rate within 1.7% of traditional round-robin on average) since at each level of scheduing a round-robin policy is used. TBC, even with SRR scheduling, does not provide as much benefit due to the synchronization required at each branch instruction.

In summary, LWM alone improves performance by 7.9%, two-level scheduling alone improves performance by 9.9%, and when our two mechanisms are combined, the benefits of each are preserved resulting in 19.1% performance improvement over the baseline and 11.5% over TBC.

5.2 Large Warp Microarchitecture Analysis

In this section we provide results relating to the LWM and use round-robin scheduling among the large warps (not two-level) in order to isolate the effects of the LWM.

Varying the Large Warp Size: We vary the warp size from the baseline of 32 threads per warp to a maximum of 512.[6] As seen in Figure 10, increasing warp size improves performance up to 256 threads. Larger warp sizes give more potential for sub-warping logic to create efficiently packed sub-warps and therefore in general, are beneficial. However, extremely large warp sizes can also be harmful. Having too large a warp size is inefficient for cases where most of the threads have reached the reconvergence point of a divergent branch and are waiting on just a few threads to arrive so

[6] All 1024 threads in a warp is not evaluated since the branch re-fetch policy would result in inefficient use of the pipeline.

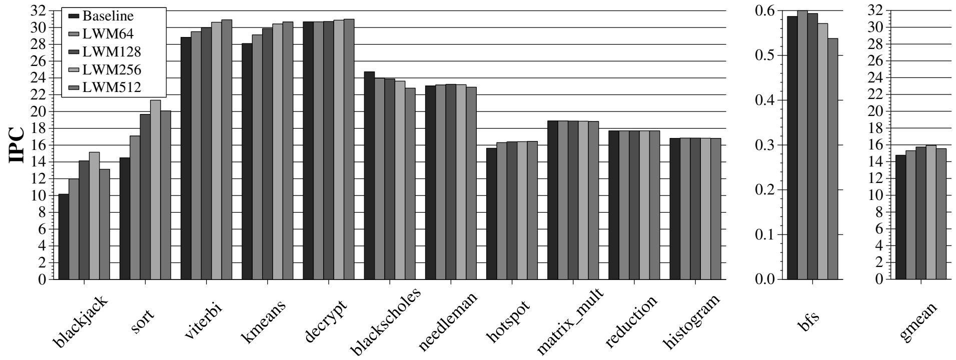


Figure 10: Effect of large warp size

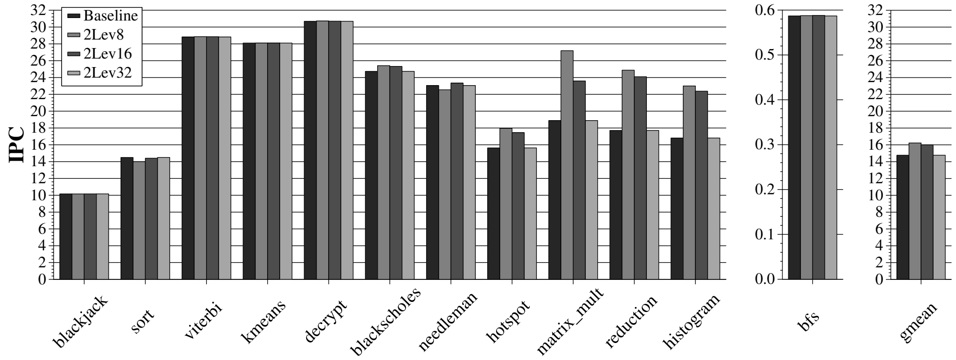


Figure 11: Effect of fetch group size on two-level scheduling

that reconvergence can be done and execution can continue. For example, if each thread of a large warp executes a loop for a different number of iterations (i.e., the number of iterations is data dependent), at some point almost all of the threads will have exited the loop but will have to wait and sit idle (on the divergence stack) while the last few threads finally exit. Only then can the reconvergence stack be popped and execution of the reconverged large warp continue. This problem becomes more severe as large warp size increases since a greater number threads will be sitting idle on the divergence stack until the final thread exits the loop. We find this behavior prominent in *blackjack* and *sort*, explaining why 256 threads performs better than 512. On the other hand, a large warp size of 512 slightly outperforms 256 for a few of the benchmarks which don't exhibit the loop divergence discussed above but rather have more regular if-then-else control flow constructs. Having too small a large warp size (64, 128 threads) does not provide enough opportunity for sub-warping logic to efficiently pack threads into sub-warps. Overall, we find a large warp size of 256 balances these tradeoffs well and provides the best average performance.

5.3 Analysis of Two-level Scheduling

In this section, we apply two-level scheduling on top of the baseline microarchitecture (not LWM) and vary the fetch group size. Since there are 32 total warps concurrently executing on our baseline 7-stage GPU core, we use fetch group sizes of 8, 16 and 32 warps. In our notation, "2Lev8" stands for two-level scheduling, with a fetch group size of 8 (i.e., 4 fetch groups each consisting of 8 warps). Figure 11 shows the IPC as we vary the fetch group size.

For the benchmarks on the left, there is no variation since these benchmarks have very few idle cycles even with the baseline round-robin policy. However, the rightmost benchmarks do illustrate that a fetch group size of 8 warps works best. Recall from Section 3.2 that the fetch group size should have just enough warps to keep the pipeline busy, but that too many warps in a fetch group limits the effectiveness of two-level scheduling. A fetch group size of 16 (half of all the warps) still improves performance over the baseline round-robin policy but not as much as 8. 16 warps is more than necessary to keep the pipeline busy and results in a larger subset of warps arriving at the long latency operation together and therefore is unable to hide latencies as well as 8 warps. Note that a fetch group size of 32 (i.e., all 32 warps in a single fetch group) is by definition equivalent to the baseline round-robin policy and therefore performs identically to it. Having only a single fetch group removes one of the levels of scheduling (the prioritization of the fetch groups) and therefore all the benefits of two-level scheduling are lost.

6. RELATED WORK

Conditional Execution on SIMD Processors: Using bit masks to execute conditional code in SIMD processors is an old concept. The Illiac IV [4] had a *mode bit* per Processing Element (PE) which turned on or off a PE during execution. CRAY-1 [22] had a *vector mask* register which was used to vectorize loops with if/else statements. These bit masks are akin to the active mask on GPU cores.

Smith et al. [25] introduced the concept of *density-time execution* whereby the time taken to execute a masked vector instruction is a function of the number of true values in the mask. False values in the vector mask register are skipped, thereby reducing the number of cycles it takes to execute the vector instruction. Rather than skipping false values, our approach finds active operations from threads in a large warp to fill the holes caused by branch divergence.

Kapasi et al. [11] introduced *conditional streams*, which allow stream processors to filter an input stream before it is processed. However, this mechanism requires 1) communication between different SIMD lanes, and 2) effort from the programmer to declare conditional streams and implement new kernels to perform the filtering. In contrast, our approach 1) does not require communication between SIMD lanes and 2) is a pure hardware mechanism and therefore does not require any programmer effort.

Krashinsky et al. [15] proposed the Vector-Thread architecture (VT), which employs a control processor and a vector of virtual processors (VPs). The control processor uses vector-fetch commands to broadcast the same instruction to all the VPs. However, if divergence occurs, each VP also has the ability to direct its own control flow with thread-fetch commands. In this sense, the architecture is not strictly SIMD. In contrast, the LWM is strictly SIMD and tolerates branch divergence by dynamically breaking down large warps into efficiently packed sub-warps.

Fung et al. [7, 8] were the first to propose the idea of combining threads from different warps to address underutilized SIMD resources due to branch divergence on GPU cores. This work has been superseded by Thread Block Compaction [6] which we compare to in our results section.

Meng et al. [17] proposed Dynamic Warp Subdivision (DWS) whereby upon divergence, two warp-splits are formed which can be scheduled independently. Although this does not increase SIMD resource utilization, it may increase memory-level parallelism since both sides of a divergent branch are executed concurrently. As such, DWS is orthogonal to the LWM and can be employed on top of the LWM by splitting up a large warp upon branch divergence.

Fetch Scheduling: Many previous proposals analyzed and proposed scheduling policies for threads on MT or SMT cores [2, 27]. However, none of these policies were designed for scheduling warps on GPUs. GPU scheduling is unique in that warps tend to have much data locality among them. Also, GPUs support many more warp contexts compared to MT and SMT cores and allow zero-cycle context switching among all concurrently executing warps.

Lakshminarayana et al. [16] evaluate several possible fetch scheduling policies for GPUs. However, the policies they evaluate do not include two-level scheduling. Furthermore, most of the scheduling policies they evaluate result in warps progressing uniformly through the program (similar to pure round-robin). In contrast, our two-level policy allows warps to arrive at a long latency instruction slightly apart from each other in time thereby effectively hiding long latencies.

7. SUMMARY AND CONCLUSION

In this paper, we propose two mechanisms to improve GPU performance by better utilizing computational resources on GPUs. To alleviate the performance penalty due to branch divergence, we propose the large warp microarchitecture. While existing GPU cores concurrently execute multiple SIMD-width sized warps, we propose forming fewer but correspondingly larger warps and dynamically creating efficiently packed SIMD-width sized sub-warps from the active threads in a large warp. This leads to improved SIMD resource utilization in the presence of branch divergence. To improve long latency tolerance, we propose a two-level round-robin warp scheduling policy. This policy prevents all warps from arriving at the same long latency operation at the same time, thereby reducing idle execution cycles.

Our experimental evaluations show that when combined, our mechanisms improve performance by 19.1% on average for a wide variety of general purpose parallel applications. We believe that our mechanisms increase the scope of general purpose parallel applications that can achieve significant speedup when executed on a GPU.

ACKNOWLEDGMENTS

Many thanks to José A. Joao, other HPS members and the anonymous reviewers for their comments and suggestions. Veynu Narasiman was supported by an NVIDIA PhD Fellowship during the early part of this work. We also acknowledge the support of the Cockrell Foundation and Intel.

REFERENCES

[1] Advanced Micro Devices, Inc. *ATI Stream Technology.* http://www.amd.com/stream.

[2] A. Agarwal et al. April: a processor architecture for multiprocessing. In *ISCA-17*, 1990.

[3] B. Amrutur and M. Horowitz. Speed and power scaling of SRAMs. *IEEE JSCC*, 35(2):175–185, Feb. 2000.

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

[5] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[6] W. W. L. Fung and T. Aamodt. Thread block compaction for efficient simt control flow. In *HPCA-17*, 2011.

[7] W. W. L. Fung et al. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO-40*, 2007.

[8] W. W. L. Fung et al. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM TACO*, 6(2):1–37, June 2009.

[9] W.-M. Hwu et al. Compute unified device architecture application suitability. *Computing in Science Engineering*, may-jun 2009.

[10] N. Jayasena et al. Stream register files with indexed access. In *HPCA-10*, 2004.

[11] U. Kapasi et al. Efficient conditional operations for data-parallel architectures. In *MICRO-33*, 2000.

[12] B. Khailany et al. Vlsi design and verification of the imagine processor. In *ICCD*, 2002.

[13] Khronos Group. *OpenCL*. http://www.khronos.org/opencl.

[14] D. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2010.

[15] R. Krashinsky et al. The vector-thread architecture. In *ISCA-31*, 2004.

[16] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010.

[17] J. Meng et al. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA-37*, 2010.

[18] R. Narayanan et al. MineBench: A benchmark suite for data mining workloads. In *IISWC*, 2006.

[19] NVIDIA. *CUDA GPU Computing SDK*. http://developer.nvidia.com/gpu-computing-sdk.

[20] NVIDIA. *CUDA Programming Guide Version 3.0*, 2010.

[21] NVIDIA. *PTX ISA Version 2.0*, 2010.

[22] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[23] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[24] B. J. Smith. A pipelined shared resource MIMD computer. In *ICPP*, 1978.

[25] J. E. Smith et al. Vector instruction set support for conditional operations. In *ISCA-27*, 2000.

[26] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS*, 1965.

[27] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *MICRO-34*, 2001.