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

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

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

†NVIDIA Research
Santa Clara, CA 95050

‡Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890

TR-HPS-2010-006
December 2010

This page is intentionally left blank.

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

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, before execution, 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 performance of GPUs, the computational resources available must be more effectively utilized. To accomplish this, this paper proposes two independent ideas: the large warp microarchitecture and a two-level warp scheduling policy. We show that when combined, our mechanisms improve performance by 17.0% over traditional GPU cores for a wide variety of general purpose parallel applications that heretofore have not been able to exploit the available resources of the GPU chip.

1. Introduction

Over the past few years, *graphics processing units* (GPUs) have become a popular platform for executing general purpose parallel applications. Programming systems such as CUDA [23], ATI Stream Technology [1], and OpenCL [14] allow programmers to parallelize an application into thousands of threads each of which executes the same code. These threads are executed by a GPU in parallel thereby significantly reducing the execution time of the application. Previous work [26, 10] has shown that some applications experience an order of magnitude speedup when run on a GPU instead of a CPU. GPUs are able to 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.

GPU cores take advantage of TLP in two major ways. First, before execution, GPUs statically group threads executing the same code 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. In a given cycle, each thread belonging to the same warp executes the same instruction, yet on a different piece of data. This style of processing amortizes the cost of fetch and decode across all threads in a warp thereby allowing more chip area to be dedicated to data processing (i.e., computational resources) rather than control [23].

Second, GPU cores concurrently execute multiple warps on a single core. For example, 32 warps, each consisting of 32 threads (for a total of 1024 threads), can all be assigned to execute on the same core. When

[1] Warp sizes for current NVIDIA [23] and ATI [1] GPUs are 32 and 64 respectively.

one warp is stalled, other warps can continue to execute which helps tolerate data dependencies, branch penalties, and long latency operations, especially memory requests that miss in the cache.

The Problem: Underutilized Computational Resources Despite these techniques, the computational resources on a GPU core are still underutilized. For example, grouping threads into warps is only efficient if those threads remain on the same dynamic execution path (i.e., same PC) throughout their execution. Although this holds true for many graphics applications, some general purpose parallel applications exhibit more complex control flow behavior among the parallel threads 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. First, the warp executes the threads that follow the taken path of the branch (the not taken threads are masked off). Then the warp executes the not taken path threads (masking off the taken path threads). 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. This loss of efficiency continues until the divergent paths finish and a control flow merge point is reached. At this time, the warp is brought back to its original active thread count (i.e., the active thread count before the divergent branch instruction) and execution proceeds efficiently.

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 the 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 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 on GPU cores 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 computational resources (i.e., functional units) are active. For this experiment, both the SIMD width and warp size is set to 32, and 32 warps are concurrently executing on the same core using a round-robin scheduling policy. As previously mentioned, 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 five benchmarks suffer from this problem indicated by the fact that there is a 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., waiting on data from memory). Even with so many warps concurrently executing (32 warps for this experiment), several benchmarks show a significant fraction of idle cycles. For example, the bfs benchmark spends approximately 95% of its execution time stalling.

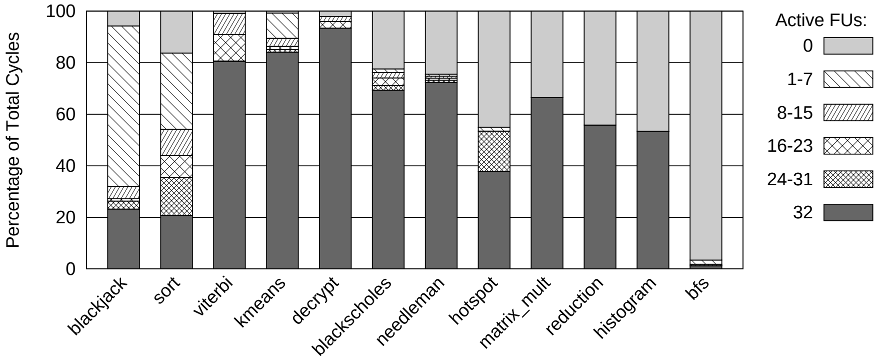


Figure 1. Computational resource utilization, SIMD width and warp size is 32

Our Goal is to improve application performance on GPUs by better utilizing computational resources. To this end, we propose two mechanisms each of which aims to reduce one of the two major causes of resource underutilization: branch divergence and long latency operations.

Key Ideas 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 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 can 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 core architectures as well as the LWM. This policy assigns a set of warps into a fetch group (e.g., 32 warps could be split up into 4 fetch groups of 8 warps each). The scheduling policy selects a fetch group to prioritize and schedules warps from *only* that fetch group in a round-robin fashion until no warp from that fetch group can be scheduled (i.e., all of the warps in the fetch group are stalled). At this point, the next fetch group is selected and the policy repeats. Note that the scheduling policy within a fetch group is round-robin, and switching from one fetch group to another is also done in a 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 between warps (which the conventional round-robin scheduling policy does very well). The overall result is reduced idle FU cycles leading to performance improvement.

Contributions: In this paper, we make the following contributions:

1. We propose a novel microarchitecture to mitigate the performance loss due to branch divergence on GPU cores. Our proposal introduces large warps and includes a dynamic mechanism to break down large warps into fully populated (or close to fully populated) sub-warps.

2. We propose a new two-level warp instruction fetch scheduling policy and show that such a policy significantly reduces the number of idle execution cycles due to long latency instructions. We show that this policy can be used with either an existing GPU microarchitecture or with the large warp microarchitecture.

3. We show that our two proposals can be combined. Together, they significantly improve performance (by 17.0% on average) over a wide range of general purpose parallel applications by more efficiently utilizing the massive computational resources found on GPU cores.

2. Background

We first describe in detail the microarchitecture of a single GPU core.[2] Although we discuss and evaluate only 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 and decode) and a SIMD backend. GPU programming models allow the programmer to create thousands of threads, each executing the same code. Before execution, those threads are grouped into fixed size SIMD batches called warps. Each warp contains threads with consecutive thread IDs and the number of threads in the warp is equal to the SIMD width of the core (N in Figure 2). Many warps (M warps 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. The baseline fetch scheduling policy uses a round-robin scheduler giving equal priority to each warp [9, 17]. Associated with each warp is a warp ID, a bit vector called the active mask, and a single Program Counter (PC). The active mask indicates whether or not the corresponding thread in a warp is currently active. When a warp is originally created, all of its threads are active.[3] However, branch divergence can cause threads within a warp to become inactive.

Our baseline processor imposes a strict barrel processing model[30, 28] where once a warp is selected in the fetch stage, it cannot be selected again until the warp completes execution. After a warp is selected by

[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, then a single warp may be created without all threads active.

the scheduler, the instruction cache is accessed at the PC of the warp and the instruction is decoded thereby completing the scalar portion of the pipeline. Next, the register values for all threads in the warp are read in parallel from the register file which is 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 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

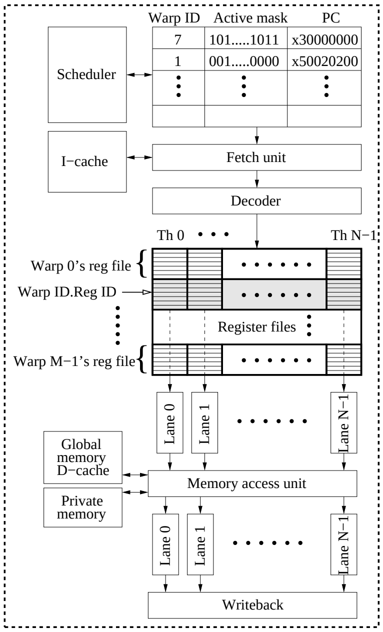


Figure 2. GPU core pipeline

Figure 2 also illustrates the memory model for the baseline GPU core. All threads have access to global memory and data from global memory is cached on chip in the global memory D-cache. An entire cache line can be read (or written) in parallel in a single transaction and therefore a warp accessing global memory can be satisfied in a single transaction if all threads in the warp access data in the same cache line. If the threads within a warp access different cache lines, those accesses will be serialized resulting in stalls in the pipeline. This is similar to the scatter/gather support in Intel's Larrabee microarchitecture [27]. 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 while the data is fetched from global memory.

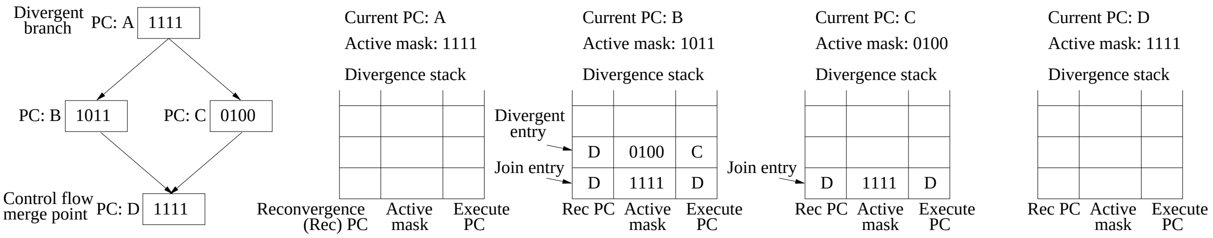
In addition to the global memory data cache, each thread also has access to a small amount of on-chip private memory which stores private data of each thread (i.e., local variables). This helps avoid costly accesses to main memory for applications where each thread's private data is too large for the 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 [14].

2.3. Conditional Branch Handling

Figure 3, adapted from Fung et al. [9, 8], illustrates the baseline branch handling mechanism currently employed by GPU cores. In this example, there is only a single warp consisting of four threads, each of which is executing the same static 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 the divergent paths serially but then re-converging at the CFM point can be accomplished by doing the following:

1) When the branch outcomes for the threads in the warp are not the same (i.e., a divergent branch), push a *join* entry onto the divergence stack. The join entry has both the re-convergence PC and execute PC fields 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 for this entry is set equal to the CFM point of the divergent branch.

2) When a warp reaches the last stage of the pipeline, its re-convergence stack is accessed to see if the next PC of the warp is equal to the re-convergence PC at the top of the stack. If so, the entry is popped, and the active mask and execute PC fields of the entry become the current active mask and PC of the warp.



(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

Figures 3(b) through (e) show the state of the current PC, the current active mask, and the divergence stack for a warp 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 or not 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 as the divergent paths (blocks B and C) execute. Once all threads reach block D, the warp is restored to having four active threads and execution once again proceeds efficiently. However, the under-utilization of SIMD resources before re-convergence at the control flow merge point can lead to significant performance degradation.

3. Solution: Large Warp Microarchitecture and Two-level Warp Scheduling

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

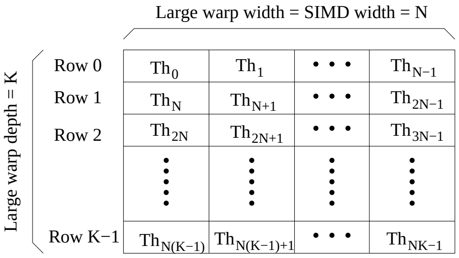

Large warp width = SIMD width = N
Large warp depth = K
Row 0
Row 1
Row 2
Row K−1
Th_0 Th_1 • • • Th_{N-1}
Th_N Th_{N+1} • • • Th_{2N-1}
Th_{2N} Th_{2N+1} • • • Th_{3N-1}
$Th_{N(K-1)}$ $Th_{N(K-1)+1}$ • • • Th_{NK-1}

Figure 4. Large warp active mask

3.1.1. Large Warp Microarchitecture Basic Operation

A large warp is statically composed of consecutive threads and has a warp ID and a single PC. It also has an active mask organized as 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., it would have K separate warps).

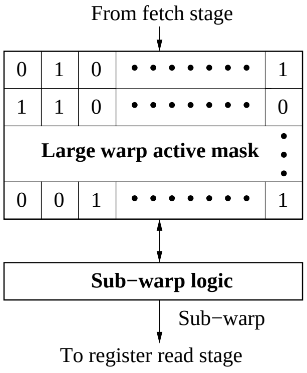
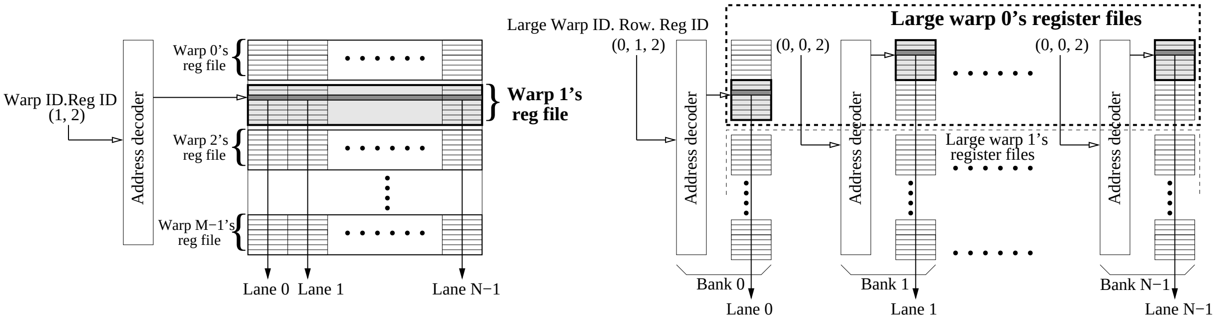


Figure 5. sub-warping logic

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 can then flow through the rest of the pipeline. Figure 5 shows the hardware structures added to support the large warp microarchitecture. 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 6(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 6(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. [11] and Fung et al. [9, 8] and shown in Figure 6(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.



(a) Baseline register files

(b) Large warp microarchitecture register files

Figure 6. Large warp vs baseline register file design

Figure 7 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 in Figure 7(a). Each cycle, the hardware searches each column of the active mask in parallel for an active thread and if found, selects those threads to be 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 7 illustrates how a large warp is dynamically broken down into four sub-warps over four successive cycles. Notice that in each cycle an active mask and row IDs are created for the newly formed sub-warp. The selected threads are highlighted each cycle and the newly created sub-warp's active mask and row IDs are shown beneath the large warp's active mask.

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,

[4] Such large SRAMs can't be built due to timing/energy considerations [4], so even the baseline register file is slightly banked.

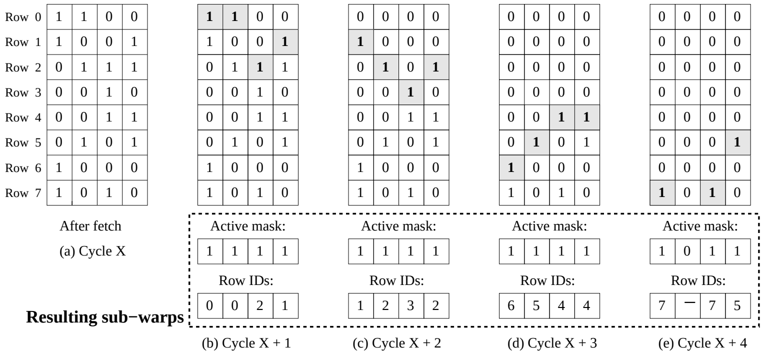


Figure 7. Dynamic creation of sub-warps

the inactive slots present in the baseline processor are filled with active threads during sub-warp creation. Therefore, only four efficiently packed sub-warps are created and SIMD resources are better utilized.

How a Large Warp Handles Divergence and Re-convergence: Large warps handle divergence and re-convergence much the same way that baseline warps do. However, 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 the large warp in fact diverged). The divergence stack is popped just as described in the baseline processor in Section 2.3 except that the divergence stack is checked for a matching entry only after all sub-warps have completed execution.

Barrel Processing in the Large Warp Microarchitecture: 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 restriction that once a large warp is selected, it is not considered for scheduling again until *all* sub-warps have completed execution. Using this restriction, there is no need for hardware interlocks for dependency handling just as in the baseline processor.

3.1.2. Large Warp Microarchitecture Optimizations We describe two optimizations for the LWM. The first optimization deals with the possible additional memory divergence caused by dynamic sub-warping. The second optimization aims to exploit the concept of large warps to reduce the latency of unconditional control instructions that cannot cause divergence and therefore only change the PC.

Memory Divergence Optimization: A well known software technique employed by many GPU programmers is to coalesce accesses to global memory [23, 22, 15]. Some consider this the single most important performance consideration for programming GPUs [22]. Therefore, GPU programmers take special effort to ensure that consecutive threads access consecutive memory locations. If the addresses are not consecutive (i.e., memory divergence), each warp will require multiple transactions to global memory which can significantly degrade memory throughput and therefore performance. The same problem exists for accessing data that has been cached. If all addresses do not map to the same cache line, multiple serial cache accesses must be made due to port contention.

The LWM can cause additional memory divergence not found in the baseline since non-consecutive threads can be grouped together into a sub-warp during dynamic sub-warping. To avoid this, when a large warp is executing an instruction that accesses global memory, sub-warps are formed corresponding to each row of the active mask. Although this does not pack threads into sub-warps as efficiently, it avoids the additional memory divergence created if we used the regular sub-warping mechanism.

Unconditional Control Flow Optimization: When a warp in the baseline processor executes an unconditional control flow instruction (i.e., a jump), only a single PC update is needed. The same is true for large warps and therefore there is no need to create multiple sub-warps when a large warp executes a jump instruction. Creating multiple sub-warps is wasteful since successive sub-warps would just overwrite the same value to the PC that the previous sub-warp wrote. Thus, sub-warping for a large warp executing a jump instruction completes in just a single cycle, allowing sub-warping for the next large warp to begin sooner. Note that for a large warp size of 256 threads and a SIMD width of 32, this optimization saves 7 cycles (assuming that the large warp is fully populated) because it creates only one sub-warp instead of 8.

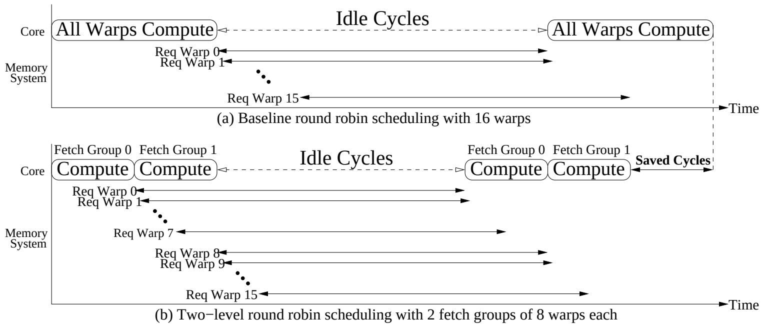
3.2. Two-level Warp Instruction Fetch Scheduling

As previously mentioned, 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 novel two-level round-robin scheduling policy which more effectively hides long latencies and therefore reduces idle 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 throughout the entire execution of the parallel program [17, 9]. This policy results in warps progressing through the program at approximately the same rate which is beneficial since warps tend to have a lot of data locality among them. Recall from Section 3.1.2 that programmers are encouraged to make consecutive threads access consecutive locations in memory to reduce memory divergence. This implies that the memory requests created by different warps have significant spatial locality (i.e., they all map to the same row buffer). Therefore, when a warp executes a memory request that misses in the cache, other warps are likely to produce cache misses that map to the 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 since an opened row buffer may be closed before other warps access it. 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 execute instructions from to hide the long latency resulting in several idle FU cycles.

To this end, we propose a two-level round-robin scheduling policy. With this policy, the concurrently executing warps are grouped into fixed size fetch groups. For example, 32 warps could be grouped into 4 fetch groups each with 8 warps (i.e., fetch group size equals 8). The scheduling policy selects a fetch group to prioritize and schedules warps from only that fetch group in a round-robin fashion until no warp from that fetch group can be scheduled (i.e., all the warps in that fetch group are stalled). When this happens, the next fetch group is selected and the policy repeats. Note that 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). Prioritizing a single fetch group prevents all warps from arriving at the same long latency operation at the same time. Instead, a smaller subset of warps (i.e., a fetch group) arrives at the long latency operation together and therefore there are instructions from other warps (in another fetch group) to execute while the warps in the original fetch group are stalled. In addition, since this policy switches between fetch groups in a fair round-robin fashion, it is able to preserve the row buffer locality between warps (which the conventional round-robin scheduling policy does very well). In summary, the two-level round-robin scheduling policy allows just enough separation between warps to effectively hide long latencies, but not too much as to destroy the row buffer locality among them.

Figure 8(a) shows how execution would proceed on a GPU core employing the conventional round-robin scheduling policy among all warps, and Figure 8(b) shows the same core using a two-level round-robin scheduling policy. In this simplified example there are 16 total warp contexts. Using the conventional round-robin scheduling policy, all warps progress evenly through the compute phase of the program but then all stall waiting on data to return from memory resulting in idle FU cycles before more computation can be done. On the other hand, using a two-level round-robin policy with 2 fetch groups of 8 warps each reduces the number of idle cycles as shown in Figure 8(b). Using the two-level policy, the warps in fetch group 0 proceed through the computation in half the time it took all 16 warps to do so and therefore reach the long latency operation (a memory request in this case) sooner. Since all warps in fetch group 0 are now stalled, a fetch group switch occurs and warps in fetch group one begin to execute the compute phase of the program while the requests created by fetch group 0 are serviced. Likewise, when data returns from memory, fetch group 0 resumes computation which again overlaps with memory requests being serviced. In summary, the two-level policy has reduced the number of idle FU cycles thereby improving performance.



(a) Baseline round robin scheduling with 16 warps

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

Figure 8. Baseline round-robin vs two-level round-robin scheduling, total warp count = 16

3.2.1. Fetch Group Size Setting the fetch group size correctly is very important for the two-level round-robin scheduling policy to work effectively. Recall from Section 2, our baseline processor uses a strict barrel processing model where once a warp is selected in the fetch stage, it cannot be fetched again until the warp flows through the entire pipeline and finishes execution. Therefore, having too small a fetch group size (less than the number of pipeline stages) will result in immediate fetch group switches since after all the warps in the current fetch group have been selected in the fetch stage, no warps in that fetch group will be ready to be fetched since they would all still be in the pipeline. Therefore fetch group switches would occur prematurely (i.e., before the warps are actually stalled on a long latency operation), which results in a scheduling policy similar to the baseline round-robin policy. On the other hand, too large a fetch group size also results in a scheduling policy similar to the baseline. For example, if the fetch group size is equivalent to the total number of warps (i.e., only 1 fetch group containing all the warps), the two-level round-robin policy is by definition equivalent to the baseline round-robin policy. The key is to have just enough warps in a fetch group to effectively fill up the pipeline so that a fetch group switch only occurs once those warps are actually stalled on a long latency operation.

3.2.2. Generalizing Two-Level Scheduling Although we have used memory requests as the long latency operation in our previous examples and discussions, the two-level scheduling policy is effective in tolerating any type of stall. For example, GPU ISAs [24] have complex instructions (e.g., sine, cosine, log, and exponent instructions) and the compute bandwidth for those operations are much less than that of simple instructions (since there are fewer complex FUs than simple FUs). If all warps arrive at a complex instruction at roughly the same time, there will be significant stalling. However, the two-level scheduling policy results in a subset of warps (i.e., a fetch group) reaching the complex operation together, and allows warps in another fetch group to execute simple instructions while the original fetch group slowly gets through the complex operation.

3.2.3. Integrating the LWM and Two-level Fetch Scheduling The Large Warp Microarchitecture and two-level round-robin scheduling can be combined. Just as in applying two-level scheduling to the baseline processor, the fetch group size must be carefully chosen when combining the LWM and two-level scheduling.

Having too small or too large a fetch group size will result in performance similar to the baseline round-robin fetch policy. However, with large warps, since there are fewer total warps on the core, there is less flexibility in choosing the ideal fetch group size. For example, with large warps of 256 threads each, there are only four large warps concurrently executing on the core and therefore only fetch group sizes of one, two, and four are possible. Recall from Section 3.1.1 that when imposing barrel processing with large warps, once a large warp is selected in the fetch stage, it cannot be fetched again until all of its sub-warps have finished execution. Therefore, a fetch group size of one is too small (even though there are 256 threads in the warp) and will result in premature fetch group switches. Likewise, a fetch group size of four (i.e., all 4 large warps in a single fetch group) is by definition equivalent to the baseline round-robin policy. Therefore, only a fetch group size of two is able to achieve any benefit. This lack of flexibility implies that two-level scheduling will be more efficient at smaller warp sizes (since there will be more flexibility in choosing the ideal fetch group size). However, larger warp sizes better tolerate branch divergence. We will evaluate this tradeoff in our results section.

3.3. Hardware Cost

The hardware cost for the LWM comes from restructuring the register file. As shown in Section 3.1.1, instead of a single address decoder, our mechanism requires a separate address decoder per SIMD lane. Previous work [11, 9, 8] estimates that using such a design results in little die area increase. For example, Jayasena et al. [11] propose stream register files with indexed access which require dedicated row decoders for each bank instead of a single shared decoder. They show that duplicating the row address decoders 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 [13]. Fung et al. [8] show similar results indicating that having separate address decoders results in an estimated 18.7% increase in register file area, which corresponds to 2.5% of GPU area.

In addition to the register file overhead, there are a few extra storage structures required by the LWM not present in the baseline GPU core. As explained in Section 3.1.1 and illustrated in Figure 5, 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 warp's actual active mask (since this would affect correctness) and therefore a copy of the large warp's active mask must be made before the warp can be broken down into sub-warps. For large warps of size 256 threads, this corresponds to 256 bits of storage (1 bit per thread in a large warp). In addition, as explained in Section 3.1.1, the large warp microarchitecture uses the 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. The total additional storage is 768 bits (96 bytes).

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

4. Comparison to Previous Work

Fung et al. [9, 8] proposed lane-aware *dynamic warp formation* (DWF) to address the branch divergence problem on GPU cores. Since their proposal also addresses the problem of underutilized SIMD resources, we extensively compare our work to DWF in this section and quantitatively evaluate DWF in Section 6.

DWF has three main disadvantages compared to our work: 1) the thread merging in DWF is not as efficient as dynamic sub-warping of a large warp, 2) DWF can cause significant additional memory divergence not found in the LWM or the baseline processor, and 3) the scheduling policies described above have several unintended effects with certain types of control flow which leads to not only lost opportunities for merging but also destruction of the data locality among warps. We elaborate on these issues below.

Incremental Merging: DWF only permits a retiring warp to be merged with the youngest matching warp in the list of warps waiting to be scheduled (i.e., the warp pool). This can result in inefficient merging by leaving holes in the active mask of older matching warps in the warp pool. We call this *incremental merging*. This is especially problematic when diverged threads reach a control flow merge (CFM) point. Both the baseline GPU processor and the large warp microarchitecture use a divergence stack to restore each warp back to its original thread count when the threads reach a CFM point. However, DWF does not use a divergence stack but instead relies on incremental merging. As a result, warps may not be brought back to their original thread count even after the CFM point has been reached. This can offset the benefit DWF achieves on divergent code resulting in performance degradation compared to the baseline processor.

Additional Memory Divergence: Since DWF reassigns threads to warps during merging, after the first divergent branch, warps in DWF may no longer contain consecutive threads. This results in additional memory divergence not found in either the LWM or the baseline processor.

Inefficiency of Scheduling Policies: The scheduling policies behind DWF try to keep warps together (i.e., progressing at the same rate) in order to maximize opportunities for merging threads. However, the proposed scheduling policies cannot always achieve this goal. As Fung et al. point out, the majority scheduling policy suffers from the problem of starving threads that take rarely executed paths. However, these threads must eventually execute and SIMD resource utilization will be very low when they do. Starving threads also present a problem with memory locality. When they eventually execute, data that used to be present in the cache may not be there anymore and open DRAM row buffers may have been closed. The post-dominator priority scheduling policy, which prioritizes warps that have passed the least number of post-dominators (i.e., CFM points) can also have a similar effect. For applications with loop divergence (i.e., where each thread iterates over a loop for a different number of iterations) and imbalanced nested branching, the number of post-dominators passed is not a good indicator of which warps need to catch up and can therefore result in excessive warp separation. In summary, it is difficult to find an ideal scheduling policy that always keeps warps together.

In contrast, by having large warps, a large number of threads are guaranteed to remain together (by definition of the large warp) even in the presence of biased branches, loop divergence, or imbalanced nested branching. Furthermore, our two-level warp scheduling policy does not suffer from the starvation issues associated with the scheduling policies in DWF since at each level of scheduling, a fair round-robin policy is used.

5. Methodology

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

Scalar frontend	1-wide fetch and decode stages, round-robin warp scheduling policy 4KB single-ported instruction cache
SIMD backend	In order, 5 stages, 32 parallel SIMD lanes
Register file and on-chip memories	64KB register file (16 32-bit registers per thread, 1024 concurrently executing threads) 32KB, 4-way set associative single cycle data cache, 1 read port, 1 write port, and 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 per bank 100 cycle row-hit latency, 300 cycle row-conflict latency, 128 GB/s memory bandwidth

Table 1. Baseline GPU core and memory configuration

Since the x86 ISA does not have instructions to aid with conditional branch divergence/re-convergence of parallel threads like GPU ISAs do [24], 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 [24].

We created parallel applications adapted from existing benchmark suites including Rodinia [7], MineBench [19], PARSEC [5], and NVIDIA's CUDA SDK code samples [21] in addition to creating one benchmark 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 [26] and [15]. Each benchmark runs to completion and consists of 100 million to 200 million dynamic instructions across all 1024 threads. Table 2 lists the benchmarks we used and briefly describes each one and also classifies each benchmark according to divergent branch intensity and idle functional unit intensity. For idle FU intensity, we consider benchmarks where the FUs are completely idle less than 20% of the time to be *low*, 20% to 40% to be *medium*, and greater than 40% idle cycles is considered to be *high*. For branch intensity, we calculated the average number of active threads per retired warp. Since warps start out fully populated (i.e., with 32 threads), only branch divergence can cause warps to retire with an active thread count of less than 32. If the average active thread count per warp was greater than 30, we consider the branch intensity to be *low*. Average active thread counts between 20 and 30 are classified as *medium*, and below 20 are considered *high*. Note that since the average active thread count is computed per retired warp (not per cycle), the idle FU cycles are factored out of this metric.

The metric we use to compare performance is retired *instructions per cycle* (IPC). Note that when a warp

Benchmark	Description	Divergent Branch Intensity	Idle FU Intensity
blackjack	Simulation of blackjack card game to compute house edge	high	low
sort	Parallel bucket sort of a list of integers	high	low
viterbi	Viterbi algorithm for decoding convolutional codes	medium	low
kmeans	Partitioning based clustering algorithm	medium	low
decrypt	Advanced Encryption Standard decryption algorithm	low	low
blackscholes	Computational finance, calculate the price of put/call options	low	medium
needleman	Calculate optimal alignment for DNA sequences	low	medium
hotspot	Processor temperature simulation	low	high
matrix_mult	Classic matrix multiplication kernel	low	medium
reduction	Parallel sum of a large vector of integers	low	high
histogram	Compute histogram for ASCII characters in a large text file	low	high
bfs	Breadth first search graph traversal	high	high

Table 2. Benchmarks

(or a sub-warp) executes an instruction, we treat each active thread in the warp (or sub-warp) as executing a single instruction. Therefore, if the warp (or sub-warp) size is 32 threads, the maximum possible IPC is 32.

6. Results

6.1. Overall IPC Results for the Large Warp Microarchitecture and Two-Level Scheduling

Figures 9 and 10 show the IPC and computational resource utilization for the baseline architecture (32 warps of 32 threads each, round-robin scheduling), dynamic warp formation (DWF), the large warp microarchitecture only (LWM), two-level scheduling only (2Lev), and the large warp microarchiteture combined with two-level scheduling (LWM+2Lev). Note that the SIMD width (32) and total thread count (1024) supported by the core is the same for each configuration. For DWF, we implemented both the majority and post dominator priority scheduling policies and chose the best performing of the two for each benchmark. For the LWM, we created 4 large warps of 256 threads each. For two-level scheduling only (i.e., two-level scheduling applied on top of the baseline), we set the fetch group size to 8 (i.e., 4 fetch groups, each consisting of 8 regular sized 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 2 (i.e., 2 fetch groups, each consisting of 2 large warps).

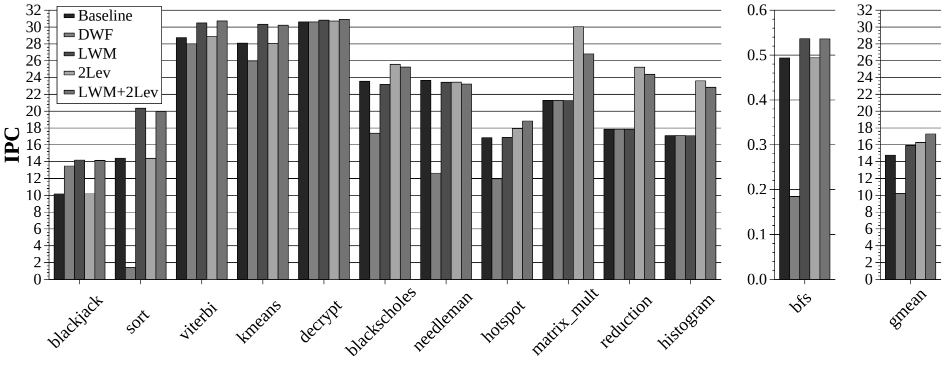


Figure 9. IPC for baseline, DWF, LWM only, 2-level scheduling only, and LWM with 2-level scheduling

As expected, the LWM only (third bar) significantly improves performance for branch-intensive applications (the leftmost 4 benchmarks), whereas two-level scheduling only (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 are compute bound. 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 10. 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.

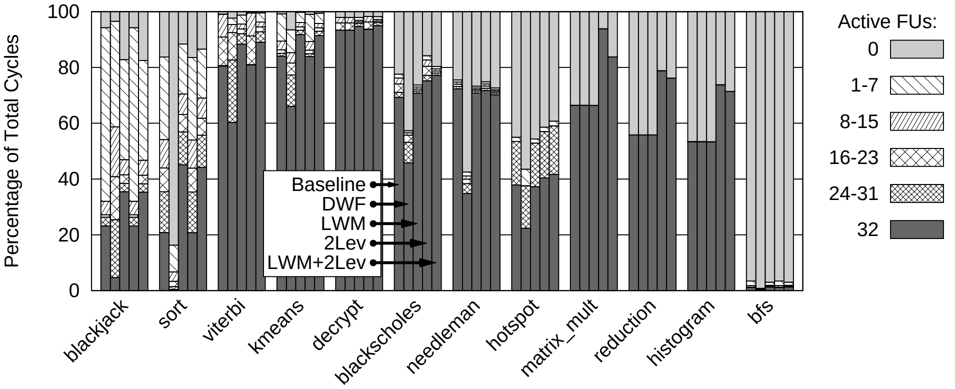


Figure 10. Functional unit utilization for baseline, DWF, LWM, 2-level scheduling, and LWM with 2-level scheduling

On the other hand, the rightmost benchmarks have the opposite behavior. For these idle-FU-intensive benchmarks, the LWM alone does not provide much benefit but two-level scheduling is very effective in reducing idle FU cycles as shown in Figure 10 and therefore improves performance for these applications.

In summary, LWM alone improves performance by 7.6%, two-level scheduling alone improves performance by 10.1%, and when our two mechanisms are combined, the benefits of each are mostly preserved resulting in 17.0% performance improvement on average across all benchmarks. DWF degrades performance compared to the baseline for the reasons described in Section 4.

6.2. Analysis of the Large Warp Microarchitecture

In this section, we show the effect of varying the large warp size and also show the effect of the LWM optimizations we presented in Section 3.1.2. In order to isolate the effect of the LWM, we used only the baseline round-robin policy (i.e., not two-level scheduling) for these results.

We vary the large warp size from the baseline of 32 threads per warp to a maximum of 512 threads per warp.[5] As seen in Figure 11, increasing the warp size improves performance up until a warp size of 256 threads. Increasing the warp size gives more potential for the sub-warping logic to create efficiently packed sub-warps and therefore in general, larger warp sizes are beneficial. However, a warp size of 512 threads

[5] We do not evaluate a large warp size of 1024 since given the strict barrel processing model we employ for large warps, having only a single large warp will result in very inefficient use of the pipeline.

actually performs slightly worse. The reason for this is that at a warp size of 512 threads, there are only 2 large warps on the core. In benchmarks with very biased branches (e.g., *blackjack* and *sort*), there will be times where each of the 2 large warps may only have a few active threads. Having only 2 large warps with just a few active threads each is not enough to fill the pipeline, resulting in several idle FU cycles and thereby reducing performance. Although a warp size of 256 threads is also affected by this, the problem is more pronounced at larger warp sizes.

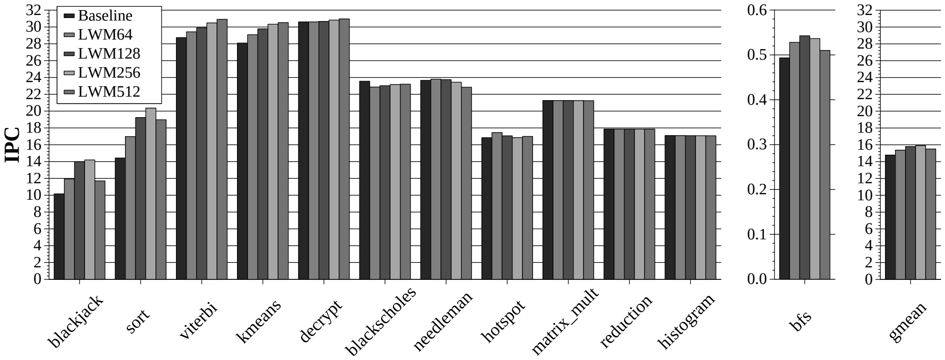


Figure 11. Effect of large warp size

Figure 12 shows the effect of the memory divergence and unconditional control flow optimizations we presented in Section 3.1.2. The memory divergence optimization does not provide much benefit. The reason for this is that memory instructions for which consecutive threads access consecutive memory locations tend to be on control-independent code. Therefore, when a large warp reaches such an instruction, its active mask is fully populated with all active threads. As such, the dynamic sub-warping mechanism cannot combine threads from different rows in the active mask and therefore no additional memory divergence occurs regardless of whether this optimization is turned on or off. This mechanism would be effective when coalesced (i.e., consecutive) memory instructions appear on control-dependent code paths. However, we find this to be rare in our benchmarks. The unconditional control flow optimization does slighty improve performance (by about 1%) by taking advantage of the fact that a large warp only needs a single PC update which is valid for all threads in the large warp.

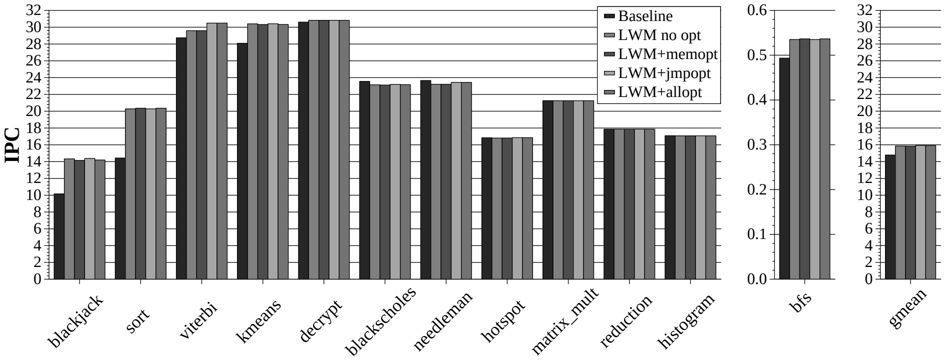


Figure 12. Effect of LWM optimizations

6.3. 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 in our baseline, we use fetch group sizes of 1, 2, 4, 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 13 shows the IPC as we vary the fetch group size.

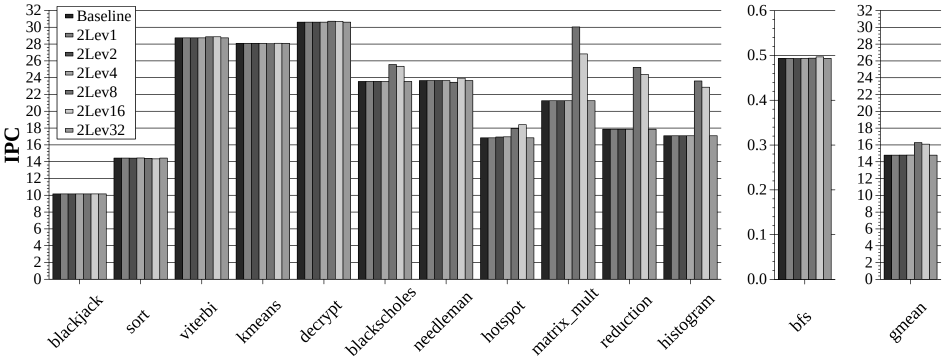


Figure 13. Effect of fetch group size on two-level scheduling

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 show that when the fetch group size is too small or too large, there is no performance benefit but fetch group sizes of 8 and 16 do provide benefit. As previously mentioned in Table 1, we model a simple 7-stage pipeline. Therefore, a fetch group size of 8 works best since 8 warps are enough to keep the pipeline busy (given the barrel processing model) and a fetch group switch only happens when warps are actually stalled on a long latency operation. A fetch group size of 16 still provides benefit but not as much. 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 (since the fetch group size is larger) and therefore is unable to hide latencies as well as 8 warps.

6.4. Interaction between the LWM and Two-level scheduling

Figure 14 illustrates the interaction between the LWM and two-level scheduling. In our notation, “LWM256+2Lev2” corresponds to the large warp microarchitecture with 256 threads per warp employed with two-level scheduling with a fetch group size of 2 large warps. For each warp size configuration, we plot only the best performing fetch group size for that configuration (e.g., for baseline warp sizes with two-level scheduling, we only plot performance for a fetch group size of 8 warps since that was the best performing option as shown in Section 6.3).

As expected, for the leftmost benchmarks, increasing the warp size up to 256 threads improves performance. However, for the rightmost benchmarks, increasing the warp size slightly decreases the benefits of two-level scheduling (yet still significantly improves performance compared to the baseline). As discussed earlier in Section 3.2.3, this is because with the LWM there are fewer total warps on the core and therefore

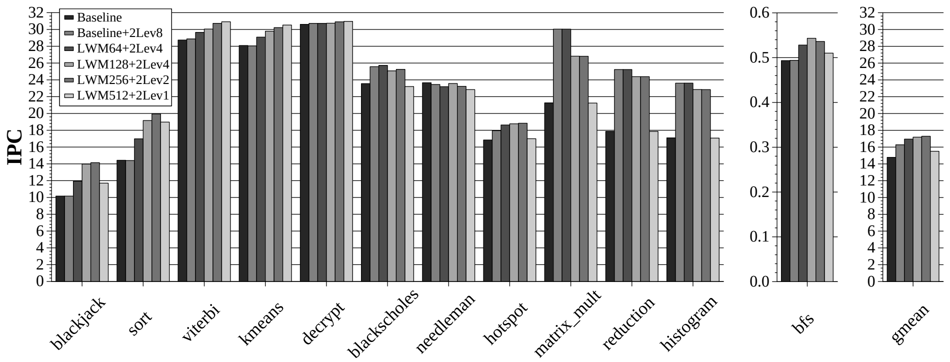


Figure 14. Interaction between LWM and two-level scheduling

less flexibility in choosing an ideal fetch group size.

Overall, a large warp size of 256 threads employed together with two-level scheduling with a fetch group size of 2 (LWM256+2Lev2) performs best when averaged across all benchmarks. Due to the fact that some applications benefit more from larger warps, and others do better at smaller warp sizes, a GPU core supporting hybrid warp sizes (the warp size could be set by the programmer or a profiler at compile time) is an interesting solution. We leave this for future work.

7. Related Work

7.1. Conditional Execution on Vector, Stream, and Graphics Processors

Using a bit mask to execute conditional code in processors that exploit SIMD parallelism is an old concept. The Illiac IV [6] had a *mode bit* per Processing Element (PE) which either turned on or off a PE during execution of a single instruction. Likewise, CRAY-1 [25] had a *vector mask* register which was used to vectorize loops with if/else statements. These bit masks are akin to the active mask currently found on existing GPU cores, where each bit in the active mask can activate or deactivate the corresponding SIMD lane.

Allen et al. [3] introduced the idea of predicated execution. Predicated execution has been extensively used on GPUs to implement conditional code. Predicated execution allows GPU cores to implement conditional code without branch instructions but does not deal with the problem of underutilized SIMD resources when SIMD lanes are masked off by predicate bits.

Smith et al. [29] 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. In their scheme, 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 over false values, our approach finds active operations from threads in a large warp to fill the holes caused by branch divergence.

Kapasi et al. [12] introduced *conditional streams*, which allow stream processors to conditionally filter an input stream before it is processed. For example, if computation needed to be done on only the positive values of a stream of integers, a separate kernel would have to be created by the programmer which conditionally filters out the negative values. This condensed output stream becomes the input stream to the kernel performing the actual computation. This leads to more efficient use of the SIMD pipeline on stream processors since computations that are not needed are not performed. However, this mechanism requires 1) communication between different SIMD lanes in order to filter an input stream which is costly in terms of hardware complexity, 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 that improves SIMD utilization in the presence of branch divergence. It does not require programmer or programming model support, and can improve the performance of existing code.

Krashinsky et al. [16] propose 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, our mechanism uses a strictly SIMD backend and tolerates branch divergence by dynamically breaking down large warps into efficiently packed sub-warps.

Meng et al. [18] propose Dynamic Warp Subdivision (DWS) whereby when a warp diverges, two warp-splits are formed which can be scheduled independently. Although this does not increase SIMD resource utilization, it may increase the amount of Memory-Level Parallelism (MLP) since both sides of a divergent branch can be executed concurrently. As such, DWS is orthogonal to our mechanism and can be employed on top of the large warp microarchitecture by splitting up a large warp upon branch divergence.

7.2. Fetch Scheduling

Many previous proposals analyzed and proposed scheduling policies for threads on MT or SMT cores [2, 32, 31]. However, none of these policies were designed for scheduling warps on GPUs. In fact, many of these policies were evaluated on workloads consisting of multiple different applications and not threads of the same application. GPU scheduling is unique in that the warps to be scheduled have much data locality among them. Also, GPUs support many more warp contexts simultaneously compared to these MT and SMT cores allowing zero cycle context switching among all concurrently executing warps.

Lakshminarayana et al. [17] evaluate several possible fetch scheduling policies for GPUs. However, the policies they evaluate do not include the two-level scheduling described in this paper. Furthermore, most of the scheduling policies they evaluate result in warps progressing uniformly through the program (similar to pure round-robin) so that data locality among warps can be exploited. In contrast, our two-level policy allows warps to arrive at a long latency instruction slightly apart from each other in time thereby more effectively hiding the latency. Our policy also exploits the data locality among warps since at each of the two levels of scheduling, a fair round-robin policy is used. We also note that the policies evaluated in [17] can be used as the scheduling policy within either of the two levels of our two-level policy.

8. Summary and Conclusion

In this paper, we propose two new mechanisms to improve GPU performance by better utilizing the computational resources on GPU cores in the presence of branch divergence and long latency operations.

To alleviate the performance penalty caused by 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 novel two-level round-robin warp instruction fetch 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 each mechanism significantly improves performance. Combined together, both techniques improve performance by 17.0% 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.

References

[1] Advanced Micro Devices, Inc. *ATI Stream Technology*. http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx.

[2] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: a processor architecture for multiprocessing. In *ISCA-17*, 1990.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL*, 1983.

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

[5] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT-17*, 2008.

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

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

[8] W. W. L. Fung et al. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM TACO*.

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

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

[11] N. Jayasena, M. Erez, J. Ahn, and W. Dally. Stream register files with indexed access. In *HPCA-10*, 2004.

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

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

[14] Khronos Group. *OpenCL Parallel Computing for Hetergeneous Devices*. http://www.khronos.org/developers/library/overview/opencl_overview.pdf.

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

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

[17] 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.

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

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

[20] J. Nickolls and W. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56 –69, 2010.

[21] NVIDIA. *CUDA C SDK Code Samples*. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[22] NVIDIA. *CUDA Best Practices Guide Version 3.0*, 2010.

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

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

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

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

[27] L. Seiler et al. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29(1):10–21, jan-feb 2009.

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

[29] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *ISCA-27*, 2000.

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

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

[32] D. M. Tullsen et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA-23*, 1996.