Phase Aware Warp Scheduling: Mitigating Effects of Phase Behavior in GPGPU Applications

Mihir Awatramani, Xian Zhu, Joseph Zambreno and Diane Rover
Department of Electrical and Computer Engineering
Iowa State University
Ames, Iowa, U.S.A.
Email: {mihir, xian, zambreno, drover}@iastate.edu

***Abstract*—Graphics Processing Units (GPUs) have been widely adopted as accelerators for high performance computing due to the immense amount of computational throughput they offer over their CPU counterparts. As GPU architectures are optimized for throughput, they execute a large number of SIMD threads (warps) in parallel and use hardware multithreading to hide the pipeline and memory access latencies. While the Two-Level Round Robin (TLRR) and Greedy Then Oldest (GTO) warp scheduling policies have been widely accepted in the academic research community, there is no consensus regarding which policy works best for all applications.**

In this paper, we show that the disparity regarding which scheduling policy works better depends on the characteristics of instructions in different regions (phases) of the application. We identify these phases at compile time and design a novel warp scheduling policy that uses information regarding them to make scheduling decisions at runtime. By mitigating the adverse effects of application phase behavior, our policy always performs closer to the better of the two existing policies for each application. We evaluate the performance of the warp schedulers on 35 kernels from the Rodinia and CUDA SDK benchmark suites. For applications that have a better performance with the GTO scheduler, our warp scheduler matches the performance of GTO with 99.2% accuracy and achieves an average speedup of 6.31% over RR. Similarly, for applications that perform better with RR, the performance of our scheduler is within of 98% of RR and achieves an average speedup of 6.65% over GTO.

I. Introduction

Graphics Processing Units (GPUs) are widely used for accelerating general-purpose applications that exhibit a high degree of data level parallelism. Sixty systems on the June 2015 TOP500 supercomputer list use GPUs as accelerators [1]. The reason for this wide adoption is the high amount of floating point throughput that GPUs can provide as compared to their CPU counterparts. For example, the peak double precision floating point throughput of an NVIDIA K80 is 1.87 TFLOPs [2] and that of an AMD Firepro is 1.48 TFLOPs [3]. In comparison, an Intel Xeon E7, a high end CPU used in the HPC domain can achieve 408 GFLOPs [4].

To achieve such high computational throughputs, GPUs dedicate a large portion of their die area to functional units. As a trade-off, GPU chips do not include the hardware units which have been traditionally used for latency hiding in CPUs. For example, GPUs use a non-speculative in-order pipeline, do not have branch predictors, and have much smaller caches compared to CPUs. Consequently, to hide the pipeline and memory access latencies, GPUs perform massive multithreading in hardware. Each core executes a very high number of threads in parallel, and a hardware scheduler interleaves their execution to hide latency. Specifically, groups of threads are executed as SIMD units called warps [5] (or wavefronts [6]), and the scheduler overlaps the latency of warps waiting on long latency operations with computation from other warps. The policy used by the warp scheduler is pivotal in being able to achieve a throughput which is close to peak, as it largely affects how well the latencies and computations are overlapped.

There is a large body of previous work on warp scheduling techniques. A majority of the initial works focused on mitigating warp-divergence, a problem that occurs when threads within a warp diverge in their program paths [18, 8, 19, 20]. Recently, a few works have designed techniques that focus on a subset of applications that exhibit specific characteristics. The authors in [9] focus on applications that are sensitive to L1 data cache, while the authors in [10] focus on applications with irregular workloads. However, the underlying policy which selects the next warp to be dispatched for execution has received rather less attention.

Two underlying policies have been widely adopted, namely round robin (RR) and greedy then oldest (GTO). The RR policy rotates the priority of warps in round robin order after each selection. The GTO policy on the other hand, always gives a higher priority to warps that are launched earlier. Today's state of the art GPUs employ hierarchical implementations of these policies for increasing energy efficiency [7, 8]. A smaller set of warps (typically 6 to 8), from all the active warps (typically 48 to 64), referred to as a fetch group, is kept in a separate queue called the ready queue. The scheduler only selects warps from the ready queue for execution. This makes the warp selection and scoreboarding logic simpler and more energy efficient. A warp in the ready queue is replaced only when it arrives at a long latency operation, such as a memory request. It has been shown that warps in a fetch group have enough parallelism to hide the shorter ALU latencies [7]. Moreover, prioritizing execution of subsets of warps spaces out the requests to main memory in time, which results in a better overlap of memory latency and computation [8]. Both the RR and GTO policies can be used for assigning priority to the fetch groups as well as warps within the fetch group.

In [9], Rogers et al. claim that using the GTO policy for prioritizing the fetch groups as well as warps within a group gives the best performance. By contrast, our experimental results show that while the GTO policy performs better for some applications, some applications show better performance when RR is used, while others have comparable performance with either scheduler. To understand the disparity between performance of warp schedulers for different applications, we analyze how the warps progress through the program's instructions. The set of warps within a fetch group proceed together through the program instructions, at approximately the same pace, until they arrive at an instruction that depends on a long latency operation. When selected the next time, they again proceed until the next long latency operation, and so on. In this way, the program is essentially divided into regions of computations, separated by instructions that depend on long latency operations. We refer to these regions as **phases**.

Our results clearly indicate that the length and arrangement of phases have a direct impact on the performance of warp scheduling policies. We show that the disparity regarding which policy performs better for a particular application can be explained by understanding how warps progress through the different phases of the application. With this understanding, we then design a warp scheduling policy that uses information regarding phases that is embedded in the program's instructions by the compiler. Our policy results in a performance which is always closer to the better performing policy for the respective application. Our contributions can be summarized as follows:

1. We characterize phase behavior in GPGPU applications, and via case studies of real-world applications show how phase characteristics impact the performance of the RR and GTO scheduling policies.

2. We describe how the phase information can be inserted at compile time and design a hardware warp scheduler that uses this information to be more robust to application phase behavior.

The remainder of this paper is organized as follows: In Sect. 2 we give an overview of the GPGPU programming model and GPU hardware architecture. In Sect. 3 we formally define program phases via an illustrative code example. We then analyze two real-world applications in detail, and show how phase behavior affects the performance of warp schedulers for those applications. In Sect. 4, we describe our phase-aware scheduling policy, and its software and hardware implementation. In Sect. 5, we provide experimental results that compare our phase-aware scheduler to the RR and GTO warp schedulers. In Sect. 6 we discuss related work and compare our contributions to the published work in this field. Finally, in Sect. 7 we provide our conclusions from this work, followed by next steps in the direction of compiler assisted warp scheduling techniques.

II. Background

A. The GPGPU Programming Model

The two most widely used GPGPU programming languages are OpenCL [6] and CUDA [5]. To evaluate our work, we used applications written in CUDA. However, the concepts are directly applicable to applications written in OpenCL as well.

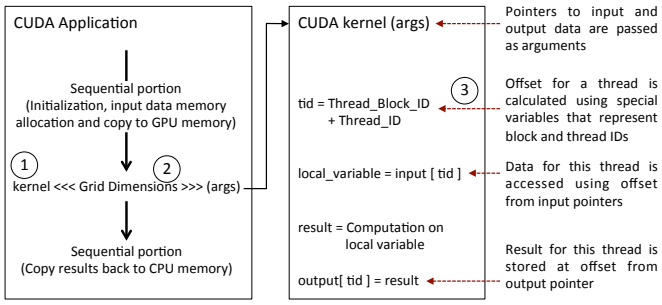


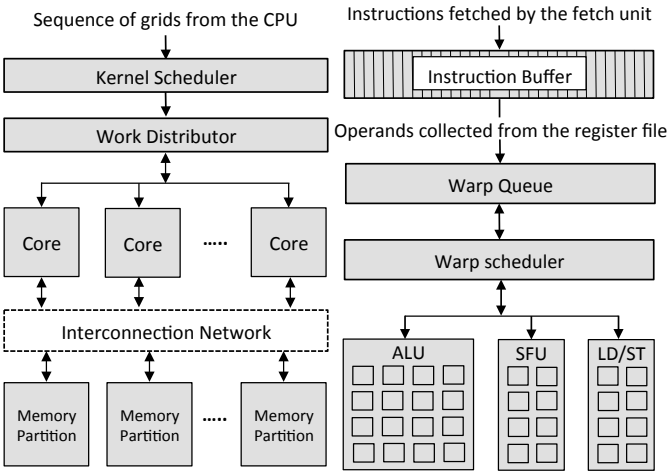
Fig. 1: Structure of a CUDA application with one kernel.

Fig. 1 illustrates the structure of a basic CUDA application. Each portion to be offloaded to the GPU is written as a separate function, called a **kernel** ①. The function launch syntax specifies the total number of threads that would execute this kernel within the <<< >>> structure ②. This group of threads, which represents the entire workload for a given instance of the kernel call, is called a **grid**. The parameters within <<< >>> describe the relationship between threads and data. The grid is partitioned into groups of threads, called thread blocks. Threads within a thread block can use synchronization barriers and share data using on-chip SRAM. Finally, each individual thread executes all instructions in the kernel function on its respective data, resulting in a Single Program Multiple Data (SPMD) form of programming model. Fig. 1 marker ③ depicts how the block and thread identifiers are used within the kernel, to index data specific to a thread from GPU memory.

B. GPU Hardware Architecture

The hierarchical structure of threads described in the previous subsection is scheduled on GPU hardware by two schedulers. Fig. 2(a) shows a high level depiction of the GPU. The Work Distributor shown in the figure is a chip-level hardware scheduler which issues thread blocks to cores. At the beginning of a kernel, it launches blocks on cores one by one, until a core runs out of resources to support an entire block. The primary goal of this scheduler is to maintain a balanced workload across all cores and keep each core completely full by launching the next block in the grid on any core that finishes a block. As the programming model does not guarantee the scheduling order of threads that belong to different blocks, blocks are assigned to cores in any order. Further details of the work distributor architecture can be found in [11, 23, 24].

1) GPU Core: Thread blocks launched on the core are further partitioned into groups, of typically 32 threads, called warps. As mentioned in Sect. I, warps are entities used for execution by the hardware units on the core. Fig. 2(b) shows a depiction of some of the units on the GPU core which are used for general purpose computing.



(a) Chip level block diagram. (b) Core block diagram.

Fig. 2: Overview of GPU architecture.

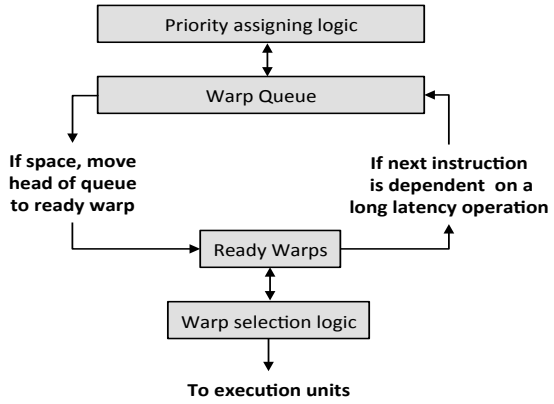


Fig. 3: Block diagram of the two level warp scheduler. An equivalent block diagram of the single level scheduler would not have the Ready Warps queue.

The fetch unit fetches instructions for each warp, from the instruction cache into the instruction buffer. The instruction buffer has a separate entry for each warp, allowing each warp to be at its own instruction in the kernel. Once the current instruction is decoded and its operands are collected from the register file, the warp is placed into the Warp Queue. The warp scheduler selects a warp from this queue and dispatches it for execution on the vectorized functional units. Each thread within a warp executes the same instruction on its data, in Single Instruction Multiple Data (SIMD) fashion. Indeed, the various hardware units are vectorized and indexed at the warp abstraction level, because of this SIMD form of execution adopted by the GPU architecture. The register file is divided at the warp granularity as well. Context for a warp is kept live in the register file until it completes the entire kernel.

2) Warp Scheduler: Initial warp schedulers used a single queue to store all the warps that are active on the core. This is depicted as the Warp Queue in Fig. 2 and Fig. 3. The warp selection logic arbitrates among all the warps in this queue every cycle. We refer to such schedulers as **single level schedulers** in this paper. As GPU cores became bigger and could support a higher number of warps (current architectures support up to 64), arbitrating among all the active warps every cycle became less energy efficient. Gebhart et al. [7] and Narasiman et al. [8] proposed hierarchical scheduling policies, which maintain a smaller subset of warps in a Ready Warps queue (refer to Fig. 3). The set of warps in the ready warps queue is referred to as a fetch group (FG). The warp selection logic arbitrates only among warps in the ready queue every cycle. When a warp becomes pending on a long latency operation, it is put in the larger warp queue, and a warp from the fetch group with the next highest priority is brought to the ready queue in its place. We refer to such schedulers as **two level schedulers** in this paper.

The policy used by the warp selection logic determines how well the warps in the ready queue can hide the shorter pipeline latencies. The priority assigning logic, on the other hand, assigns priority to the warps in the larger Warp Queue. Its policy dictates which fetch group would be moved next to the ready queue. Consequently, it has an effect on how the fetch groups progress through the kernel instructions. The authors of [7, 8] use a round robin policy to assign priority to the fetch groups, while the authors of [9] showed that using the GTO policy yields a better performance. As mentioned in Sect. I, we show that the performance of these policies depends on how the computations and memory accesses are laid out in the kernel instructions (program phases). In the next section we formally define program phases and depict this effect. To compare our work, we use the RR and GTO policies for the single level scheduler; and scheduling the fetch groups for the two level scheduler. Warps within the ready warps queue are always scheduled using RR.

III. Phase Behavior in GPGPU Kernels

In this section, we formally define phases in the context of GPGPU kernels. We then illustrate the impact of phase characteristics on warp scheduling policies using two real-world applications.

A. Definition of Kernel Phases

Fig. 4 shows an example of a simple CUDA kernel that adds two vectors. Three high level sections are shown in the code. First, the index for a thread is calculated using special variables that store the thread and block identifiers (`threadIdx.x` and `blockIdx.x`), and the thread block dimension (`blockDim.x`). The index is then used as offset from the base addresses (passed as arguments to kernel call) to load data for that thread. The computation is performed and result is stored back in the third section. The right box in the figure shows the corresponding assembly code.

We define ***phase*** as *a set of consecutive instructions such that, no instruction in the current phase has an input operand that is produced by a long latency instruction from the current phase.*

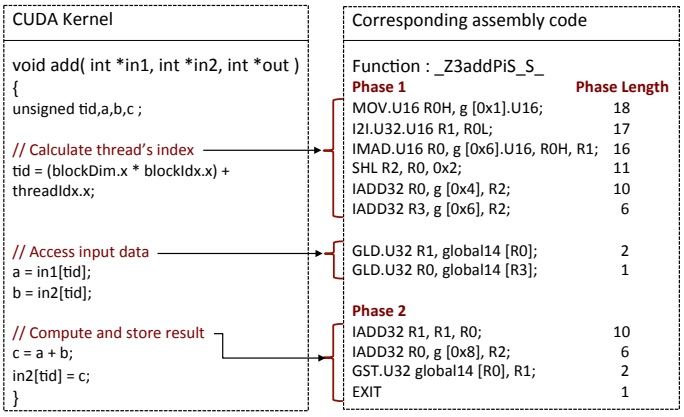


Fig. 4: CUDA kernel for vector addition and its corresponding assembly code showing phases and phase length.

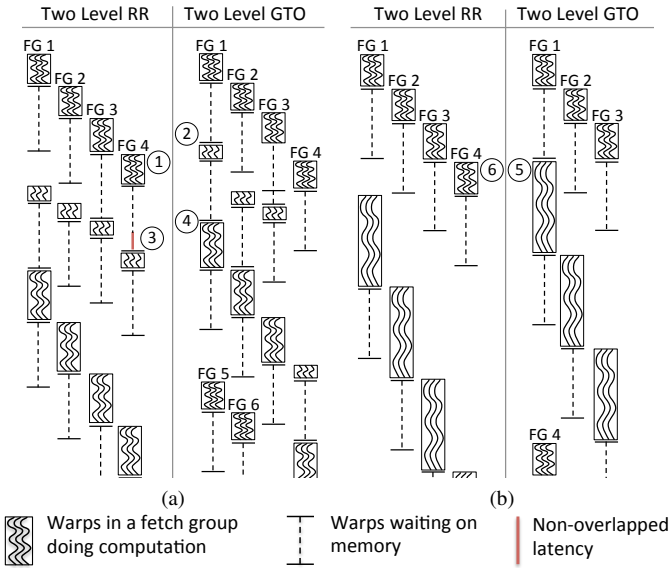


Fig. 5: Impact of phase length on warp scheduling.

Observe that this kernel has two phases (refer to the assembly code in Fig. 4). Phase 2 begins at the `IADD32 R1,R1,R0` instruction. The input operands `R0` and `R1` are produced by memory load instructions (`GLD`), which belongs to the set of long latency instructions[1]. It should be noted that in real-world applications, contrary to the example shown, it is common to have multiple instructions in a phase that depend on long latency instructions from the previous phase. A new phase begins only if an instruction depends on a long latency instruction from the current phase. We show a simple example here for brevity. The structure of a phase is often as follows: multiple memory loads at the beginning of a phase (initial loads), followed by computations that depend on loads from the previous phase, and then an instruction that depends on one of the initial loads (this would begin a new phase). For the kernels we studied for this work, the number of phases in a kernel varies from 3 to 45.

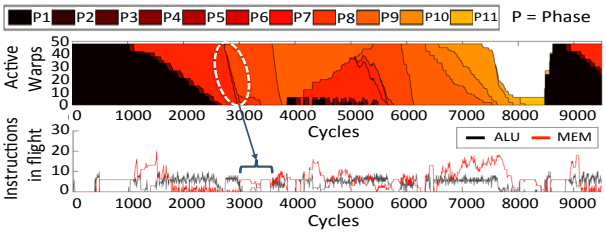
B. Effect of Kernel Phases on Warp Schedulers

In the two level warp scheduler, a warp is moved from the Ready Warps queue to the larger pool of all warps, when it arrives at an instruction that depends on a long latency operation. As such instructions lie at phase boundaries, warps proceed through the kernel instructions one phase at a time. When a warp reaches the end of a phase, it is put back in the larger warp pool and a different fetch group gets a chance to execute. As the fetch group maintains its priority until it reaches the end of a phase, one of the main factors that affects the performance of the warp schedulers is phase length (refer to Fig. 4). Phase length is computed by summing the instruction latencies, from the last to the first instruction of a phase. It is an approximation of the minimum number of cycles that a warp would take to reach the end of a phase. The lengths of phases 1 and 2 in Fig. 4 are 18 and 10 respectively.

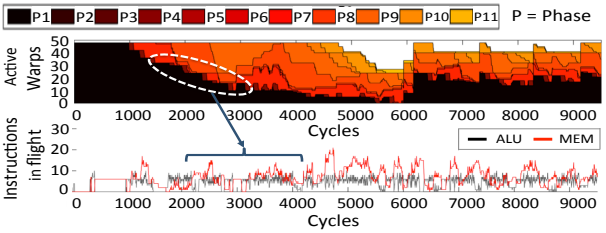
Fig. 5 is a depiction of the effect of phase length on performance of the warp schedulers. For simplicity, the illustration assumes that all warps in a fetch group (FG) arrive at the end of a phase simultaneously. Fig. 5(a) is an example of an application where the GTO scheduler performs better than RR. It has a medium length phase, followed by a short phase and then phase of long length. Notice that the computation from medium length phase of FGs 2 and 3 are enough to hide the memory latency of FG 1. At this time, the RR scheduler selects FG 4 ①, while the GTO scheduler selects FG 1 ②. Observe that, with RR scheduling all the FGs arrive at the short length phase at the same time. As computation from three short length phases is not enough to hide the memory latency, some of the latency is exposed ③. In contrast, as the GTO scheduler selects FG 1 at ②, FG 1 arrives at the long length phase earlier ④. This long phase is then used to hide the latency which was exposed in the case of RR scheduling. In general, for applications that have shorter length phases in the middle of the application, the GTO scheduler performs better than RR.

Fig. 5(b) is an example of an application where the RR scheduler has a better performance compared to GTO. Notice that the application has a medium phase, followed by a phase of long length. Similar to the example of Fig. 5(a), computation from three FGs is enough to hide the memory latency of FG 1, and the GTO scheduler switches back to FG 1 ⑤. As this phase is long enough to overlap the latency of outstanding memory requests, the load on the memory subsystem is reduced. On the contrary, the RR scheduler selects FG 4 ⑥. This results in sending out more requests to memory, and thus better utilization of the memory bandwidth while executing the long length phase. We can observe that around the time when the GTO scheduler begins fetch group 4, the RR scheduler has already executed a significant portion of it. In general, for applications that have extremely long phases in the middle of the kernel, the RR scheduler performs better than GTO.

[1]In addition to load and store, conditional and unconditional branches, and synchronization instructions are also considered as long latency instructions.

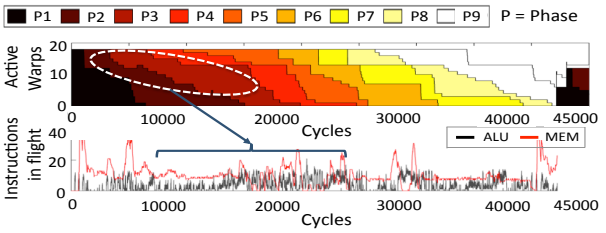


(a) Two Level Round Robin

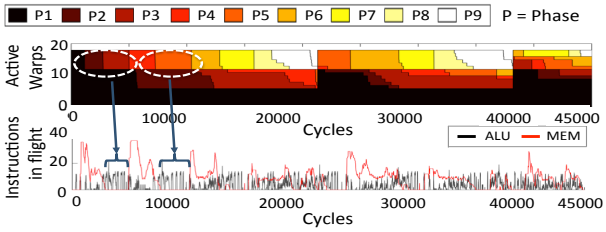


(b) Two Level GTO

Fig. 6: B+Tree Phase Graphs



(a) Two Level Round Robin



(b) Two Level GTO

Fig. 7: CFD Phase Graphs

C. Illustrative Applications

The effects of phase length on warp scheduling policies described in the previous section, can be summarized as follows:

1. Performance of the RR policy is adversely affected in kernels that have shorter length phases, due to all the warps arriving at these phases at the same time.
2. Performance of the GTO policy is adversely affected in kernels that have long length phases, as the scheduler keeps choosing warps from these phases, thereby under-utilizing the memory bandwidth.

In this section, we explore two real-world applications that clearly demonstrate these effects. The top figures in each of the subplots of Fig. 6 and Fig. 7 plot the total number of warps in each phase at a given cycle. The bottom figure plots the total the number of ALU instructions and memory requests that are in flight.

1) B+Tree: B+Tree is an example of a kernel for which the GTO scheduler achieves a better performance compared to RR. It launches 48 warps on each core, as six thread blocks of eight warps each. The fetch group size is six.

Observe in Fig. 6(a) that, when warps from the first fetch group (FG) complete the first phase, warps in the next FG execute. Warps completing a phase can be seen in the plot when the number of warps in a particular phase decreases. This is followed by the third FG, and so on. Thus, due to RR scheduling warps proceed through the kernel instructions one phase at a time. As mentioned in Sect. III-B, this trait of the RR scheduler becomes an issue in kernels which have short length phases. Observe that phase 4 is extremely short (marked oval). All the warps finish this phase and arrive at the start of phase 5 at around the same time. Observe that during this time, the memory load is high and ALU load becomes almost zero. This is because all warps are waiting for the memory requests issued in phase 4. The phase problem alleviates a little after cycle 4000 due to some warps branching back to phases 1 and 3. However, only a small portion of the runtime is shown here. The application grid is of 65535 blocks and this pattern keeps repeating throughout the kernel execution after every 6^{th} block.

In contrast, observe in Fig. 6(b), that after three fetch groups complete, the GTO scheduler selects FG 1. This is observed in the figure when the number of warps in phase 1 stop decreasing and the number of warps in phase 2 start to decrease. Due to this decision, only a small set of warps arrive at the short length phase 4, at the same time. As the other warps are in the longer length phases, the scheduler can switch to them to hide the memory latency. Observe in the lower plot of Fig. 6(b), the amount of time when the ALU and memory load becomes zero has reduced as compared to the RR scheduler. Moreover, observe in the area plot that the distribution of warps across different phases (marked oval) is much better in GTO as compared to RR.

2) Computational Fluid Dynamics (CFD): CFD is an example of a kernel for which RR scheduling achieves better performance as compared to GTO. It launches 18 warps on each core, as three thread blocks of 6 warps each. Observe in Fig. 7(b) that the first fetch group (FG) of 6 warps starts executing ahead and reaches phase 2. Notice that the GTO scheduler selects FG 1 again (first marked oval). This is because, as phase 1 is a long length phase, the memory requests sent near its beginning are completed by the time warps reach phase 2. As the next two phases are long as well, the warps in FG 1 become pending only when they reach phase 4. At this time fetch group 2 gets to execute. Observe that when warps from FG 2 reach phase 2 and become pending, FG 1 gets selected again and executes until it reaches phase 6 (second marked oval).

In this manner, due to phases of long length, a group of warps keeps getting the priority. Consequently, other warps do not get a chance to execute and issue their memory requests. Observe in the lower plot of Fig. 7(b) that the memory load becomes zero in the duration when warps from FG 1 are in

the long compute phases. In comparison, the RR scheduler switches priority to the next fetch group at phase boundaries. Due to this, warps that were in the shorter length phases are able to execute and send the memory requests. We can observe in Fig. 7(a) that the warp distribution across different phases is much better. Also, notice that the memory load is more regular and has lesser fluctuations as compared to GTO. The average memory load for this kernel with the two level RR scheduler was 4% higher as compared to the two level GTO.

IV. Phase Aware Warp Scheduling

In this section, we first propose a dynamic scheduling policy based on phase length that can mitigate the negative effects of phases outlined in the previous section. We then provide details of the compiler frontend that adds the required phase information in program instructions and hardware implementation of the warp scheduler that uses this information at runtime.

A. Scheduling Policy

In the previous section, we showed that performance of the RR scheduler is affected if the kernel has phases of short length and all the warps arrive at such phases simultaneously. On the other hand, performance of the GTO scheduler is adversely affected when the kernel has long phases and the scheduler keeps selecting the set of warps that are in the long phase.

Our policy is based the following simple observation: *Adverse effects of the RR and GTO scheduling policies can be mitigated by always choosing the warp that has the shortest length for its next phase.*

Consider a kernel with two phases P_i and P_j, such that P_j is shorter than P_i.

Case (1): The RR policy chooses a warp in phase P_i. This implies that warps in P_j were selected before this.
(a): P_i is before P_j in program order. This is the more common case given that warps in P_j were selected earlier. Selecting warps in phase P_i would get all warps to the shorter phase P_j, leading to a possibility of non-overlapped memory latency. Hence, it would be ideal to first select warps in P_j.
(b): P_i is after P_j in program order. This would happen if warps (currently in P_j) executed before this and branched back to an earlier phase. Selecting warps in phase P_j would get all warps to the longer phase P_i. Note that in this case, selecting warps from P_i would harm performance only if they branch back to phase P_j. Nonetheless, selecting warps in P_j would not negatively impact performance.

Case (2): The GTO policy chooses a warp in phase P_i. This implies that warps in P_i were launched before warps in P_j.
(a): P_i is after P_j in program order. This is the common case as GTO gives priority to older warps, causing them to be ahead in the program. If phase P_i is extremely long, choosing a warp from phase P_i might under-utilize the memory bandwidth. Hence, it would be better to first execute warps in P_j and then overlap the memory latency using warps in phase P_i.
(b): P_i is before P_j in program order. This would happen if warps with the lower index have branched back to P_i. Again, selecting warps in phase P_j would issue memory requests which can then be overlapped by warps from P_i. Note that in this case, selecting warps from P_i would have the same effect, if warps in P_j also branch back to P_i. Nonetheless, selecting warps which are in phase P_j would not negatively impact performance.

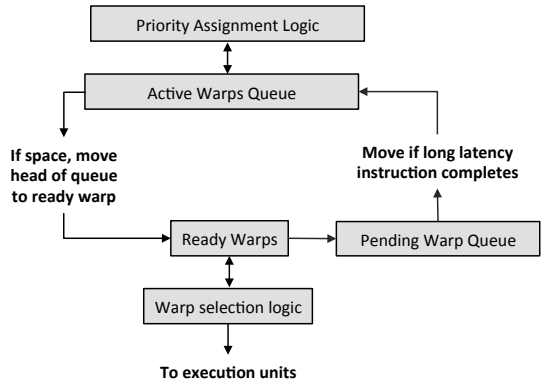


Fig. 8: Block diagram of our two level warp scheduler.

B. Implementation

1) Front-end: The phase length information is computed at compile time and included in the kernel instructions. As the CUDA ISA is not open-source, we use PTX-Plus [12]. PTX is an intermediate assembly language generated during the compilation of CUDA kernels. PTX-Plus is a modified version of PTX which closely matches the ISA that executes on the hardware. We chose to work with PTX-Plus because we observed that PTX is generated before the compiler has performed instruction scheduling. Due to this, the memory load and store instructions are scheduled very close to their dependent instructions, resulting in the instruction sequence being fragmented into several phases.

A long-op register is defined as a register that is a destination operand of a long latency instruction. To create phases, a set of long-op registers (empty at initialization) is maintained. The PTX-Plus assembly code is parsed at compile time from top to bottom and long-op registers are added to the set one instruction at a time. The first instruction that consumes any register currently in the set marks the start of a new phase. At the start of each phase, the set is cleared. This assumes that long latency instructions issued close to each other would complete around the same time, and avoids creating several short phases. In addition, each basic block starts a new phase.

Once the phases are created, the code is traversed backwards to calculate phase lengths. Within each phase, instruction latencies are accumulated from the last to the first instruction as mentioned in Sect. III-A. This simultaneously calculates two pieces of information. Each instruction is assigned a phase distance, which approximates the number of cycles a warp executing this instruction would take to reach the end of the phase. Additionally, each phase is assigned a phase length, which approximates the number of cycles a warp takes to execute the phase.

2) Hardware Implementation: We implemented the schedulers in GPGPU-Sim v3.2 [12], a cycle-accurate GPU architecture simulator. The phase distance mentioned in the previous subsection is used by the phase-aware single level scheduler, while the phase length is used by the phase-aware two level scheduler.

Single Level Schedulers: As mentioned in Sect. II-B, single level schedulers maintain a queue of all warps that are active on a core. All the warps in the active queue are checked every cycle and the one with the highest priority is chosen. The priority assigning policy is the key difference between the different schedulers. The RR scheduler rotates the priority after each selection, while the GTO scheduler always assigns the highest priority to the oldest warp. The phase aware scheduler compares the phase distance of the current instruction for each warp. Warp at an instruction with the lowest distance from the next phase is chosen. For warps that are at the same distance, the oldest warp is chosen first. This requires the distance information to be added for each instruction. In real GPU implementations, this can be achieved via opcode extensions. Our experiments showed that a phase distance of 512 covers more than 98% of all kernels[2]. Considering instruction size of 8 bytes and L1 cache line size of 128 bytes, adding phase distance would increase the static instruction size by 12.5%.

Two Level Schedulers: In the baseline two level scheduler (refer to Fig. 3 and subsection II-B2), when a warp in the ready queue arrives at a long latency operation, it is put in the active queue and replaced by the warp from the head of the queue. This implementation works well if priority of the warps in the active queue is rotated in a round robin order. However, notice that this is an issue for any scheme that requires warps to be prioritized. If a single queue is used to store the active warps, as well as the warps waiting on long latency operations, all the warps would need to be checked when replacing a warp from the ready queue. To solve this, we use an additional pending queue to store warps that are waiting on long latency instructions.

Fig. 8 shows a block diagram of our implementation of the two level scheduler. A warp waiting on a long latency instruction is first moved to the pending queue. When all the long latency instructions complete, it is moved to the tail of the active queue. The priority assignment logic is triggered at this point and warps in the active queue are sorted. This design effectively allows implementation of different scheduling policies by modifying the policy of the priority assignment logic. The RR scheduler does not sort the warps, while the GTO scheduler sorts the warps in launch order. The phase-aware scheduler uses the phase distance of the next phase to sort the warps. Consequently, contrary to the single level scheduler, length information is required only once for an entire phase. Our experiments show that adding phase length increases the static instruction size by less than 1%. After sorting, warp at the head of the active queue is moved to the ready queue.

[2]For longer phases, multiple opcode extensions can be used.

TABLE I: GPGPU-Sim configuration

Chip configuration	
Number of cores	16
Core frequency	1300 MHz
DRAM clock frequency	1850 MHz
Peak SP / DP floating point throughput	1330 / 650 GFLOPs
Peak DRAM bandwidth	177 GB/sec
Core configuration	
Maximum thread blocks per core	8
Maximum warps supported per core	48
Execution units per core	32 ALUs, 4 SFUs 16 LD/ST units
Scheduler configuration	
Warp schedulers per core	2
Instruction dispatch throughput per scheduler	1 instruction every 2 cycles
Ready warps queue size	6

V. Experimental Results

A. Methodology

We configured the simulator [12] to match the architecture of NVIDIA Tesla M2090 GPU [15] (refer to Table I). To perform our evaluations we chose kernels from the CUDA SDK [13] and the Rodinia benchmark suites [14]. The SDK has 49 applications, while Rodinia has 19; with each application having multiple kernels. We pruned our workload list by omitting the applications provided in the SDK for hardware profiling and demonstrating interoperability with graphics APIs. We also omitted kernels that did not have a grid size large enough to fill all the cores, and the size could not be increased without significantly changing the application code. Our final workload list had 13 kernels from 11 applications of the SDK and 17 kernels from 12 applications of Rodinia. For brevity, we discuss results of kernels for which either the RR or the GTO scheduling policy showed a better performance. For the remaining kernels, the GTO, RR and phase-aware scheduling policies had comparable performance (within 1% of each other).

B. Impact on Performance

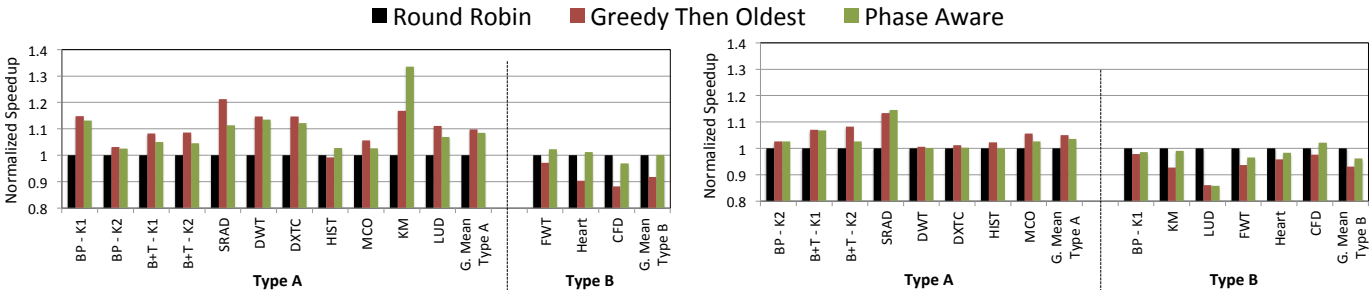
Fig. 9 plots the speedup of the Greedy Then Oldest (GTO) and phase-aware schedulers normalized to the Round Robin (RR) scheduler. The single level schedulers have an advantage of selecting from all the warps on the core. Hence, we compare the performance of the single level and two level schedulers separately. We have grouped the kernels into two types. For kernels grouped under **Type A**, the GTO scheduler achieves a better performance compared to RR, while for kernels grouped under **Type B**, the RR scheduler performs better than GTO.

1) Type A Kernels: Kernels for which the GTO policy performs better (grouped under type A), have phases of extremely short length. As mentioned in subsection III-C1, our simulations showed that when the RR policy is used, warps get accumulated in these short phases at around the same time, causing the core to become idle.

Eight kernels always perform better with the GTO policy (refer to Fig. 9(a)). The BP-K2, DWT, HIST and MCO kernels have their shortest phase (refer to Table II) at the beginning

Name	Applications from Rodinia									Applications from CUDA SDK				
	Back Propagation		B+Tree Search		Heart Wall	K-means Clustering	LU Decom-position	Speckle Reducing Anisotropic Diff.	Comp.Fluid Dynamics	Discrete Wavelet Transform	DXT Compression	Fast Walsh Transform	Histogram	Monte Carlo Option Pricing
Abbreviation	BP - K1	BP - K2	B+T - K1	B+T - K2	Heart	KM	LUD	SRAD	CFD	DWT	DXTC	FWT	HIST	MCO
Total Thread Blocks	65535	65535	65535	65355	56	841	16129	16384	1817	4096	4096	4096	256	2048
Threads / Block	256	256	256	256	256	256	256	256	192	512	256	512	192	256
Thread Blocks / Core	6	5	5	6	4	6	6	6	3	3	4	3	6	5
Total Phases	5	5	16	11	45	2	3	4	9	2	3	2	2	2
Longest Phase	206	99	40	32	212	48	150	218	985	144	1159	174	44	284
Shortest Phase	26	14	5	3	41	14	7	31	86	44	24	45	24	38

TABLE II: Workloads from the CUDA SDK [13] and Rodinia [14]



(a) Single Level Schedulers

(b) Two Level Schedulers

Fig. 9: Performance comparison. **Type A** - GTO performs better than RR. **Type B** - RR performs better than GTO.

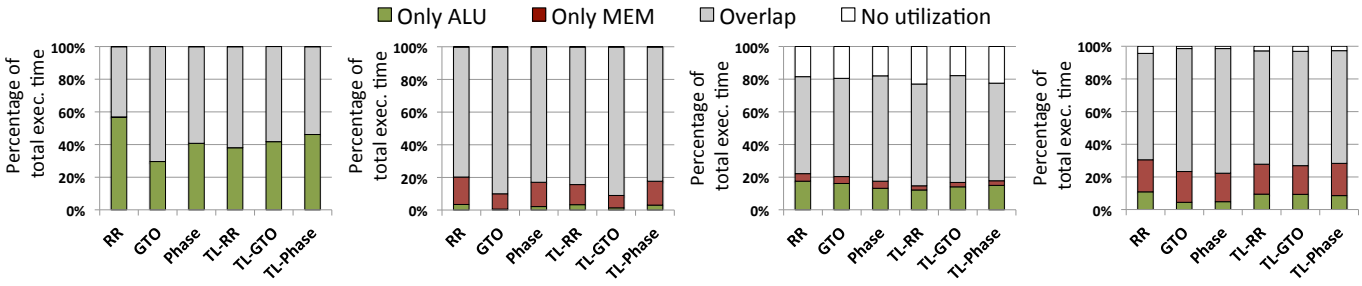
of the kernel. The SRAD kernel has two short phases in the middle of the kernel code, while the B+Tree kernels have phases of length shorter than 20 cycles intermixed throughout the code. Notice in Fig. 9 that the BP-K1, LUD and KM kernels have a better performance with the GTO policy for single level implementation and with the RR policy for two level implementation. This happens due to a phenomenon we refer to as intra-block tail effect, which we will explain in detail in subsection V-E.

For the type A kernels, the single level GTO has a speedup of 10% over single level RR on average. The performance of the phase-aware scheduler is close to that of GTO for all these kernels and it achieves a speedup of 9% on average over RR. A similar performance trend was observed for the two level schedulers. However, as warps progress through the kernel in fetch groups, all the warps do not arrive at the short phases simultaneously. This reduces the negative effect on performance of the RR scheduler. Notice in Fig. 9(b) that for the DWT, DXTC and HIST kernels RR scheduling now performs comparable to GTO and phase-aware. Consequently, the speedup of GTO over RR is lesser with an average of 4%. Again, the performance of the phase-aware scheduler is close to that of GTO and it achieves a speedup of 3% on average over RR.

2) Type B Kernels: Kernels for which the RR policy performs better (grouped under type B) typically have extremely long length phases. Heartwall is a very long kernel with 1523 instructions and 45 phases, with several long length phases of over 100 cycles. CFD has pairs of medium and long length phases, with an average phase length of 290 cycles, and FWT has just two phases of length 45 and 174. As discussed in subsection III-C2, memory operations issued at the beginning of these long phases complete before the phase ends. Consequently, the GTO policy keeps prioritizing a small set of warps causing it to under-utilize the memory bandwidth.

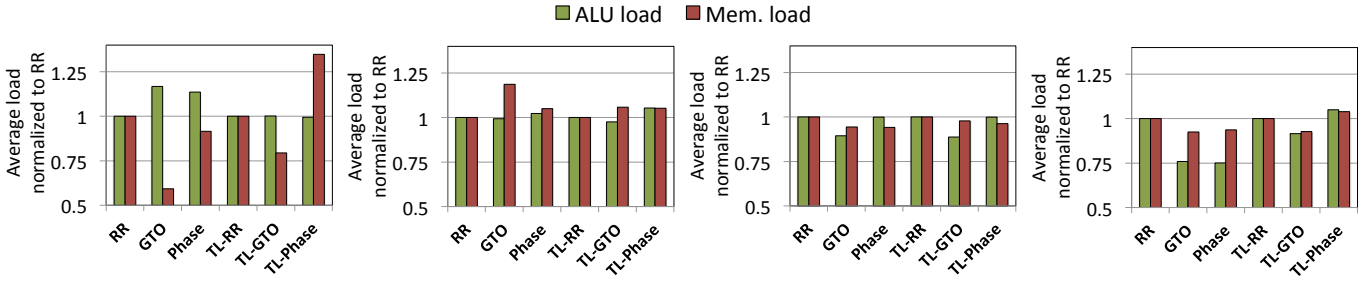
For the kernels in our benchmark suite, RR scheduling performs better than GTO only for three kernels for the single level implementation and has an average speedup of 9.2% over GTO. Notice in Fig. 9(a) that the phase-aware scheduler now performs closer to the RR policy. Performance of the phase-aware scheduler is comparable to RR and it achieves a speedup of 9% over GTO. For the two level implementation, similar to the type A kernels, the impact of phase behavior on performance (now on the GTO policy) reduces. The RR policy achieves a better performance for six kernels, with an average speedup of 7% over GTO. The phase-aware scheduler matches the performance of RR for all kernels except LUD and achieves a speedup of 4% over GTO. The reason for the negative impact on performance of the LUD kernel is explained in subsection V-E.

3) Summary of Performance Impact: Observe in Fig. 9(a) and Fig. 9(b), that for kernels grouped under type A, performance of the phase-aware scheduler is always closer to that of the GTO scheduler. On average, its performance is within 99% of GTO for both the single level and two level implementations, and it achieves a speedup of 9% and 3% respectively over RR. For the kernels grouped under type B, the phase-aware scheduler now performs closer to RR. It matches the performance of RR for the single level implementation and performs within 96% for the two level implementation. Consequently, it achieves a speedup of 8.9% and 4% over the GTO policy. These results indicate that application phase behavior has an impact on performance of the GTO and RR scheduling policies. As application type is not known a priory, performance can be negatively impacted if static policies are used. For example, if the RR policy is used, we would incur performance loss for type-A kernels, and similarly for type-B if the policy was GTO. On the contrary, the phase-aware policy always performs closer to the better performing policy for any kernel. Hence, by utilizing information regarding program phases, our scheduling policy becomes applicable to a wider range of applications.



(a) BP - K1 (Type A - Comp. intensive) (b) B+T - K2 (Type A - Mem. intensive) (c) Heartwall (Type B - Comp. intensive) (d) CFD (Type B - Mem. intensive)

Fig. 10: Breakdown of execution time for two kernels of type A and type B.



(a) BP - K1 (Type A - Comp. intensive) (b) B+T - K2 (Type A - Mem. intensive) (c) Heartwall (Type B - Comp. intensive) (d) CFD (Type B - Mem. intensive)

Fig. 11: Average ALU and memory load of two kernels of type A and type B.

C. Impact on Scheduler Idle Time

At any given cycle during kernel execution, the scheduler can be in either one of the four states: 1-no warps have a new instruction, 2-all warps that have an instruction are not ready (waiting for an operand from a previous instruction), 3-all warps that have a ready instruction cannot issue due to a pipeline stall or 4-at least one warp can issue an instruction. We refer to the sum of the cycles spent in the first three states as scheduler idle time. As scheduler idle time is the cycles when no new instructions are issued, it is a good indicator of the performance trend.

Fig. 12 plots the scheduler idle time as a percentage of the total execution time. For brevity, the data is averaged across kernels of the same type and normalized to RR. For type A kernels, the GTO and phase-aware policies had a 21.56% and 21.03% lower idle time compared to RR for the single level scheduler, while the average idle time was comparable to RR for the two level scheduler. As expected, for the kernels grouped under type B, the average scheduler idle time for GTO was higher than the RR policy. It was higher by 13.66% for the single level and by 8.73% for the two level scheduler. As the phase-aware scheduler performs closer to RR, its idle time was lower than GTO, but higher than RR by 6.8% for the single level and 3.9% for the two level. Notice that similar to the performance trend, the difference in the idle time between schedulers for a given type of kernels is lower in the two level implementations. This again shows that the impact of phase behavior is reduced with two level scheduling. Also, the idle time for the phase-aware warp scheduler is always closer to the better performing policy for each kernel type.

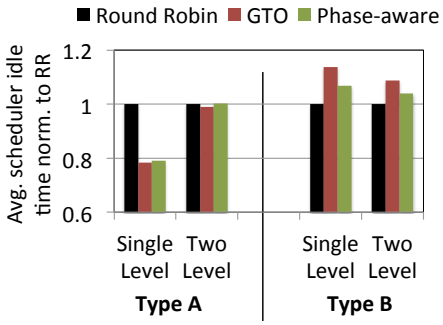
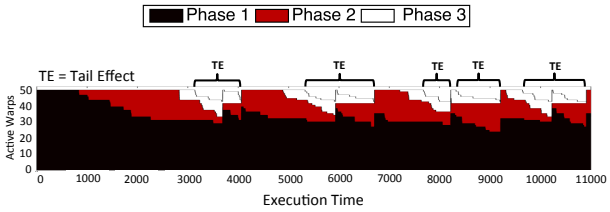


Fig. 12: Comparison of average scheduler idle time.

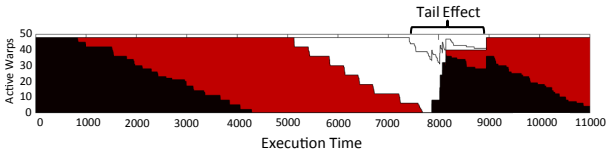
D. Impact on Functional Unit Load

The total runtime of a kernel can be broken down as 'Idle cycles' + 'Total Computation Cycles' + 'Total Memory Access Cycles' − 'Cycles of Overlap of Computation and Memory Access'. The amount of time spent by a kernel doing only arithmetic or only memory operations indicates whether the kernel is compute or memory intensive. To measure this, we instrumented the simulator and monitored the ALU pipeline and the memory system. The kernel is considered as performing computation (or memory access), if there is at least one ALU (or memory) instruction in flight at that cycle. Load on the ALU and memory pipelines is defined as the total number of instructions that are in flight during that cycle. Fig. 10 shows the breakdown of execution time of two kernels each, of type A and type B, and Fig. 11 plots their respective ALU and memory loads.

BP-K1 is an example of a type A kernel that is compute intensive. A significant portion of the execution time is spent



(a) Two-level GTO



(b) Two-level Round-robin

Fig. 13: Intra-block tail effect of the LUD kernel.

performing only ALU operations (refer to Fig. 10(a)). Notice in Fig. 11(a) that the single level GTO and phase-aware schedulers maintain a lower load on the memory subsystem. However, as the average ALU load achieved is higher, they perform better than RR. Also, notice that for the two level implementation, the RR scheduler maintains a ALU load that is similar to the GTO and phase-aware schedulers. Consequently, the GTO and phase-aware scheduler do not achieve a speedup over RR for the two level implementation (refer to Fig. 9(b)). B+Tree-K2 is a type A kernel that is memory intensive (refer to Fig. 10(b). Again, observe in Fig. 11(b) that although the ALU load is comparable for all the schedulers, the GTO and phase-aware schedulers maintain a higher memory load, and consequently achieve a better performance compared to RR.

Similarly, Figs. 10(c) and 10(d) are examples of type B kernels (RR performs better than GTO), which are compute and memory intensive respectively. Around 20% of the execution time of Heartwall is spend performing ALU operations by all schedulers (refer to Fig. 10(c)). Notice in Fig. 11(c) that the average ALU load achieved by the GTO policy is lower than that of RR and phase-aware schedulers, thereby achieving a lower performance. CFD is a type B kernel that is memory intensive. The GTO scheduler achieves a lower average memory load for both the single and two level implementations, and hence achieves a lower performance as compared to RR. Notice that the single level phase-aware scheduler maintains a lower memory load compared to RR, while the two level implementation achieves a slightly higher load. This is reflected in the performance results (refer to Fig. 9), as the single level phase-aware scheduler is 3% slower than RR, while the two level achieves a speedup.

E. Impact of Intra-Block Tail Effect on Performance

As mentioned in Sect. V-B, performance of the two level GTO and phase-aware policies, for the LUD, KM and BP-K1 kernels, is adversely affected due to intra-block tail effect. Intra-block tail effect is a phenomenon where some warps of a thread block complete the kernel before others. This happens when some warps from a thread block complete their long latency instructions before others and are selected in an earlier fetch group. The LUD, KM and BP-K1 kernels have a short phase of length 7, 14 and 16 respectively at the end. The warps selected in the earlier fetch groups run ahead and complete the last phase. However, a new thread block is not launched until all warps of a block complete, thus reducing the number of warps on the core. The effect is reduced in the single level schedulers as warps are not scheduled in fetch groups. Moreover, with single level RR, all warps arrive at the last short phase at the same time leading to the GTO and phase-aware policies performing better (refer to Fig.9(a)).

Fig. 13 depicts the intra-block tail effect for the LUD kernel by plotting the total number of active warps on a core. Notice in Fig. 13(a) that 6 warps (size of the fetch group) finish all the three phases and the number of warps reduce to 42 (the first marked TE). However, new warps are not launched until more warps finish phase 3. Notice that the tail effect repeats each time warps in phase 3 complete. On the contrary, as the RR scheduler switches to a different fetch group after each phase (refer to Fig. 13(b)), all warps reach phase 3 at around the same time. Notice that the length of the tail for RR is around 2000 cycles for all the 6 blocks, as compared to 5000 cycles for GTO (sum of all TEs in Fig. 13(a)). Performance of the phase-aware scheduler is also affected as it chooses the shortest length phase which is the last phase for these kernels. To mitigate this problem, we are currently evaluating implementations for adding thread-block affinity to our two level scheduler. Note that this would increase the complexity of the scheduler considerably and its overhead for the hardware implementation should be carefully considered.

VI. Related Work

A. General Warp Scheduling Techniques

Lakshminarayana and Kim [16] analyzed the performance of GPU kernels under various instruction fetch and memory scheduling policies. They showed that applications in which warps have a uniform execution latency, a fairness based warp and DRAM access scheduling approach results in the best performance. The goal of their work was to explore the effects of the scheduling policies on the performance of various applications. In contrast, we try to use information about the application to make scheduling decisions at runtime and perform closer to the better performing policy for each application.

Gebhart et al. [7] and Narasiman et al. [8] proposed the hierarchical warp scheduler that we used as a baseline for our work. The focus of authors in [7] was making the warp scheduling logic simpler and more energy efficient. Choosing from a smaller set of warps every cycle saves energy spent on scheduling. Their results show that using an active queue size of 6 results in less than 1% performance loss compared to the single level scheduler. The focus of authors in [8] was to use the two level policy to improve latency hiding. As the two level policy makes the warps progress through the kernel instructions at different speeds, it results in a better overlap of memory latency and computation. We showed that although the two level schedulers perform well, the policy to select the

fetch group has an effect on their performance. We build upon the schedulers proposed in their works, and make them more robust to application phase behavior.

Our work is closest to the work published by Chen et al. in [17]. They identify the adverse effects of short phases on the two level RR scheduler and propose a scheduler that shifts to the next fetch group only at priority shift instructions. Short phases are identified by the compiler and merged with the previous phase by adding a priority shift instruction after the short phases. This effectively makes the priority selection policy as greedy until all the short phases are completed by a fetch group. Our policy to select the phase with the shortest length has the same result. The problem they identify with the RR scheduler can be mitigated if the GTO policy is always used. Instead, our work identifies the scenarios when the GTO policy has a lower performance as well, and proposes a more robust scheduling policy that would be applicable to a wider range of application types.

B. Scheduling Techniques to Mitigate Warp Divergence

In addition to the generic techniques mentioned in the previous subsection, there is a significant amount of published work that has focused on techniques to mitigate warp divergence. Warp divergence occurs when threads within a warp execute different program paths due to branches. Traditional GPUs execute each path sequentially with lesser number of threads, thereby under-utilizing the GPU core's computational throughput. Fung et al. [18] proposed Dynamic Warp Formation (DWF) which creates new warps from threads that fall on the same program path after the divergence point. Their technique suspends a warp when it reaches an instruction that causes threads in a warp to diverge. When other warps arrive at the same instruction, new warps are created and all warps proceed from that point. To reduce the synchronization overhead due to warps from different thread blocks forming a warp, Fung et al. [19] later proposed Thread Block Compaction (TBC), which adds thread block affinity to DWF. TBC creates new warps from diverged warps of the same thread block, similar to the Large Warps technique proposed in [8].

Meng et al. [20] proposed Dynamic Warp Subdivision (DWS) which divides a warp into warp splits on branch divergence. The focus of DWS is to improve latency hiding in scenarios when one warp split encounters a cache miss. The latency of memory access can be overlapped by executing the other warp split. Their technique creates warp splits to improve latency hiding upon memory divergence as well. Memory divergence occurs when some threads of a warp hit in the cache and others miss. The DWS technique creates a warp split with threads that hit and lets them continue. This allows those threads to reach the next memory request and possibly prefetch for the warp split that missed the first memory request. A similar technique of dividing warps into separate scheduling entities was proposed by Steffen and Zambreno [21]. They divide the kernel instructions, which are potential program paths after divergence, into smaller instruction blocks called μ-kernels. On warp divergence, threads wait in a partial warp pool. When there are enough threads to make a complete warp, the new warp is allowed to execute the μ-kernel.

Our phase scheduler can be used along with the warp divergence techniques mentioned above. Our technique marks each basic block as a new phase. Thus on warp divergence warps would be first put into the pending pool. Each of the techniques mentioned above creates new scheduling entities at this point. The new entities can then be scheduled by our phase scheduler using the phase length information.

C. Thread Throttling

There is body of work that focuses on throttling the amount of thread level parallelism available on the core. Although all of these are not at the warp level, changing the number of active warps on a core has an effect on warp scheduler performance. Guz et al. [22] designed an analytical model to study the impact of amount of TLP on performance of highly multi-threaded architectures. They showed that performance increases initially due to increase in TLP, and then starts to reduce once the total working set of the threads does not fit in the cache. Performance continues to decrease if more threads are launched until the TLP is high enough to hide the increased memory latency. The region of performance dip is referred to as the "performance valley". The authors in [9, 11, 23, 24] effectively detect at runtime when the number of active threads causes the GPU to get into the performance valley.

Rogers et al. [9] detect scenarios when the L1 data cache is trashed. They monitor the cache lines to detect warps that have lost intra-warp locality because of other warps evicting data that would have been used by them. A scoring system increases the score of such warps. Warps below a certain score are not selected by the scheduler, thereby reducing the number of active warps. Kayiran et al. [23] monitor the amount of time spent waiting for memory. The number of threads is reduced if the time is more than an empirically found threshold. Lee et al. [11] find the *optimal* amount of TLP by launching the maximum number of warps initially and then using a greedy scheduling policy. After the first thread block completes, the optimal number of blocks is estimated using the number of instructions that have been completed until then. Awatramani et al. [24] detect the optimal thread block count by comparing the pipeline stalls at different block counts. They launch half the maximum number of warps and then use history information of the previous block counts to guide the scheduler. Lee et al. [10] identify a problem similar to the tail effect mentioned in Sect. V-E for workloads that have varying warps execution times. They throttle warp execution by assigning a time slice to each warp, proportional to its the execution time with the RR scheduler. The tail effect is reduced by giving a larger time slice to longer running warps.

Each of the above techniques reduces the number of warps on a core to a smaller set. However, the underlying policy to select warps from this reduced set to dispatch to the execution units is either RR or GTO. Hence, our phase-aware warp scheduler can be easily used in conjunction with the above-mentioned techniques.

VII. Conclusion

In this work we analyze phases in GPGPU kernels and show that the performance of warp schedulers depends on characteristics of these phases. Using real-world application examples, we show that an efficient warp scheduling policy can be designed by understanding how warps progress through these kernel phases. Based on these observations, we propose a novel phase-aware warp scheduling policy that uses information provided by the compiler to make scheduling decisions. We implement this scheduler in a GPU simulator and demonstrate that it is more robust to phase behavior as compared to the static policies like round robin and GTO. As more and more applications are being ported to the GPU for acceleration, we believe that to cater to the wide array of workloads, application-aware warp scheduling policies will become even more relevant in the near future.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under awards CNS-1116810 and CCF-1149539.

References

[1] *The Top 500 Supercomputers List*, June 2015. [Online]. Available: http://www.top500.org/lists/2015/06

[2] *The NVIDIA Tesla K80 GPU Architecture for Servers*, 2014. [Online]. Available: http://www.nvidia.com/object/tesla-servers.html

[3] *AMD FirePro S10000 Server Graphics*, 2014. [Online]. Available: http://www.amd.com/en-us/products/graphics/workstation

[4] *The Intel Xeon Processor E7-4890 v2*, 2014. [Online]. Available: http://ark.intel.com/products/75251/

[5] NVIDIA, *The CUDA C Programming Guide*, 2013.

[6] A. Munsi, *The OpenCL C 2.0 Specification*, 2013.

[7] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors," *ACM Trans. Computer Syst.*, 2012.

[8] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. of the 44th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2011.

[9] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. of the 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2012.

[10] S. Lee, and C. Wu, "CAWS: Criticality Aware Warp Scheduling for GPGPU Workloads," in *Proc. of the 23rd Int. Conf. on Parallel Architectures and Compilation Techniques*, 2014.

[11] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *IEEE 20th Int. Symp. on High Performance Computer Architecture (HPCA)*, 2014.

[12] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. of the IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[13] *NVIDIA CUDA SDK*. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Sokadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE Int. Symp. on Workload Characterization*, 2009.

[15] P. Glaskowsky, *Next Generation CUDA Compute Architecture: Fermi*, 2010.

[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. Chen, X. Tao, Z. Yang, J.-K. Peir, X. Li, and S.-L. Lu, "Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency," in *IEEE 27th Int. Symp. on Parallel & Distributed Processing (IPDPS)*, 2013.

[18] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware," *ACM Trans. Archit. Code Optim.*, 2009.

[19] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proc. of the 2011 IEEE Int. Symp. on High Performance Computer Architecture*, 2011.

[20] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *Proc. of the 37th Annual Int. Symp. on Computer Architecture*, 2010.

[21] M. Steffen and J. Zambreno, "Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels," in *Proc. of the 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2010.

[22] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. Many-Thread Machines: Stay Away From the Valley," *IEEE Computer Architecture Letters*, 2009.

[23] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proc. of the 22nd Int. Conf. on Parallel Architectures and Compilation Techniques*, 2013.

[24] M. Awatramani, D. Rover, and J. Zambreno, "Perf-Sat: Runtime Detection of Performance Saturation for GPGPU Applications," in *Proc. of the Int. Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)*, 2014.