2015 IEEE 29th International Parallel and Distributed Processing Symposium

PRO: Progress Aware GPU Warp Scheduling Algorithm

Jayvant Anantpur
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
jayvant@hpc.serc.iisc.ernet.in

R. Govindarajan
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore, India
govind@serc.iisc.ernet.in

***Abstract*—Graphics Processing Units (GPUs) contain multiple SIMD cores and each core can run a large number of threads concurrently. Threads in a core are scheduled and executed in fixed sized groups, called warps. Each core contains one or more warp schedulers that select and execute warps from a pool of ready warps. In spite of having a large number of concurrent warps - 48 on NVIDIA Fermi architecture GPU - on many GPGPU applications, current warp scheduling algorithms can not effectively utilize the hardware resources, resulting in stall cycles and loss in performance. The main reason for this is current warp scheduling algorithms mostly focus on long latency operations, especially global memory accesses, and do not take into account factors such as the progress of each thread block and the number of ready warps.**

In this paper, we propose, *PRO*, a progress warp scheduling algorithm that not only focuses on finishing individual thread blocks faster but also on reducing the overall execution time. These goals are achieved by dynamically prioritizing thread blocks and warps, based on their progress. We implemented our proposed algorithm in the GPGPU-SIM simulator and evaluated on various applications from GPGPU-SIM, Rodinia and CUDA SDK benchmark suites. We achieved an average speedup of 1.12x and a maximum speedup of 1.94x over the commonly used Loose Round Robin warp scheduling algorithm. Over the Two Level warp scheduler, our algorithm showed an average speedup of 1.13x and a maximum speedup of 1.6x. Our proposed solution requires only a very small increase in the GPU hardware.

***Index Terms*—GPUs; Warp Scheduling; CUDA; OpenCL;**

I. Introduction

Graphics Processing Units are being used in general purpose computing especially to accelerate data parallel code. Programming languages such as CUDA [4] and OpenCL [15] have been the catalyst in the growth of GPUs in general purpose computing. These programming models use Single Instruction Multiple Threads (SIMT) computation model [4]. In this model a large number of threads run in parallel on Single Instruction Multiple Data (SIMD) cores using hardware multi-threading. Current GPU architectures consist of multiple cores, known as Streaming Multiprocessors (SMs).[1]. Each SM has a register file, L1 cache and software managed shared memory. For example, NVIDIA Fermi GPU [6] has 16 SMs, each SM containing 32K 4 byte registers, 64 KB configurable shared memory and L1 cache.

[1]We use NVIDIA GPU terminology

Programming models such as CUDA, OpenCL, etc., provide extensions to programming languages such as C and C++, to define and invoke functions - called kernels - which are to be executed on a GPU. Typically a thread running on a CPU invokes a kernel with an execution configuration that describes a hierarchical structure of threads called grid. A grid defines the size of a thread block and number of thread blocks. A thread block (TB) is a group of threads that can cooperate with each other using shared memory and barrier synchronization. Threads from different thread blocks execute independently and in any order, following a relaxed consistency model.

When a kernel is invoked, a global work distribution engine (Thread Block Scheduler) in the GPU schedules thread blocks to all SMs of the GPU. The number of thread blocks that can reside on an SM depends on the resources - registers and shared memory - used by a TB and also the maximum number of threads and thread blocks that can concurrently reside on it. For example, in NVIDIA Fermi GPU [6], at most 1536 threads or 8 thread blocks can reside on an SM. When a kernel is invoked, the Thread Block Scheduler will assign as many TBs to SMs as allowed by resource constraints and the remaining TBs are assigned one at a time to an SM as and when a previously assigned TB finishes. One thing to note is, the unit of work allocation to an SM is a TB and a TB can be assigned to an SM only if the SM has enough resources for it.

The threads of a thread block are further partitioned into groups of consecutive threads. In CUDA terminology this group is called a warp and the size of a warp is 32 threads. Each SM contains one warp scheduler and every cycle the warp scheduler schedules one of the ready warps and issues the next instruction to the ALUs, load/store units or Special Function Units. In practice, an SM may contain multiple warp schedulers (e.g. two in NVIDIA Fermi) and each warp scheduler can potentially schedule one warp every cycle depending on the availability of hardware resources. The goal of a warp scheduler is to schedule warps in such a way that it can hide long execution latencies of various instructions effectively and reduce the overall execution time. The ability of a warp scheduler to issue a warp every cycle and avoid stall cycles, largely depends on the number of ready warps. An algorithm like Loose Round Robin(LRR), assigns equal priority to each warp and hence all warps make more or less equal progress

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.26

and reach long latency instructions close to each other in time. Since each SM core issues instructions of a warp *in order*, if the current instruction of a warp cannot be issued for any reason such as, an instruction or data cache miss, operand values not available or hardware resources busy, etc., then the warp is not ready for scheduling. With fewer ready warps for scheduling, a warp scheduler cannot hide long latencies effectively.

CUDA and OpenCL support a barrier synchronization statement which guarantees that the statement after it will be executed by a thread of a TB only when all threads of the TB have executed the barrier statement. When a warp reaches a barrier instruction, it has to stall for all sibling warps i.e., warps from the same TB, to reach the barrier instruction before it can be considered for scheduling by the warp scheduler. This means, as warps reach barrier statements, the number of warps available for scheduling goes down and hence the warp scheduler may not have enough warps to hide long execution latencies. Also, when a warp finishes executing a kernel, its resources cannot be released till all its sibling warps finish executing, at which point the TB is considered to be finished executing the kernel and it is deallocated i.e., its resources are released and a new TB is assigned to the SM. Due to various reasons such as unequal amounts of work, cache misses, etc., warps of a TB can take different amounts of time to reach barrier statements or finish. This is termed as warp-level divergence [16]. Current warp level schedulers such as Loose Round Robin(LRR), Two-Level(TL)[22], etc., do not take any special actions to handle warp-level divergence.

Long latency instructions, warp level divergence and work allocation at thread block level decrease the number of ready warps, reducing the ability of warp schedulers to hide stall cycles.

In this paper we propose a simple and intuitive warp scheduling algorithm, *PRO*, which takes a collective look at the problems due to long latency operations, warp-level divergence and thread block as the unit of allocation and deallocation of threads. Our solution prioritizes TBs and warps based on the progress they have made so far, so that they reach long latency instructions such as global memory loads, at different times and hence their long latencies can be hidden in a better way [22]. Our approach also reduces the effect of barrier statements, by prioritizing TBs that have warps waiting for their sibling warps at the barrier statements. Further, it prioritizes TBs with warps that have finished executing, so that those TBs can finish executing quickly and make way for TBs waiting at the GPU level Thread Block Scheduler. We achieved a geometric mean speedup of 1.13x over TL and 1.12x over LRR warp scheduling algorithms.

To the best of our knowledge, our work is the first to use the concept of thread block and warp progress in a warp level scheduler to improve GPU runtime performance.

II. Motivation

In this section we will discuss the three main reasons for stalls and how they affect the runtimes of kernels.

A. Long Latency Instructions

In case of a scheduling algorithm like LRR, all warps and TBs are given equal priority and hence there is a good chance that they all will make more or less equal progress. Due to this all warps in an SM reach long latency instructions such as global memory loads close to each other in time and hence they all will stall for their memory requests to be serviced which can take several hundreds of cycles. This may cause the SM to stall due to unavailability of ready warps. Narasiman et al. [22] proposed a solution to this problem, which divides the warps in an SM into *fetch groups*, and schedules the fetch groups and warps in each fetch group in round robin fashion. This causes each fetch group to reach long latency instructions at different points in time [22]. Instead of fetch groups, we prioritize TBs and warps in each TB based on the progress they have made.

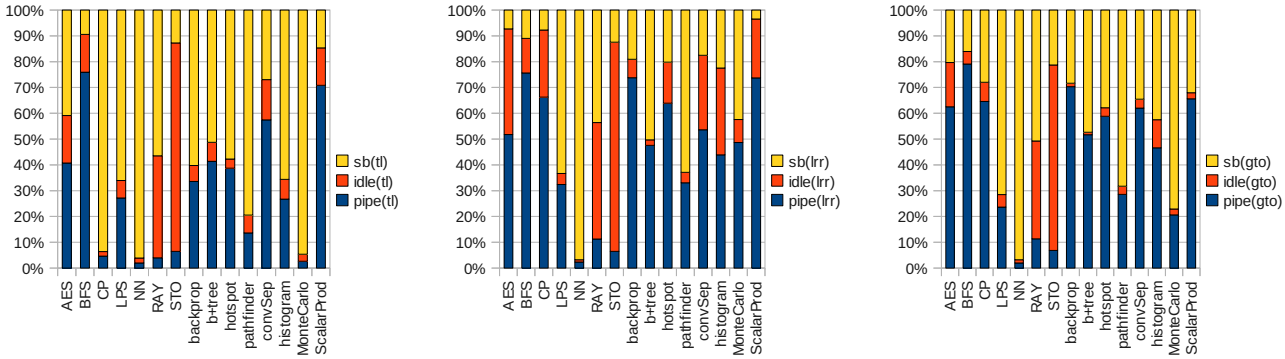
B. Warp-level Divergence

Some applications exhibit a lot of warp-level divergence [16]. Warp-level divergence occurs due to various reasons e.g., different amounts of work, cache misses, thread divergence, etc. This means, in addition to finishing at different points in time, warps of a TB may reach barrier statements also at different points in time. A warp scheduler oblivious to both these situations will not schedule the right set of warps, increasing the amounts of time warps wait for their sibling warps and consequently suffering from more stalls. Our proposal to reduce the waiting times of warps is to prioritize a TB as soon as one of its warps reaches a barrier statement or finishes execution.

Figure 1 shows the contribution of different types of stalls experienced in three existing warp scheduling methods, viz., TL, LRR and GTO (Greedy Then Old) obtained using GPGPU-Sim [1]. Refer to section IV for the details on simulation configuration and benchmarks. GPGPU-Sim classifies the stalls as below. If in a cycle, all available warps are issued and no warp is ready to issue then the cycle is considered to be an Idle cycle. Reasons such as warps waiting at barrier, empty instruction buffer, branch divergence, etc., cause Idle cycles. If there is at least one warp with a valid instruction but none has all its input operands ready, then the cycle is counted as a Scoreboard stall cycle. The third type of stall (Pipeline stall) happens when all the execution pipelines are full and in spite of having valid instructions with ready operands, no warp can be issued. It can be seen that LRR has the highest percentage of Idle stalls. Warp-level divergence and work allocation at TB level, together contribute to an increase in Idle stalls. Our proposed algorithm reduces Idle stalls considerably by dynamically prioritizing TBs and warps.

C. SM Residency

As discussed in Section I, the number of TBs resident in a GPU depends on the resources used by each TB. If a kernel is invoked with more TBs than that can reside concurrently on the GPU, the remaining TBs are assigned to SMs one at a time as and when a previously assigned TB finishes execution.



(a) TL stalls (b) LRR stalls (c) GTO stalls

Fig. 1: Details of stalls in TL, LRR and GTO scheduling algorithms, *sb* refers to Scoreboard stalls, *idle* refers to Idle Stalls and *pipe* refers to Pipeline stalls.

Also threads are assigned to an SM at the granularity of a TB i.e., an entire TB is assigned to one SM. So, a warp scheduling algorithm oblivious of this fact may schedule warps in such a way that all the resident TBs on an SM finish close to each other in time, as a batch of TBs and hence the new TBs will only start executing together as a new batch. For example, consider a kernel with 192 TBs launched on a GPU with 16 SMs. Assuming each SM can accommodate 8 TBs at a time, initially 128 TBs will start executing. If these 128 TBs take similar amount time to finish then they all will finish close to each other in time. Hence, when the remaining 64 TBs start executing two things can happen: i) the hardware resources are only partially utilized for the entire duration of their run and ii) the number of ready warps per SM may not be enough to hide long latencies. Our solution allows some TBs to make more progress than others and hence finish earlier, which will enable the new TBs to start executing sooner and overlap their execution with older TBs. This increases the hardware resource utilization as well as hides long latencies.

Figure 2 shows execution of thread blocks on one SM for LRR and our proposed scheduling algorithm, PRO. Each bar shows the time duration for which a thread block is running. So, for example, with LRR algorithm, thread block 7 starts executing at simulation cycle 19042 and finishes at simulation cycle 32699. It is quite obvious from the figure that in case of LRR, thread blocks execute in batches and there is very little overlap between the execution of thread blocks from two different batches. In case of PRO, as the figure shows, when a new thread block starts executing on an SM, the resident thread blocks on the SM are in different phases of execution, i.e. with very different amount of progress. The figure also shows how PRO manipulates priorities of thread blocks to reduce the overall runtime. It shows reduction in runtimes of first 11 thread blocks and the last 2 thread blocks, compared to LRR. It also shows that some thread blocks take more time to finish compared to LRR but overall runtime reduces. For example, thread block 12, in spite of starting well before thread blocks 13-17, takes more simulation cycles and finishes at around the same time as them. Due to various reasons such as total number of thread blocks, differences in execution times, etc., each SM may not execute the same number of thread blocks.

III. PRO: Progress Aware Warp Scheduler

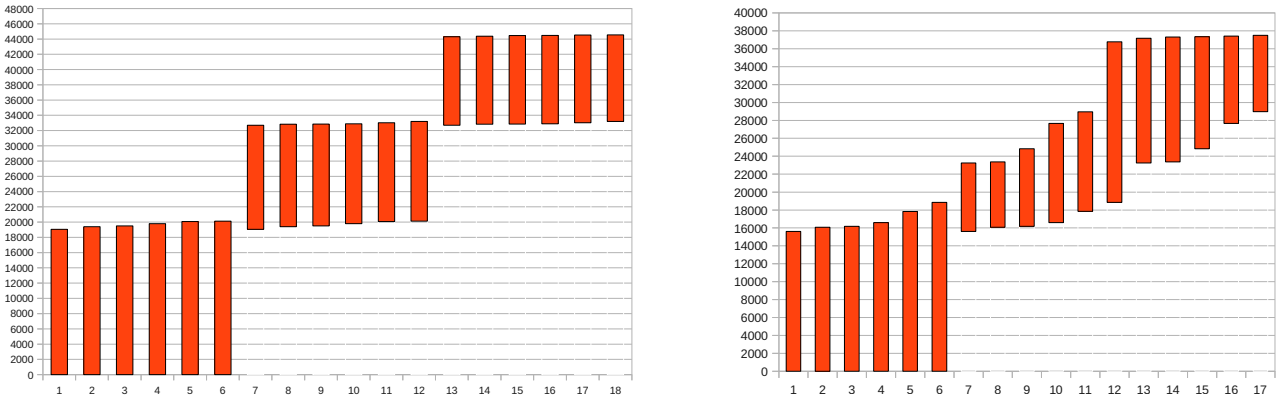
In this section we explain in detail our proposed solutions for warp-level divergence and stalls due to long latency operations.

First, we define terms used by our algorithm. The progress of a TB, *TBProgress*, is defined as the sum of number of instructions executed by all its threads. Similarly, *Warp-Progress* is the sum of number of instructions executed by the constituent threads of a warp. The execution of a kernel is divided into two phases:

- *fastTBPhase* is from the time a kernel starts executing to the time there are TBs waiting in the Thread Block Scheduler to be assigned to SMs, and
- *slowTBPhase* is from the time the Thread Block Scheduler assigns the last TB of a kernel to one of the SMs, to the completion of the kernel.

A TB is said to be in the *barrierWait* state if it has at least one warp waiting at a barrier statement. Similarly, a TB is said to be in the *finishWait* state if at least one of its warps has finished execution.

The primary design goal of PRO is to reduce the number of *stallCycles* which are the cycles in which no warp is scheduled. A warp scheduler can potentially reduce *stallCycles* if the number of ready warps is large. So, PRO tries to increase the number of ready warps. As discussed before, LRR causes a large number of warps to reach a long latency instruction close to each other in time and hence can not hide long latencies effectively. The solution proposed by the two level warp scheduling algorithm, *TL* [22], is to form groups of warps and achieve unequal progress among the groups so that when one group stalls, warps from other groups can be scheduled. It uses round robin algorithm among the groups and within each group. We propose to assign different priorities to TBs and also to warps within each TB.



(a) LRR (b) PRO

Fig. 2: Execution of Thread Blocks in LRR and PRO. X axis shows thread block indices and Y axis shows simulation cycles.

A. Thread Block and Warp Prioritization

Now, we explain why prioritizing TBs and warps helps. At a high level, a program execution can be divided into multiple phases and the phases can be characterized by their compute and memory intensities. Allowing different TBs to make unequal progress enables them to be in different phases, and hence a better mix of instruction execution latencies can be achieved. Even though, TL achieves some amount of unequal progress, since the warp groups are scheduled in a round robin way, they are more likely to be close to each other in program execution phases. There are two other reasons for prioritizing TBs. The first reason is that the unit of allocation of threads to SMs is a TB and so a new TB can be assigned to an SM only when one of the currently running TBs finishes completely. Allowing TBs to make unequal progress also enables new TBs to start executing sooner and achieve a good overlap of old and new TBs. The second reason is barrier statements which synchronize the execution of all threads of a TB. Prioritizing a TB that has warps waiting at a barrier statement allows the TB to cross the barrier quickly reducing its *barrierWait* period.

In addition to prioritizing TBs, our proposal is to prioritize warps of a TB to avoid all of them reaching long latency instructions close to each other in time. Dynamic prioritization of TBs and warps helps our technique reduce long latencies as well as waiting periods.

B. Thread Block States and Transitions

PRO classifies TBs into various states. In the *fastTBPhase*, a TB can be in one of 3 states viz., *barrierWait*, *finishWait* and *noWait*[2], where *noWait* state is the default state. When the kernel execution moves from *fastTBPhase* to the *slowTBPhase*, TBs in *noWait* and *finishWait* states are merged into a new state called *finishNoWait*, and TBs in *barrierWait* state are moved to *barrierWait1* state. State *barrierWait1* is created to enable transition to *finishNoWait* state when all warps of a TB reach the barrier statement. Figure 3 shows the state

[2]Unlike *barrierWait* and *finishWait*, the warps of a TB in *noWait* state do not have to wait for their sibling warps.

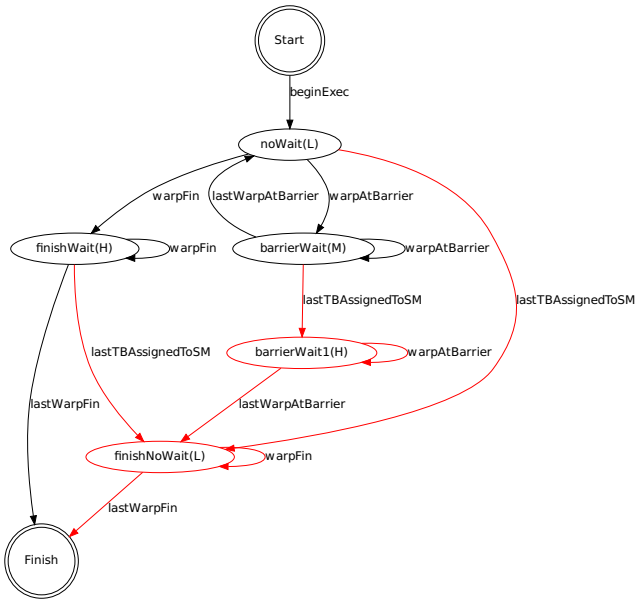


Fig. 3: Thread Block State Transition Diagram

transition diagram of a TB. Each TB starts in the *noWait* state. It goes to *barrierWait* state when any of its warps reaches a barrier statement. When all warps of the TB reach the barrier statement, the TB comes back to *noWait* state. Similarly, the TB transitions to *finishWait* state when any of its warps finishes executing. When all warps of a TB finish, the TB goes into the terminal *Finish* state. States *barrierWait1* and *finishNoWait*, shown in red colour belong to the *slowTBPhase*.

C. Scheduling Algorithm

Algorithm 1 shows the progress aware warp scheduling algorithm. A kernel begins execution in *fastTBPhase*. In this phase the scheduler gives the highest priority to TBs in

finishWait state followed by TBs in *barrierWait* state which are followed by TBs in *noWait* state. This is indicated by means of H(igh), M(edium) and L(ow) priorities in the figure. If TB *Ti* has higher priority than TB *Tj*, then the warps of *Ti* will have higher priority than the warps of *Tj*. Below we discuss, how priorities are assigned for TBs in each of these states.

1) *noWait* state Thread Block Prioritization

In *fastTBPhase* TBs in *noWait* state are prioritized in decreasing order of their progress, so that the highest priority TB is the one with the maximum progress. When two TBs have made the same amount of progress, they are prioritized based on their global indices. This way of prioritizing TBs gives some TBs more compute time and hence they finish faster. As mentioned before, finishing TBs faster allows new TBs to start sooner and achieve a good diversity of progress among the resident TBs. Since different TBs can potentially take different time to complete, we can assume that the time to completion of a TB is inversely proportional to its progress. Note that, a TB *Ti* which has collectively executed more instructions than TB *Tj*, but still may take more time to complete (compared to *Tj*), may get higher priority, although this is rare in practice. Alternatively one could use the number of instructions executed by a TB which has completed and use this to normalize progress across TBs. However even this is an approximation as different TBs can execute different numbers of instructions. Thus, a TB with the maximum amount of progress can be assumed to have the shortest remaining time. Hence our technique to prioritize faster TBs mimics the *Shortest Remaining Time First* job scheduling algorithm.

As shown in Algorithm 1, at regular intervals (after THRESHOLD number of cycles), TBs are sorted based on their progress (function *sortTBs*). Since, the number of resident TBs in an SM is very small - 8 in case of NVIDIA Fermi architecture GPUs [6]- and sorting is based only on the number of instructions executed by TBs so far, the time required to sort is at most a few tens of cycles, even with a single comparison per cycle. Sorting is not blocking in the sense, it can overlap with the execution of TBs. In our experiments we set THRESHOLD to 1000 cycles. In addition to sorting TBs, warps in each TB are also sorted in decreasing order of their progress (function *sortWarps*). The effect of prioritizing thread blocks in *noWait* state can be seen from the vastly different runtimes of thread blocks (1-11) in Figure 2(b).

2) *finishWait* state Thread Block Prioritization

As warps of a TB start finishing, till a new TB gets assigned to the SM, the number of ready warps available for scheduling goes down, reducing the ability of the warp scheduler to hide long latencies. When a warp of a TB in *noWait* state finishes, PRO takes the following steps:

- The TB state is changed from *noWait* to *finishWait* and since *finishWait* state TBs have higher priority than *noWait* state TBs, its priority increases to H from L. If there are multiple TBs in *finishWait* state, then they are sorted in decreasing order of the number of warps finished. This gives higher priority to the TB with more warps finished, enabling it to finish sooner. In case of a tie, TB with more instructions executed is given higher priority. Algorithm 1 achieves this using function *sortFinishWaitStateTBs*.
- Warps of the TB are sorted in increasing order of their progress. This gives highest priority to the warp that has made the least progress and lowest priority to the warp that has made the most progress, enabling the warps with less progress to get more compute time than warps with more progress.

These two steps together help finish a TB in *finishWait* state quickly. Procedure *insertFinishWarp* implements these two steps.

3) *barrierWait* state Thread Block Prioritization

In the *barrierWait* state, till all warps of the TB reach the barrier statement, the number of ready warps goes down. When a warp of a TB *Ti* in *noWait* state reaches a barrier statement, PRO takes the following steps:

- The TB state is changed from *noWait* to *barrierWait*. Since *barrierWait* state TBs have higher priority than *noWait* state TBs, *Ti*'s priority increases from L to M. If there are multiple TBs in *barrierWait* state, then they are sorted in decreasing order of the number of warps waiting at the barrier statement. This gives higher priority to a TB with more warps waiting at the barrier. In case of a tie, TB with higher progress is given higher priority. Algorithm 1 achieves this using function *sortBarrierWaitStateTBs*.
- Warps of the TB are sorted in increasing order of their progress. This gives the highest priority to the warp that has made the least progress and the lowest priority to the warp that has made the most progress.

Together these two steps help reduce the *barrierWait* period of the TB. Procedure *insertBarrierWarp* implements these two steps.

When a TB in *noWait* state goes to *finishWait* or *barrierWait* state, its warps are approximately in decreasing order of progress (depending on when they were sorted last and the execution latencies experienced by them since then, they may not remain in the same order) and in most cases simply reversing the order will achieve the required order. Similar to sorting TBs, sorting warps of a TB is a non-blocking step. In algorithm 1, function *sortWarps* is used to sort warps of a TB.

To summarize, in the *fastTBPhase* of execution, the three states are prioritized in the order *finishWait*(H), *barrierWait*(M) and then *noWait*(L). TBs in *finishWait* state are prioritized based on the number of warps finished, higher number of finished warps gives more priority. TBs in *barrierWait* state are prioritized based on the number of warps waiting at barrier, higher number of warps at barrier means more priority. Warps in each TB in *finishWait* and *barrierWait* states are prioritized based on their progress, lower progress means higher priority. TBs in *noWait* state are prioritized based on their progress,

Algorithm 1 Warp Scheduler

```
1: procedure insertFinishWarp(tb)
2: incrNumWarpsFinished(tb)
3: warpsFinished ← numWarpsFinished(tb)
4: if warpsFinished = 1 then
5:    fastTBPhase ← TBsWaitingInThrdBlkSched()
6:    if fastTBPhase then
7:       setTBInFinishWaitState(tb)
8:    end if
9:    sortWarps(tb, INC_ORDER)
10: end if
11: if warpsFinished = numWarpsPerTB then
12:    setTBFinished(tb) {/*TB finished*/}
13: end if
14: sortFinishWaitStateTBs()
15: end procedure
16:
17: procedure insertBarrierWarp(tb)
18: incrNumWarpsAtBarrier(tb)
19: warpsAtBarrier ← numWarpsAtBarrier(tb)
20: if warpsAtBarrier = 1 then
21:    setTBInBarrierWaitState(tb)
22:    sortWarps(tb, INC_ORDER)
23: end if
24: fastTBPhase ← TBsWaitingInThrdBlkSched()
25: if warpsAtBarrier = numWarpsPerTB then
26:    if fastTBPhase then
27:       setTBInNoWaitState(tb)
28:    else
29:       setTBInFinishNoWaitState(tb)
30:    end if
31: end if
32: sortBarrierWaitStateTBs()
33: end procedure
34:
35: procedure scheduleWarps
36: if fastToSlowTBPhaseTransition() then
37:    mergeFinishAndNoWaitTBs()
38:    clearFinishTBs()
39:    clearNoWaitTBs()
40: end if
41: finStateTBs ← TBsInFinState()
42: barrierStateTBs ← TBsInBarState()
43: noWaitStateTBs ← TBsInNoWaitState()
44: finNoWaitStateTBs ← TBsInFinNoWaitState()
45: orderedWarpList ← 0
46: if finStateTBs then
47:    insertWarps(finStateTBs, orderedWarpList)
48: end if
49: if barrierStateTBs then
50:    insertWarps(barrierStateTBs, orderedWarpList)
51: end if
52: if noWaitStateTBs then
53:    remTBs ← noWaitStateTBs
54: else
55:    remTBs ← finishNoWaitStateTBs
56: end if
57: if currCycle − lastSortCycle > THRESHOLD then
58:    lastSortCycle ← currCycle
59:    sortTBs(remTBs, INC_ORDER)
60:    sortWarps(remTBs)
61: end if
62: insertWarps(remTBs, orderedWarpList)
63: end procedure
```

higher progress means higher priority. Warps of each TB in *noWait* state are prioritized based on their progress, higher progress means higher priority.

D. Prioritization in *slowTBPhase*

Next, we will see how PRO prioritizes TBs and warps in the *slowTBPhase*. As shown in the state transition diagram in Figure 3, when the kernel execution phase changes to *slowTBPhase*, TBs in *finishWait* are merged with TBs in *noWait* state to form a new state called *finishNoWaitState*. TBs in *finishNoWaitState* are sorted in increasing order of their progress, so that the highest priority TB is the one with the least progress. The intuition behind sorting this way is to give more compute time to TBs that have made less progress, as there are no more TBs waiting in the Thread Block Scheduler in *slowTBPhase*, This implies TBs in *finishWait* state get the least priority when they move to *finishNoWait* state. In addition, warps in each TB are sorted in increasing order of their progress giving higher priority to warps with less progress. The effect of prioritizing thread blocks this way can be seen from the Figure 2(b), which shows that thread block 12 gets the least priority once the *slowTBPhase* starts and hence it takes lot more time to finish. It also shows that due to higher priority to thread blocks with lower progress, their runtimes decrease, e.g., thread block 17 takes the least amount of time in the last batch of thread blocks i.e. thread blocks 12-17.

In *slowTBPhase*, TBs in *barrierWait* state are handled the same way as in *fastTBPhase*. In order to increase the number of ready warps, TBs in *barrierWait* state are given higher priority than TBs in *finishNoWaitState*.

Function *TBsWaitingInThrdBlkSched* returns true if there are TBs waiting in the thread block scheduler to be assigned to an SM. The sorted list of warps is maintained in *orderedWarpList*. Functions *TBsInFinState*, *TBsInBarState*, *TBsInNoWaitState* return the number of TBs in *finishWait*, *barrierWait* and *noWait* states respectively.

E. Hardware Requirements

PRO requires maintaining the following information:

- Progress of each warp and TB,
- Number of warps of a TB waiting at barrier statements, *nWarpsAtBar* and
- Number of warps of a TB that have finished execution, *nWarpsFin*.

The progress of a warp is the sum of number of instructions executed by its constituent threads. We used one 4 byte register per warp to maintain its progress. Similarly, we used one 4 byte register per TB to maintain its progress, which is the sum of number of instructions executed by its constituent warps. Both these registers are incremented by the number of active threads[3] every cycle in which a warp is scheduled. This implies we need (4 * W + 4 * T) bytes of extra storage, where W is the maximum number of warps per SM and T is the maximum

[3]Due to thread divergence, the number of active threads in a warp can be less than the number of threads in a warp.

number of thread blocks per SM. In addition, two adders per warp scheduler are required to increment the progress of all warps and TBs on each SM.

The number of warps of a TB waiting at a barrier, *nWarpsAtBar*, is maintained using a 1 byte register (total T bytes) which is incremented every time a warp reaches a barrier instruction. This can be managed using one adder per warp scheduler[4]. The same hardware can be used to maintain the number of warps of a TB that have finished execution, *nWarpsFin*, as all warps of a TB will either reach a barrier statement or finish execution. When a TB goes from *barrierWait* state to *noWait* state, the 1 byte register needs to be reset to 0. Since the SM hardware already has a mechanism to know when a TB can cross a barrier statement, the same mechanism can be used to reset the 1 byte register. Some of this extra hardware to maintain *nWarpsAtBar* and *nWarpsFin* may not be needed as the hardware warp scheduler on an SM has to keep track of this information to know when a TB can cross a barrier statement or finishes execution. Nevertheless, we account for this in the overhead of our scheduler algorithm. We assume the current GPU hardware maintains a unique identification number for each warp and TB.

We assume that the existing warp scheduler maintains an array of all the warps allocated to it such that all warps of a TB are contiguous in the array. The warp sorting algorithm, *sortWarps*, sorts warps of a TB only and can be an in-place sorting algorithm. We assume one comparator per TB to sort its warps. The TB sorting algorithms viz., *sortFinishWaitStateTBs*, *sortBarrierWaitStateTBs* and *sortTBs* together can use one comparator to sort all TBs on an SM. The sorted order of TBs can be maintained in an array of 1 byte registers (total T bytes).

To schedule a warp, the warp scheduler has to go over the sorted order of TBs and select the highest priority ready warp from the TB. The starting position of warps of a TB in the array of warps can easily be computed using the index of the TB and the number of warps per TB, both of which can be assumed to be part of existing hardware warp scheduler.

So, PRO needs (4 * W + 4 * T) + T + T bytes of extra storage per SM, where W is the maximum number of warps and T is the maximum number of thread blocks on an SM. For NVIDIA Fermi architecture GPU, with W = 48 and T = 8, the extra storage per SM amounts to 240 bytes.

IV. Experimental Methodology

We used GPGPU-Sim [1] simulator version 3.2.2 to implement and evaluate PRO, our progress aware warp scheduling algorithm. GPGPU-Sim is a cycle level simulator. We used the configuration (Table I) for GTX480, an NVIDIA Fermi based GPU architecture. We compiled all applications using NVCC version 4.2 with default optimization level and used the PTX code for simulation.

[4]On Fermi GPU [6], each SM has two warp schedulers and warps of a TB are divided between the two warp schedulers, so in one cycle two warps of the same TB can potentially reach a barrier statement or finish executing.

Table II shows the kernels used for evaluation. We used applications from GPGPU-SIM[1], Rodinia [18] and CUDA-SDK[5] benchmarks suites. The first 10 kernels are from GPGPU-SIM benchmark suite, the next 6 are from Rodinia benchmark suite and the last 9 are from CUDA_SDK benchmark suite. The number of TBs in each kernel is specified in column 3. All the selected applications have at least 3% performance difference between the best and worst among TL, LRR and GTO scheduling algorithms. We used existing implementations of TL, LRR and GTO in GPGPU-SIM.

TABLE I: GPGPU-Sim Configuration

Architecture	NVIDIA Fermi GTX480
Number of SMs	14
Max No of Thread Blocks per SM	8
Max No of Threads per Core	1536
Shared Memory per Core	48KB
L1-Cache per Core	16KB
L2-Cache	768KB
Max No of Registers/Core	32768
No-of Schedulers	2
DRAM Scheduler	FR-FCFS

We used the number of simulation cycles reported by the simulator to compare performance. Figure 4 shows the performance gains of PRO over other scheduling methods. The geometric mean (shown as GEOMEAN) is 1.13x, 1.12x and 1.02x over TL, LRR and GTO respectively. Over TL, PRO achieves a maximum of 1.6x improvement in the application *scalarProd*. PRO achieves more than 1.2x improvement over TL in 7 out of 25 applications. Only 3 applications slowdown with a maximum slowdown of 4% in *mergeHistogram256Kernel*.

The maximum improvement over LRR is also seen in application *scalarProd*. Six applications gain more than 20% improvement over LRR. Only 4 applications perform worse

TABLE II: Benchmark Applications

Application	Kernel	Thread Blocks
AES	aesEncrypt128	257
BFS	kernel	256
CP	cenergy	256
LPS	GPU_laplace3d	100
NN	executeFirstLayer	168
NN	executeSecondLayer	1400
NN	executeThirdLayer	2800
NN	executeFourthLayer	280
RAY	render	512
STO	sha1_overlap	384
backprop	bpnn_layerforward	4096
backprop	bpnn_adjust_weights_cuda	4096
b+tree	findRageK	6000
b+tree	findK	10000
hotspot	calculate_temp	1849
pathfinder	dynproc_kernel	463
convolutionSeparable	convolutionRowsKernel	18432
convolutionSeparable	convolutionColumnsKernel	9216
histogram	histogram64Kernel	4370
histogram	mergeHistogram64Kernel	64
histogram	histogram256Kernel	240
histogram	mergeHistogram256Kernel	256
MonteCarlo	inverseCNDKernel	128
MonteCarlo	MonteCarloOneBlockPerOption	256
scalarProd	scalarProdGPU	128

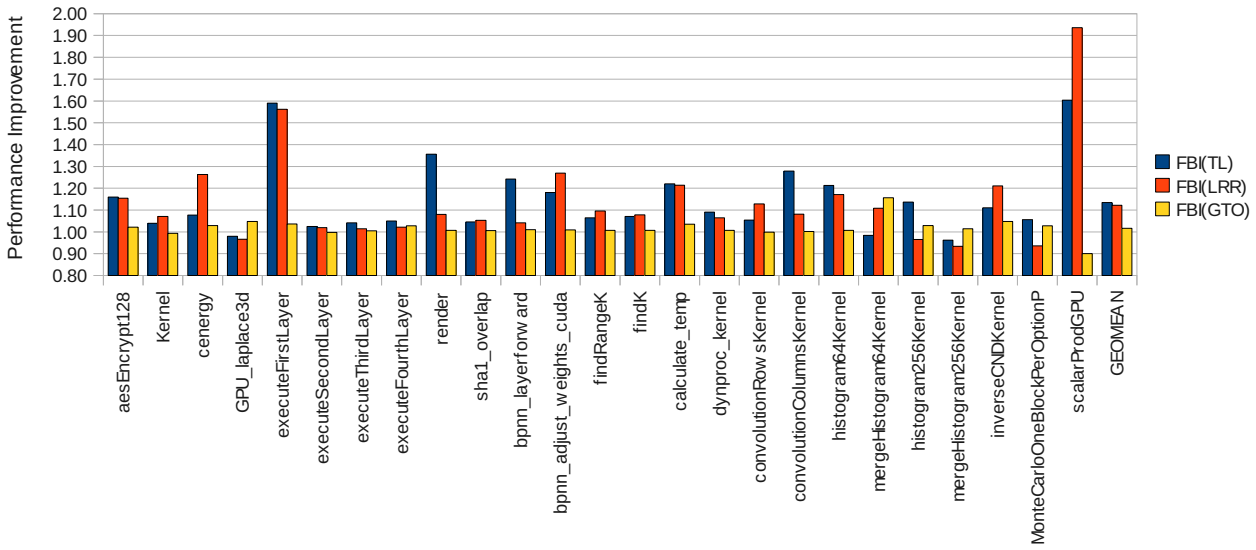


Fig. 4: Performance of Progress Aware Warp Scheduler

than LRR and the maximum slowdown is only 7%. One of the reasons for this slowdown is the increase in L1 and L2 cache miss rates.

GTO algorithm prioritizes older warps and hence causes unequal progress among warps which helps hide long latencies. But it does not handle warp divergence. Since PRO handles both, we see improvements over GTO, although small. In kernel *mergeHistogram64Kernel*, PRO achieves 16% performance improvement over GTO. PRO performs worse than GTO in only 2 out of 25 applications. Out of the two slowdowns, one is slower by only 1%. Only in application *scalarProdGPU*, our algorithm is slower by 10%. We investigated it and found that the application is very sensitive and its performance improved as much as 11% when we disabled special handling of warp divergence at barriers. As a future work, we would like to dynamically enable or disable special handling of barrier statements, long latency statements, etc., by profiling each application.

We also report the reduction in stalls with our algorithm in Figure 5. GPGPU-SIM reports 3 types of stalls viz., Idle, Scoreboard and Pipeline stalls. Stall cycles reported by GPGPU-SIM are at the GPU level and not per SM level. The numbers reported are not per kernel but per application. PRO achieves an improvement (geometric mean) of 1.32x over TL, 1.19x over LRR and 1.04x over GTO.

Table III shows comparison of PRO with the three baseline algorithms. For each application, it shows ratios of each type of stall as well as the total stalls. For example, Column 'Improvement Over TL/Idle' shows ratio of Idle stalls in TL to Idle stalls in PRO. The numbers reported are for each application and not for each kernel.

Compared to the TL algorithm, PRO has 2.4x fewer Idle stalls and 1.58x fewer Scoreboard stalls. Even if TL has fewer (0.7x) pipeline stalls, overall PRO has 1.32x fewer stalls. As against LRR, PRO has 1.24x fewer Pipeline stalls and 3.21x

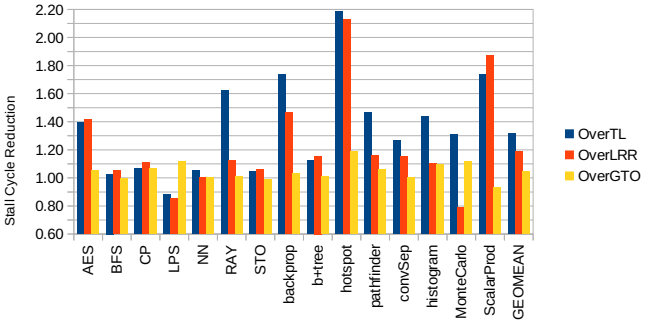


Fig. 5: Improvement in stalls with Progress Aware Warp Scheduler, Y axis is the ratio of stalls cycles, where the denominator is the stalls in Progress Aware Warp Scheduler.

fewer Idle stalls. In spite of an increase in Scoreboard stalls, overall PRO has 1.19x fewer stalls.

PRO is able to reduce 2 out of the 3 types of stalls without much increase in the third type. Hence, PRO is able to achieve considerably better performance compared to TL and LRR.

Compared to the GTO algorithm, PRO reduces both Idle and Scoreboard stalls by 10%, with no change in Pipeline stalls and achieves a reduction of 4% in total stall cycles.

The reduction in stall cycles and improvement in performance shows that PRO is able to change priorities of TBs and warps dynamically in the two phases of execution and different execution states of TBs.

To get an idea of how priorities of TBs change during the course of their execution, we show in Table IV the sorted order of TBs in application AES for the first batch of 6 TBs (TB indices 0 to 5) that executed on SM 0. They all finished between cycle 16000 and 17000. Each row shows the TBs in decreasing order of priority going from left to right. PRO sorts

TABLE III: Improvement in stall cycles with PRO. Number greater than 1 means PRO has fewer stalls.

Application	PRO Stall Cycles			Improvement Over TL				Improvement Over LRR				Improvement Over GTO			
	Pipe	Idle	SB	Pipe	Idle	SB	**Total**	Pipe	Idle	SB	**Total**	Pipe	Idle	SB	**Total**
AES	276470	83319	77275	0.90	1.34	3.22	1.39	1.16	3.04	0.58	1.41	1.04	0.94	1.21	1.05
BFS	16256179	944089	3170756	0.98	3.25	0.62	1.03	1.00	3.04	0.74	1.05	0.99	1.05	1.03	1.00
CP	1870086	102575	782519	0.07	0.51	3.51	1.06	1.08	7.74	0.30	1.11	1.02	2.15	1.05	1.07
LPS	350570	44118	891228	0.88	1.74	0.84	0.88	1.01	1.06	0.78	0.85	0.97	1.57	1.16	1.12
NN	835844	375793	43891259	1.10	2.42	1.04	1.05	1.27	1.10	1.00	1.00	1.05	1.54	1.00	1.00
RAY	228420	753553	1004409	0.56	1.69	1.81	1.62	1.10	1.34	0.97	1.12	1.00	1.01	1.02	1.01
STO	447805	4403628	1547667	0.97	1.23	0.55	1.05	0.98	1.25	0.55	1.06	0.96	1.03	0.87	0.99
backprop	2002173	36809	741869	0.81	8.15	3.93	1.74	1.50	7.85	1.05	1.46	1.01	1.03	1.10	1.03
b+tree	6986311	116761	6326897	0.89	9.51	1.22	1.12	1.06	2.80	1.24	1.16	1.01	1.17	1.02	1.01
hotspot	741013	31960	398403	1.34	2.84	3.71	2.18	2.15	12.37	1.27	2.13	1.11	1.45	1.33	1.19
pathfinder	803621	88212	1949788	0.71	3.25	1.70	1.46	1.36	1.53	1.06	1.16	1.07	1.07	1.06	1.06
convSep	127652468	7940692	72377480	1.19	5.19	0.98	1.27	1.01	8.71	0.58	1.15	1.01	0.90	0.99	1.00
histogram	34801184	12015072	23159622	0.77	0.64	2.85	1.44	0.97	2.16	0.75	1.10	1.02	0.69	1.40	1.09
MonteCarlo	2225476	263926	7036821	0.15	1.30	1.68	1.31	1.64	2.54	0.45	0.79	0.98	0.91	1.16	1.11
ScalarProd	83801	33889	260654	1.66	8.44	1.11	1.74	1.86	14.24	0.28	1.87	0.83	0.73	1.30	0.93
GEOMEAN				0.70	2.40	1.58	1.32	1.24	3.21	0.70	1.19	1.00	1.10	1.10	1.04

TABLE IV: Sorted order of TBs in AES

Cycle	1	2	3	4	5	6
1000	0	1	2	3	4	5
2000	0	4	3	1	2	5
3000	0	4	3	1	2	5
4000	0	4	3	1	2	5
5000	0	4	3	1	2	5
6000	0	1	2	3	4	5
7000	0	1	2	3	4	5
8000	0	1	2	4	3	5
9000	0	1	4	5	3	2
10000	0	1	4	5	2	3
11000	0	1	4	5	2	3
12000	0	1	4	2	3	5
13000	0	1	4	2	3	5
14000	0	1	4	2	3	5
15000	0	1	4	2	3	5
16000	0	1	2	4	3	5

TBs every 1000 cycles in the decreasing order of their overall progress (not just in the preceding 1000 cycles). The first row shows that TB 0 is first in the sorted order i.e. has made the most progress and TB 5 is the last in the sorted order i.e. has made the least progress. So, for the next 1000 cycles, TB 0 has the highest priority and TB 5 has the lowest priority But in those 1000 cycles, TB 4 makes more progress than TBs 1, 2, and 3 and goes to position 2 in the sorted order. Again, at cycle 6000 we see that the sorted order changes with 4 TBs changing places. The sorted order changes 7 times as shown in the table.

V. Related Work

There has been a lot of work on warp scheduling. Narasiman et al. [22] proposed a two level warp scheduler to hide long latencies by allowing warps to make unequal progress and hence reach long latency instructions at different times. Our proposed solution uses progress made by each thread block and warp to prioritize warps and not only effectively hides long latencies but also handles problems due to warp divergence and resource allocation at thread block granularity. Gebhart et al. [11] focussed on energy efficiency in their two level warp scheduler. Their proposed algorithm uses two different sets of warps, one to hide short latencies such as ALU and local memory access latencies and another for long latencies such as global memory accesses. OWL [3] focusses on improving performance using various CTA aware techniques to reduce cache contention and improve DRAM performance in bandwidth-limited GPGPU applications. Their proposed scheduler forms groups of CTAs and executes warps in each group in round-robin fashion. The groups are also selected in a round-robin fashion. Jog et al. [2] propose techniques to reduce long memory latencies using coordinated thread scheduling and prefetching. NMNL [14] proposes to dynamically adjust the number of thread blocks based on application characteristics. The focus is again on reducing contentions in caches and memory. Rogers et al. [21] propose a technique to dynamically vary thread level parallelism by controlling the number of wavefronts considered for scheduling, to reduce cache contention. DAWS [20] proposes a mechanism to improve intra-warp locality in loops by estimating the L1 data cache capacity needed. Lakshminarayana et al. [13] discuss the effects of instruction fetch and memory scheduling on GPU performance.

A patented register management scheme [7] uses the concept of more virtual registers than the actual physical registers. Hence it can launch more thread blocks than allowed by the physical registers. The extra thread blocks can be used to hide long latencies. Yang et al. [25] discuss the problem of underutilization of shared memory and reduced thread block residency caused by allocation and deallocation of shared memory at the thread block granularity. They propose hardware and software solutions to effectively use the shared memory and improve performance.

Warp-Level Divergence in GPUs [16], shows that in many GPGPU applications, warps of a TB have varying runtimes and hence finish at different times. They have proposed a hardware mechanism to allocate threads to an SM at a warp level granularity when there are not enough registers to allocate a thread block, to mitigate the losses due to spatial as well as temporal underutilization of registers. CAWS [19] proposes techniques to prioritize critical warp(s) to reduce the execution

time disparity among warps within the same thread block. Our solution to handle warp-level divergence is to identify slower warps based on their progress and prioritize them.

Various hardware and software techniques to handle thread divergence have been proposed in [23] [24] [9] [12] [10] [17] [8].

VI. Conclusion

We proposed a warp scheduling algorithm to reduce the stalls due to long latency instructions as well as warp-level divergence. Our algorithm achieves unequal progress among TBs and warps in each TB to reduce the chances of many warps reaching the same long latency instruction close to each other in time. Warp-level divergence is reduced by prioritizing TBs that have one or more warps waiting at barrier statements or finished execution. The algorithm achieved an improvement in performance of 1.13x over TL and 1.12x over LRR. It also achieved a an improvement in stall cycles of 1.32x over TL and 1.19x over LRR.

In future, we plan to look at improving cache and memory performance of high priority warps to further reduce stall cycles. We intend to look at more parameters to define the progress of a warp/TB.

VII. Acknowledgements

We thank the anonymous reviewers for their suggestions and comments. We also thank members of the Lab for HPC for discussions and feedback on improving the paper. The first author acknowledges the funding received from Google India Private Limited.

References

[1] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. *Analyzing cuda workloads using a detailed gpu simulator*. ISPASS-2009.

[2] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. *Orchestrated scheduling and prefetching for gpgpus*. ISCA-2013,

[3] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, C. R. Das. *OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance*. ASPLOS-2013.

[4] *CUDA C Programming Guide*.

[5] CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/gpu-computing-sdk

[6] http://www.nvidia.in/content/PDF/fermi_white_papers /NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[7] D. Tarjan and K. Skadron. *On demand register allocation and deallocation for a multithreaded processor*, June 30 2011. US Patent App. 12/649,238.

[8] G. Diamos, B. Ashbaugh S. Maiyuran A. Kerr H. Wu and S. Yalamanchili. *SIMD Re-Convergence At Thread Frontiers*. MICRO-2011

[9] J. Meng, D. Tarjan and K. Skadron. *Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance*. ISCA-2010

[10] J. Anantpur and R. Govindarajan. *Taming Control Divergence in GPUs through Control Flow Linearization*. CC-2014

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, K. Skadron *Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors*. ISCA-2011

[12] M. Rhu and M. Erez. *The dual-path execution model for efficient gpu control flow*. HPCA-2013.

[13] N. B. Lakshminarayana and H. Kim. *Effect of Instruction Fetch and Memory Scheduling on GPU Performance*. Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.

[14] O. Kayiran, A. Jog, M. T. Kandemir and C. R. Das. *Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs*. PACT-2013.

[15] OpenCL. *www.khronos.org/opencl*.

[16] P. Xiang, Y. Yang, H. Zhou. *Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation*. HPCA-2014.

[17] Q. Xu and M. Annavaram. *PATS: Pattern Aware Scheduling and Power Gating for GPGPUs*. PACT-2014

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. *Rodinia: A benchmark suite for heterogeneous computing*. IISWC-2009.

[19] S. Lee and C. Wu *CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads*. PACT-2014

[20] T. G. Rogers, M. OConnor, and T. M. Aamodt. *Divergence-Aware Warp Scheduling*, MICRO-2013

[21] T. G. Rogers, M. OConnor, and T. M. Aamodt. *Cache-conscious wavefront scheduling*. MICRO-2012.

[22] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, Y. N. Patt. *Improving GPU Performance via Large Warps and Two-Level Warp Scheduling*. MICRO-2011.

[23] W. Fung and T. Aamodt. *Thread Block Compaction for Efficient SIMT Control Flow*. HPCA-2011

[24] W. W. L. Fung, I. Sham, G. Yuan and T. M. Aamodt *Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow*. MICRO-2007

[25] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. *Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput*. PACT-2012.