Exploring the Limits of GPGPU Scheduling In Control Flow Bound Applications

Roman Malits, Avi Mendelson, Avinoam Kolodny, Evgeny Bolotin
{romanma@tx, kolodny@ee}.technion.ac.il , EE Department, Technion-IIT, Israel
evgeny.bolotin@intel.com , Intel Corporation, Haifa, Israel
avim@microsoft.com , Microsoft R&D Israel

ABSTRACT
GPGPUs are optimized for graphics, for that reason the hardware is optimized for massively data parallel applications characterized by predictable memory access patterns and little control flow. For such applications' e.g., matrix multiplication, GPGPU based system can achieve very high performance. However, many general purpose data parallel applications are characterized as having intensive control flow and unpredictable memory access patterns.
Optimizing the code in such problems for current hardware is often ineffective and even impractical since it exhibits low hardware utilization leading to relatively low performance.
This work tracks the root causes of execution inefficacies when running control flow intensive CUDA applications on NVIDIA GPGPU hardware.
We show both analytically and by simulations of various benchmarks that local thread scheduling has inherent limitations when dealing with applications that have high rate of branch divergence. To overcome those limitations we propose to use hierarchical warp scheduling and global warps reconstruction. We implement an ideal hierarchical warp scheduling mechanism we term ODGS (Oracle Dynamic Global Scheduling) designed to maximize machine utilization via global warp reconstruction. We show that in control flow bound applications that make no use of shared memory (1) there is still a substantial potential for performance improvement (2) we demonstrate, based on various synthetic and real benchmarks the feasible performance improvement.
For example, MUM and BFS are parallel graph algorithms suffering from significant branch divergence. We show that in those algorithms it's possible to achieve performance gain of up to x4.4 and x2.6 relative to previously applied scheduling methods.
Keywords: multicore processing; performance analysis; parallel machines; scheduling algorithm

1. INTRODUCTION

GPGPU calls to extend the use of GPUs for general purpose applications. Accordingly, the graphics hardware was extended and new programming models such as CUDA[3] and OpenCL[19] were introduced and enabled the use of GPUs for wide verity of applications [17].
Current GPGPU programming languages, such as CUDA, expose GPU micro-architectural internals. This allows programmers to perform fine grained code level optimizations by explicitly specifying and hand-tuning available application thread level parallelism to the specific hardware.
Such optimizations allows to achieve high performance in a class of applications that can be expressed as massive parallel with very simple control structure and dependencies between the threads, since such applications naturally fit the current GPGPU design philosophy. However, for many other applications; e.g., those that have intensive control flow and unpredictable memory access patterns, optimizing the code for current GPGPU hardware is often impractical resulting in severe drop in the system utilization and lowered performance [1][2][4][6][7].
In this paper we explore the limitations and pin down the root causes of execution inefficacies in several classes of CUDA applications. GPGPU architectures assume that the software is highly parallel and can take advantage of two levels of parallelism; (1) TLP (task level parallelism), that assume that each phase of the execution can be divided into independent parallel parts and (2) inner thread parallelism that take advantage of the data parallelism within each task. While the TLP is mainly used to mitigate stalls caused by long latencies, usually related to memory access time, the inner thread parallelism is supported via the warp scheduling and SIMD-like hardware. In order to achieve best performance, one should carefully balance these two levels of parallelism. This task can be quite challenging in particular when threads within the task use complex control flow as in [11][18][21][22][23]. As indicated in [1][2][7], the scheduling mechanisms in GPU hardware such as NVIDIA G200 or even in Fermi incurs several inefficiencies associated with branch divergence and scheduling in the resolution of blocks.

Current hardware is built out of several streaming multiprocessors (SMs) and employs scheduling optimizations local to each SM. This approach is effective when control flow is simple but, as we will show later on, when dealing with applications that have high rate of branch divergence the use of local optimizations have inherent limitations.
In this work we analyze those scheduling related bottlenecks analytically and through simulations by using a modified version of the GPGPU-SIM [2] cycle accurate simulator. To optimize the performance of control flow bound applications we propose a novel approach based on hierarchical warp scheduling and global warp reconstruction. We implemented (in GPGPU-SIM) an ideal hierarchical scheduling mechanism we termed ODGS (Oracle Dynamic Global Scheduling) designed to maximize the amount of effective parallelism in the system through the use of global warp reconstruction.
We use ODGS to study the potential for performance improvement if global warp reconstruction is used. We show by simulations of synthetic and real benchmarks that there is still a substantial potential for performance improvement via a more sophisticated global scheduling approaches especially when targeting control flow bound applications that make no use of shared memory.
For example, MUM and BFS are parallel graph algorithms suffering from significant branch divergence. We show that in those algorithms it's possible to achieve performance gain of up to x4.4 and x2.6 relative to previously applied scheduling methods in ideal conditions.
We perform a sensitivity analysis and show that implementing such a hierarchical global warp scheduling in real hardware is feasible. Our findings indicate that if future graphics will be built to allow global warps reconstruction and thread migration between SMs it may ease the optimization problem and allow to improve the performance of applications that otherwise would not be able to efficiently run in GPGPU environment, or at least will require spending long time at the software optimization phase.
Those findings manifest for the need for future research in the field of global thread scheduling and migration in GPGPU hardware.
The rest of the paper is organized as follows: section 2 presents an analytical study on the various phenomena that hamper performance of control flow intensive application programs running on GPGPU. Section 3 presents experimental study involving different scheduling methods and CUDA benchmarks. Section 4 discusses previous work and section 5 presents conclusions and future work.

2. CONTROL FLOW INFLUENCE ON PERFORMANCE

Applications with abundant control flow exhibit some special bad behaviors which result in lowered utilization of the GPGPU hardware. This section presents a detailed study of those phenomena which hamper the application performance.

2.1 Baseline Compute Model

CUDA extends C programming language by allowing the programmer to define kernels (also called shader programs). In the CUDA programming model the GPGPU is treated as a co-processor to the CPU. CUDA program execution is started on the CPU (host). At some point a data parallel portion of the application is sent for execution to the GPGPU (device) as multi-threaded kernel. This can happen multiple times until the program completes.
The design goal of modern GPU architectures such as NVIDIA G200 and Fermi is to achieve high performance in massively data parallel applications while using relatively inexpensive, in term of complexity and power, hardware. To achieve this goal, GPGPUs use multiple streaming multiprocessors (SMs) for parallel execution with each SM being responsible for a portion of the work. SMs use SIMD instruction scheduling which allows efficiently handling applications with extensive data level parallelism by amortizing data-independent control hardware across multiple scalar pipelines.
The kernel is comprised of a large group of threads called grid, with each thread having unique ID. When launching the kernel, the grid is divided by the programmer into equally shaped thread groups called blocks. The system can assume that threads belonging to different blocks are always independent of each other. Threads that belong to the same block can synchronize their execution and collaborate through fast shared memory.

Current hardware schedules work to SMs at the granularity of blocks. If resources permit, multiple blocks can be assigned to a single SM, thus sharing a common pipeline for their execution. Per-block resources are not freed until all the threads within the block have completed execution.
Threads within a block are scheduled to the pipeline in a fixed group of 32 threads called warp. All threads in a given warp are processed in "lock-step" in parallel with a single instruction on different data values. Fine-grained multithreading is used to mask stalls of warps waiting on long-latency events. When a thread is blocked by a memory request, the corresponding warp is removed from the pool of "ready" warps allowing active warps to proceed.
In this work we use GPGPU-SIM v2.1.1b with the same GPGPU configuration used in [2].

2.2 Branch Divergence

Branch divergence occurs when threads inside a warp scheduled to the SIMD pipeline take different paths due to conditional branch. Due to SIMD restrictions only one path of the branch can be executed at a time per decoder, for that reason branch divergence creates a control flow hazard.
To deal with this situation the hardware serializes the execution of warps with diverged threads. Serialization is implemented by means of predication to mask off threads taking the branch in the alternate direction. This approach allows to support branch divergence on SIMD hardware at the cost of lowered utilization. In Fermi, each SM has two decoders which allow to improve performance on the expense of a severe increase in SM hardware complexity, cost and needed power.
In this work we show that local SM scheduling optimizations have inherent limitations which prevent them to fully recover the performance loss caused by branch divergence.
Figure 1 presents a simple branch divergence example: warp diverges with half of the threads taking path A and the others follow path B.

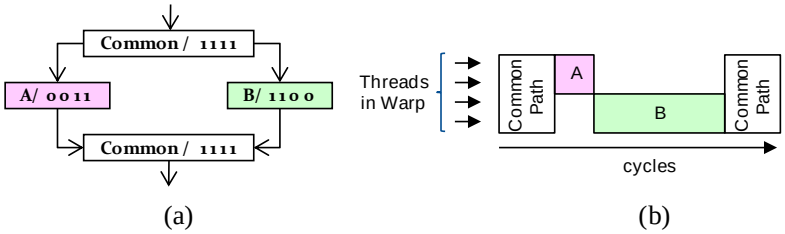


Figure 1: (a) branch divergence (b) utilization loss caused by warp serialization upon conditional branch

It is easy to see that during the execution of parts A and B of the code, utilization is only 50%. When execution path is long enough, warp may have multiple divergence points and so the execution time may be exponential with the number of diverged branches.

2.3 Execution Path Variance

Branch divergence might cause significant execution path variance between warps. Some warps might have short execution path and others very long. Such an imbalance can be another source for major inefficiencies and significant under-utilization of the pipeline even if the number of warps that have branched to long execution paths is low. This situation is illustrated in Figure 2.

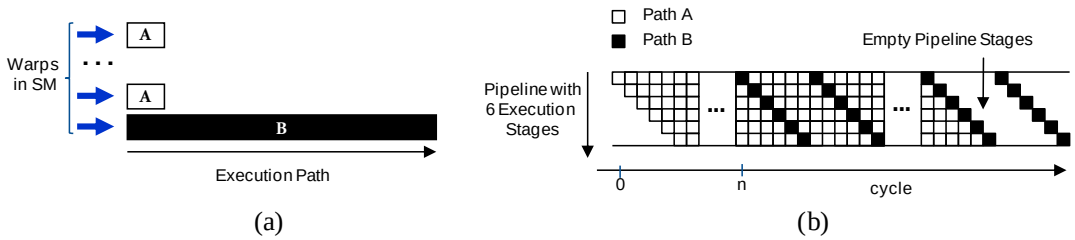


Figure 2: (a) Execution path length variance caused by branch divergence (b) Impact on the runtime pipeline utilization

In this example, a single block is scheduled to the SM. In this block one warp has long execution path B and all the remaining warps have short execution path A. If warp with path B is still running while all the warps with short path A have finished, warp executing path B will remain the only warp running in the SM which incurs a major utilization penalty.

2.4 Ability to Hide Memory Access Latency

GPGPUs are an example of a many-core design that adopted the use of fine-grained multi-threading to better tolerate memory and pipeline latencies.
Each SM has a pool of warps. During the kernel execution many different warps can co-exist in the pipeline of each SM at the same time. When threads within a warp issue a memory request, this warp is stalled inside the memory unit, meanwhile others warps are allowed to continue their execution, keeping the pipeline utilized. When threads inside the memory unit obtain their data, they write it to the register file and resume execution in the pipeline.
This organization allows more efficient use of the pipeline since long memory latency can be hidden, given enough "live" warps are available inside the SM.
NVIDIA made an important linkage between the internal structure of the application and the internal structure of the graphics hardware. The application threads are divided into blocks and the system can assume that threads belonging to different blocks do not share state and so can run independently of each other on different SMs, which are also independent of each other.
During the runtime, multiple blocks can be scheduled to the same SM, this number depends on the resources needed for each block and resources available to the SMs. After block is scheduled, its resources are freed only when all the threads within the block have finished. This limitation, coupled with execution path variance can severely undermine the SM ability to hide long memory accesses. If a significant number of warps in the SM has finished and few warps with long execution paths continue to run and prevent resources reuse, the SM will have no way to hide long memory access latency.

2.5 Choosing Optimal Block Size

In current architecture, threads in a grid are segmented and scheduled to SM's in the resolution of blocks. Only when all the threads within the block have finished, resources assigned to this block can be reused and another block can be scheduled to the SM. This type of scheduling is not optimal for a number of reasons.
CUDA programming model allows the programmer to optimize kernel execution by choosing a handful number of parameters such as the size of the block. Choosing optimal parameter can be quite challenging since in many cases the programmer need to take into consideration many conflicting factors. When choosing the block size we would like to increase the number of blocks assigned to a SM in order to increase the parallelism within the SM, but on the other hand, all resources belonging to the SM are shared among the different blocks so that increasing the number of blocks per SM reduces the amount of resources each block can get.
For optimal performance the choice must take into consideration factors such as the number of registers needed for each thread, thread execution path length, number of control points, number of blocks that can be scheduled to SM simultaneously, number of blocks relative to SM's number, available shared memory, synchronization and other technical features of the hardware.
Obviously, choosing the right block size is crucial for good performance.
Experiments showed that even for relatively simple programs that can be developed in a few hours, the optimization process might take weeks. As GPGPU evolves, choosing the optimal block size is becoming increasingly complex, each generation of hardware introduces changes and optimizations made for one generation of GPU may not fit to the new one.

As mentioned before, when branching is involved various length execution paths might occur. Resources of threads that finish early could theoretically be used however they are locked since the block hasn't finished yet. This in turn reduces the ability to hide long memory accesses since less ready warps are available in the SM.

For most applications, even if the initial load assignment of thread groups to SM was perfectly done at the allocation time, at final execution stages of the grid, some SMs stay inactive and underutilized. In different situations such as when using small grids this can lead to significant performance loss. Although this issue can be partially addressed in Fermi by executing multiple kernels in parallel, not every application can actually use it.

3. EVALUATION AND DISCUSSION

Here we present simulation results showing the ability of various scheduling methods to mitigate the performance loss caused by branch divergence.
The study was performed using our modified version of the GPGPU-SIM [2], a publicly available open source cycle-accurate performance simulator of an NVIDIA GPU.
First we describe the CUDA benchmarks we used in this study and lay out the various scheduling methods we checked. As a baseline, we perform a very detailed runtime analysis of MUM benchmark and discuss the limitations of local scheduling methods when dealing with control flow bound applications.
Then we discuss possible ways to mitigate performance loss caused by branch divergence by using our new proposed global hierarchical warp scheduling mechanism termed ODGS (Oracle Dynamic Global Scheduling). First we check the performance improvement potential by using ODGS on synthetic benchmark and then compare ODGS to other scheduling mechanisms in order to estimate the potential performance benefits of such scheduling approach.

3.1 Common CUDA Benchmarks

This section uses benchmarks listed in Table 1. Those benchmarks are a subset of the benchmarks used in [2]. We selected those programs from the suit that have abundant control-flow, in addition none of the benchmarks are using shared memory.

BFS[11] performs breadth-first search on a graph. Each node in the graph is mapped to a different thread. We perform breadth-first search on a random graph with 65,536 nodes and an average of 6 edges per node.

MUMmerGPU[18] uses a suffix tree to efficiently find alignments of short DNA sequences against a reference genome. The tree is traversed from the root in a data dependent manner, with each edge holding a variable number of base pairs which must all match for the traversal to proceed to the next node. Computation is parallelized by mapping each input string to a thread.

Ray Tracing (RAY)[22] is using *u*p to 5 levels of reflections and shadows, each pixel rendered corresponds to a scalar thread with each thread behavior depending on what object the ray hits. We simulate rendering of a 256x256 image.

NN[21] uses a convolution neural network to recognize handwritten digits. Pre-determined neuron weights are loaded into global memory along with the input digits. We simulate recognition of 28 digits from the Modified National Institute of Standards Technology database of handwritten digits. We used the updated version of this benchmark described in [4].

Weather Prediction (WP)[23] uses the GPGPU to accelerate a portion of the Weather Research and Forecast model (WRF), which can model and predict condensation, fallout of various precipitation and related thermodynamic effects of latent heat release. We simulate the kernel using the default test sample for 10 time steps.

Table 1.Benchmarks used in the study

Benchmark	Grid Size	Block Size	Blocks/SM	Total Threads	Branches Number	Barriers
BFS	(128,1,1)	(512,1,1)	2	65536	4	No
MUM	(782,1,1)	(64,1,1)	3	50000	15	No
RAY	(16,32,1)	(16,8,1)	3	65536	20	Yes
NN	(6,28,1)	(13,13,1)	5	28392	4	No
WP	(9,8,1)	(8,8,1)	3	4608	61	No

3.2 Compared Scheduling Methods

GPGPU-SIM includes an implementation of three different local scheduling methods. Those methods represent various levels of optimization with respect to their ability to deal with branch divergence: basic (POD), sophisticated (DWF) and optimal (MIMD).

Immediate Post Dominator (POD)

When threads diverge, there is usually some point in the future where their execution path is the same again. Immediate post dominator is the first PC of that path. POD is a mechanism for managing branch divergence within a warp. This mechanism supports branch divergence by masking off threads within a warp that take alternative direction and serializing the warp execution. Immediate post dominator is used as the re-convergence point. This mechanism is closely resembling the way divergence is handled by G200 hardware. The method is modeled in GPGPU-SIM using a stack based mechanism described in [1].

MIMD (Multiple Instruction Multiple Data)

In the context of this work, MIMD refers to architecture similar to our baseline SIMD configuration, except that the scalar pipelines are allowed to execute different instructions at the same time. This simulates a situation where every scalar pipeline has its own decoder. Under the perfect memory model this scheduling algorithm achieves the best possible performance with regard to the possible local scheduler optimizations.

Dynamic Warp Formation (DWF)

This scheduling algorithm is presented in [1]. The algorithm proposes to minimize the performance loss caused by branch divergence by creating a pool of diverged threads within the SM. The local thread scheduler will then try to form new warps from the local pool of threads and send them for execution. We used the non lane-aware version of the algorithm with the majority scheduling policy for the warp selection.

3.3 Detailed analysis of MUM benchmark

We take the MUM benchmark as an example for control flow bound application and present a deep analysis of its runtime behavior. Being a parallel graph algorithm, this benchmark represents a worst case scenario type of application for the GPGPU exhibiting both high rate of branch divergence and highly irregular memory access pattern. In this work we use GPGPU-SIM with the same GPGPU configuration used in [2]. Table 2 summarizes the basic GPGPU parameters.

Table 2: GPGPU-SIM parameters used in the study

Number of shader cores	28
Warp size	32
SIMD pipeline width	8
Max Number of threads/core	1024
Max Number of blocks/core	8
Number of registers / core	16384
Perfect Memory Model	Yes

Figure 3 compares the IPC (instruction/cycle) using POD scheduling with perfect memory (all requests are cache hit) and normal memory.

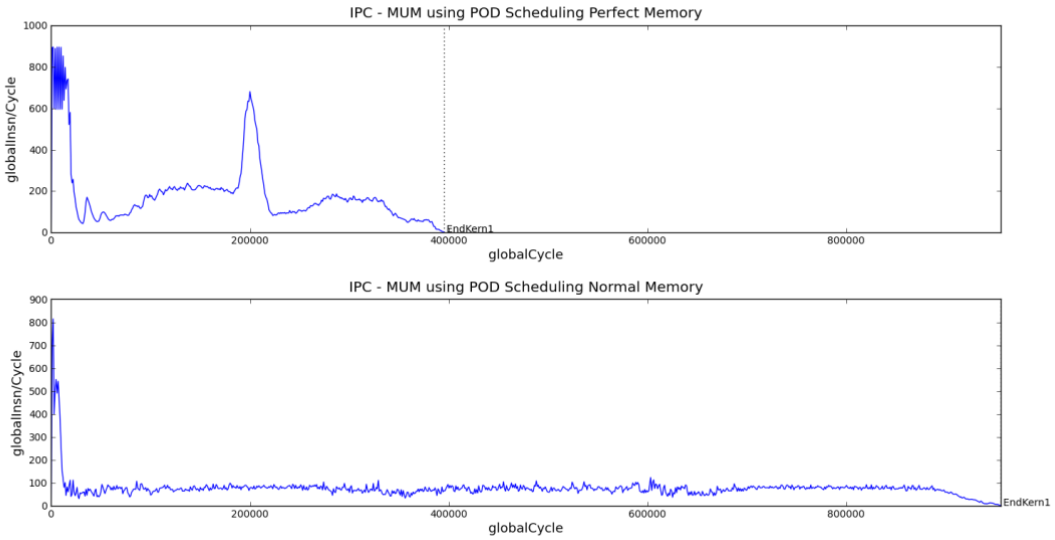


Figure 3: Comparison of IPC/SM for MUM benchmark using POD scheduling for normal and perfect memory models

The application runtime increases ~x2.25 when using normal memory compared to perfect memory. This result indicate that improving scheduling efficiency have the potential to significantly improve performance.

Figure 4 shows the memory utilization per DRAM channel while using POD scheduling method.

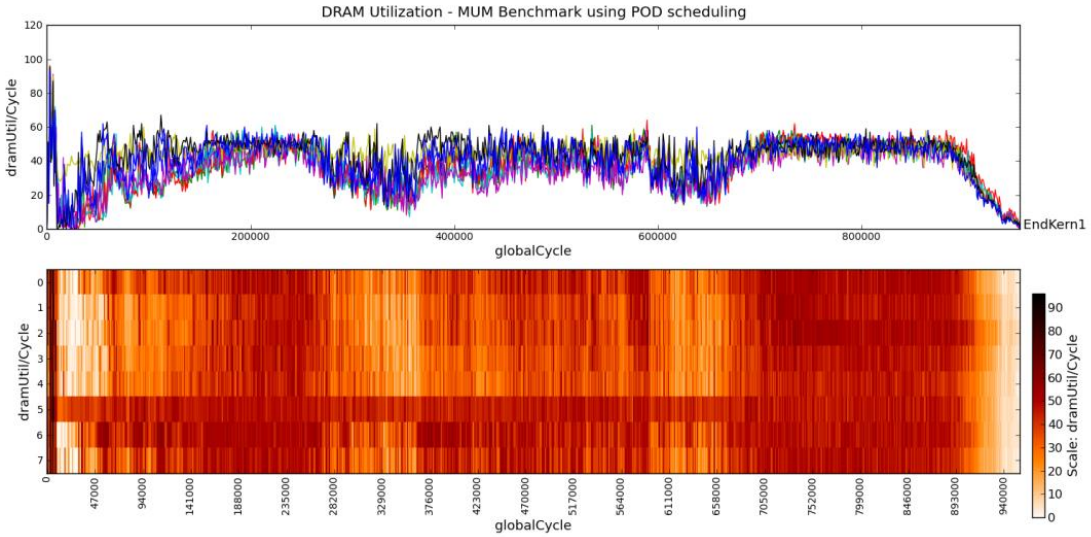


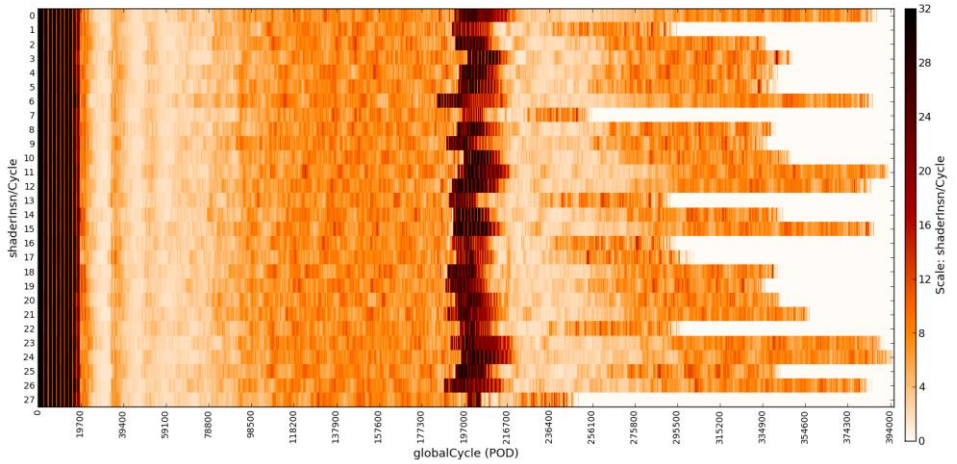
Figure 4: Memory utilization per DRAM channel in MUM benchmark using POD scheduling

Both parts of the figure represent the same information in different ways. The upper part plots the DRAM utilization of each channel on the same axes while the lower part shows utilization of each channel separately in different rows. It's easy to see that the utilization is 30-60% most of the time and varies across different DRAM channels. This proves that in this application there is still a significant memory bandwidth available to be used by various scheduling optimizations.

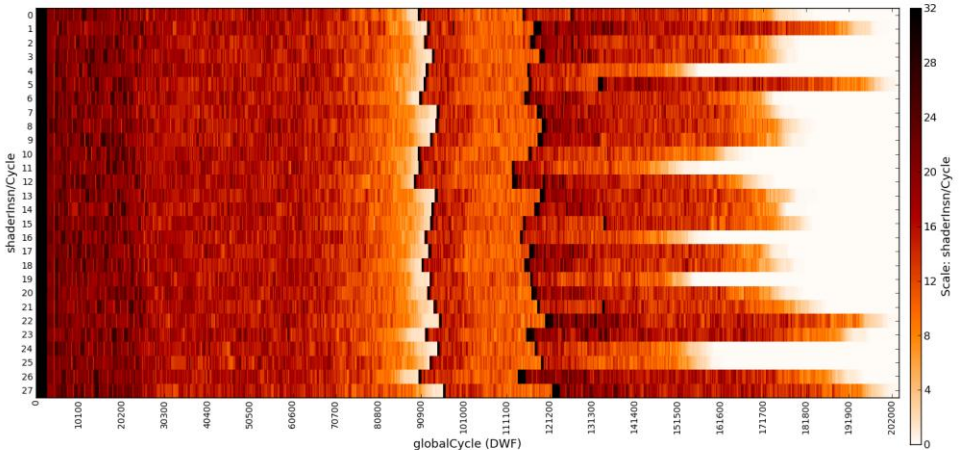
Figures 3 and 4 show that both scheduling and memory model have significant effect on the performance of MUM. Since the focus of this work is to explore the ability of various scheduling methods to improve the performance of GPGPU when dealing with control flow bound applications, we decided to isolate the effect of the memory model and run all further simulations

using perfect memory mode (all requests are cache hits) since this allows to see more clearly the various scheduling related phenomena that hamper the performance of such applications.
Figure 5 compares the IPC/SM during the execution of the MUM benchmark for POD, DWF and MIMD scheduling methods. Each row in the figure corresponds to IPC of one of the 28 SMs during the MUM benchmark execution.

(a)



(b)



(c)

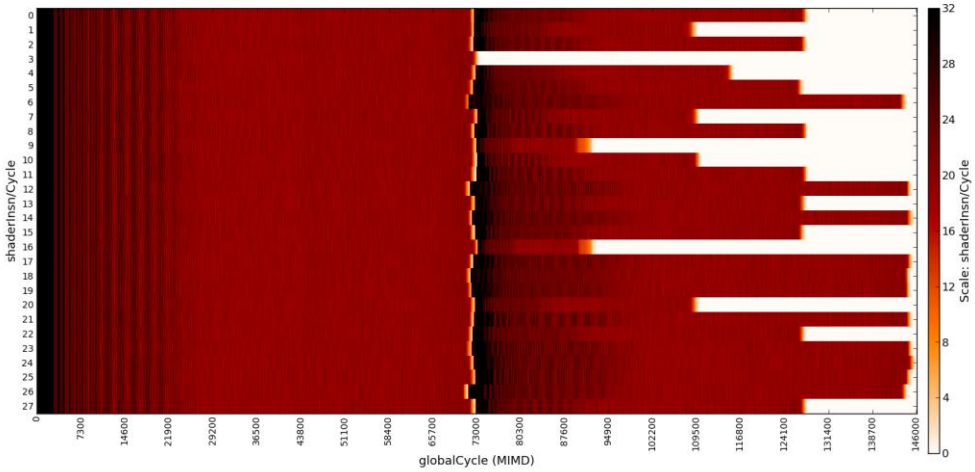
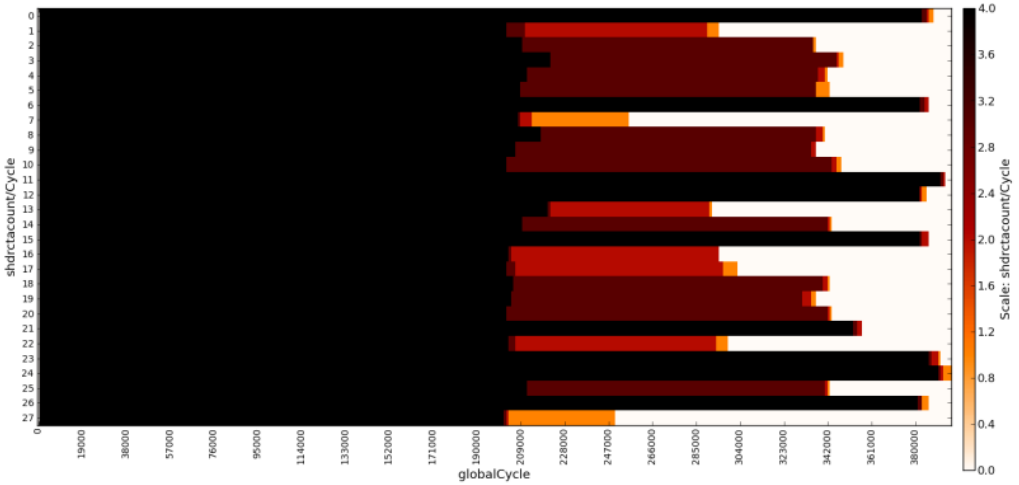


Figure 5: (a) IPC/SM for MUM benchmark under the POD scheduler (b) IPC/SM under the DWF scheduler. (c) IPC/SM under the MIMD scheduler.

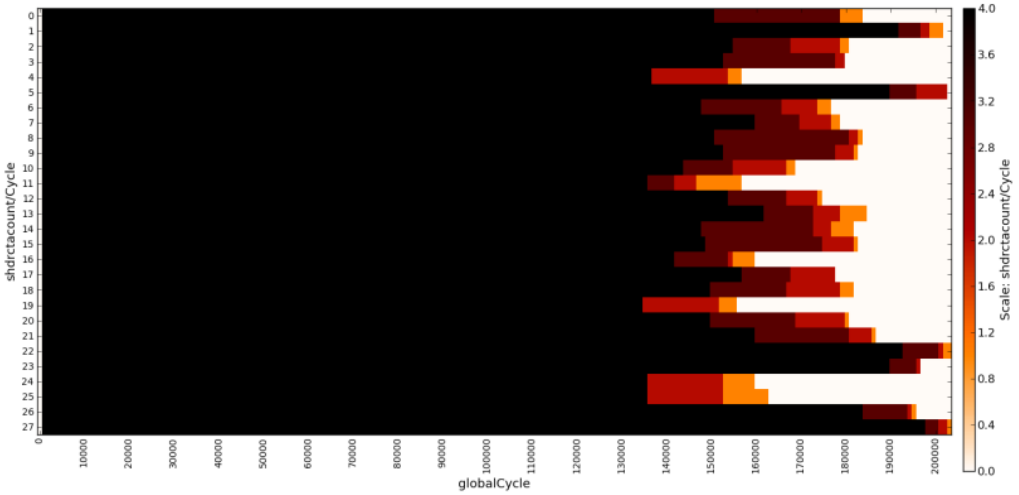
When using POD, DWF and MIMD scheduling algorithms it's easy to see the gradual loss of utilization from the moment a block is scheduled to the SM until its execution is finished and a new block is being scheduled. We can see that even MIMD which is the best possible scheduler under the perfect memory model is showing lower utilization since many threads in each block finish their execution early and the TLP offered by the amount of the remaining threads is not enough to fully utilize the SM.

Figure 6 shows the distribution of blocks during the execution of MUM benchmark for POD, DWF and MIMD benchmarks. As discussed in chapter 2, in the last stages of the simulation, various SMs are underutilized because there are no more new blocks that are available to be scheduled to SM.

(a)



(b)



(c)

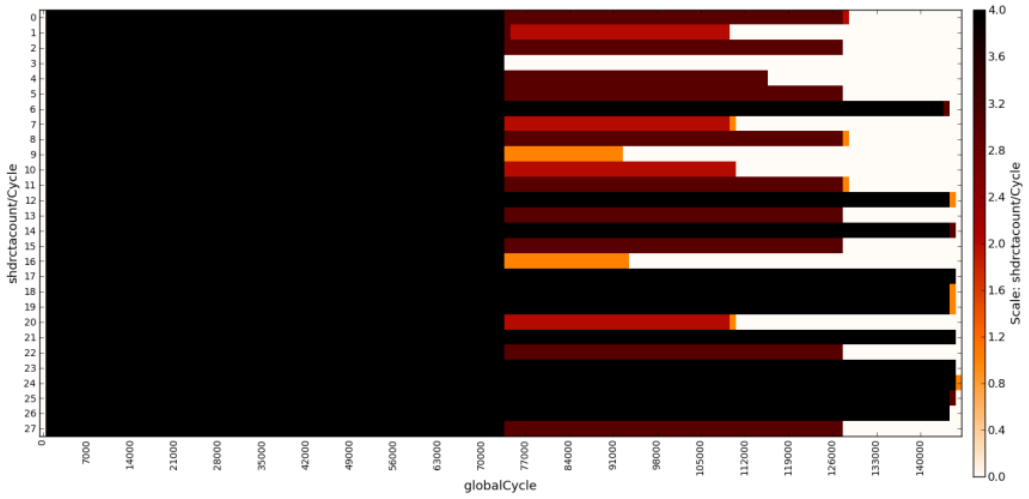


Figure 6: Distribution of blocks under the following scheduling methods: (a) POD (b) DWF (c) MIMD

Figure 7 shows warp occupancy distribution during the execution of MUM benchmark for POD scheduling algorithm.

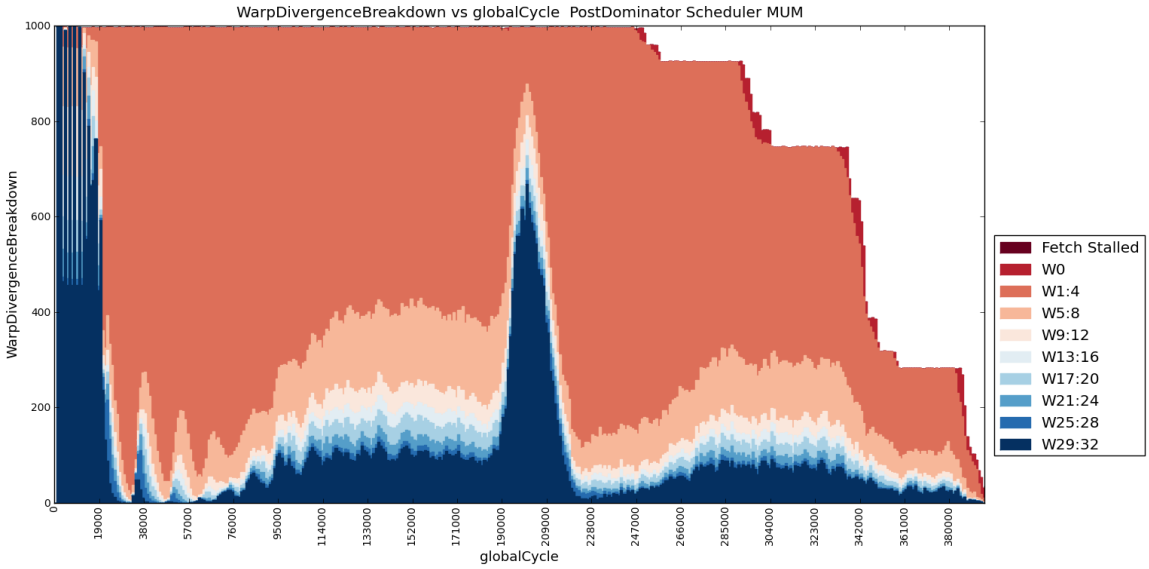


Figure 7: warp occupancy distribution for MUM benchmark under the POD scheduler

Each column in Figure 7 is a histogram showing the warp size distribution over a period of last 1000 cycles. It's easy to see branch divergence is causing severe underutilization of the pipeline with POD scheduling method.

The results presented in this section show that local schedulers have inherent limitations, especially when dealing with programs with abundant control flow such as MUM.

In MUM branch divergence causes severe execution path variance which in turn lowers the number of warps available to scheduling in the SMs during the blocks execution.

Even MIMD scheduler which is the optimal scheduler under the perfect memory model suffers from this phenomenon and is unable to keep the SMs fully utilized. Another reason for severe performance loss is scheduling the work to SMs in the resolution of blocks. This causes underutilization of the SMs in the last stages of the application runtime which is something local schedulers are unable to prevent.

The next section is exploring various ways to mitigate the effect of those phenomena.

3.4 How to improve the hardware utilization?

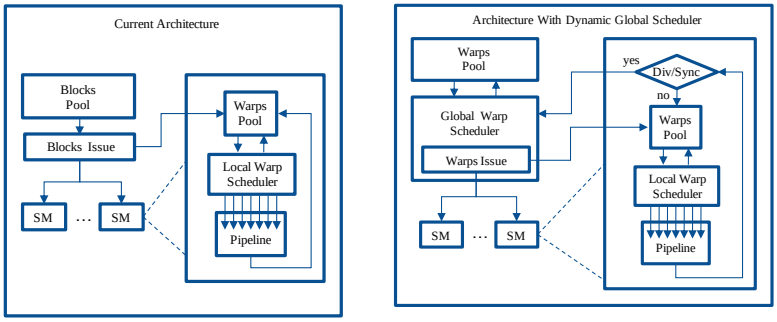
In order to improve the performance of control flow bound applications in GPGPU we must lower the impact of branch divergence and improve the load balancing among different SMs. DWF [1] and TBC[28] (Thread Block Compaction) scheduling methods address branch divergence by creating new warps out of diverged warps inside the SM. This allows to mitigate some of the performance loss caused by branch divergence; however the ability to perform local reconstruction heavily depends on the spatial distribution of branch targets in the grid.

The use of shared memory is known to be one of the most important optimizations needed for achieving high performance on GPGPU. A closer look at the behavior of control flow bounded applications shows that many of them are not using shared memory. For example, let's look at BFS benchmark. Here, shared memory is not beneficial as the data required by each vertex can be presented anywhere in the global edge array. As a consequence, finding the locality of data to be collectively read into the shared memory is as hard as the BFS problem itself.
In general, algorithms with high rate of branch divergence often have random memory access patterns preventing an efficient use of shared memory, caches and memory access coalescing. The inability to use shared memory is a major source for inefficiency in such applications.

In current GPGPU design, the system assumes no direct data races between threads belonging to different blocks. However, when threads are not using shared memory there is usually no need for sync either. In that case it's possible to extend the assumption and consider no direct data races between threads at all. In practice, this allows to migrate threads between different SMs during the execution of the kernel and opens new directions for new and much more sophisticated optimizations for control flow abundant applications. For example, it's possible to implement an algorithm similar to work stealing [24] in order to improve load balancing. In addition, it's possible to perform warps reconstruction both locally inside SMs and globally in order to improve the handling of varying execution path length and ease the dependence on branch targets distribution in the grid.
In this work we show the potential benefits of using thread migration and global warp reconstruction. For that purpose we implemented a global hierarchical warp scheduling mechanism we termed ODGS (Oracle Dynamic Global Scheduler). This algorithm is described in the next section.

3.5 ODGS - Oracle Dynamic Global Scheduler

In order to study the potential benefits of global warp reconstruction, we designed ODGS algorithm to achieve the best possible SMs utilization using global reconstruction.
Figure 8a depicts the way scheduling works in current hardware; the program is divided into independent blocks, the system builds warps from threads belonging to the same block and puts them in the SM's warp pool, the local scheduler in each SM decides the order in which the warps are executed.



(a) (b)

Figure 8: The architectural differences between: (a) current architecture and (b) dynamic global scheduler

ODGS is depicted at Figure 8b. Here, warps belonging to the same block can be assigned to different SMs by a global scheduler. For that reason ODGS needs the ability to migrate threads between SMs. In kernels where thread migration is impossible (use of local shared memory) or having a negative effect, global restructuring can be simply turned off.

Upon divergence, the diverged warp is moved from the SM to the global pool. The global scheduler will then try to create new full warps by using threads from the diverged warp and from other threads in the global pool having the same instruction pointer (PC).
As soon as a warp in SM finishes its execution or moved to global pool, it is replaced by another warp from the global scheduler. First to be scheduled to SMs are full reconstructed warps, then new warps. If there are no more full warps to schedule, incomplete warps with the lowest PC are scheduled to allow later branches to be processed by the global scheduler.
This re-construction and scheduling policy allows better load balancing and utilization of the GPU hardware.
Under the new scheduling mechanism, block size mainly serves as a logical entity that reflects the problem structure and determine the barrier synchronization boundaries.
If a warp reaches a barrier it is recalled from the SM and replaced by a ready warp. When all the warps from the block have reached the barrier, those warps are scheduled to the SM's.

In the next section we evaluate ODGS performance by using simple synthetic CUDA benchmark.

3.6 Comparing ODGS to POD on synthetic CUDA benchmark

Figure 9 describes a simple synthetic benchmark that represents the building blocks of any application control flow, namely if-else constructs.
In this example the code contains two branches: A and B that form four possible execution paths.

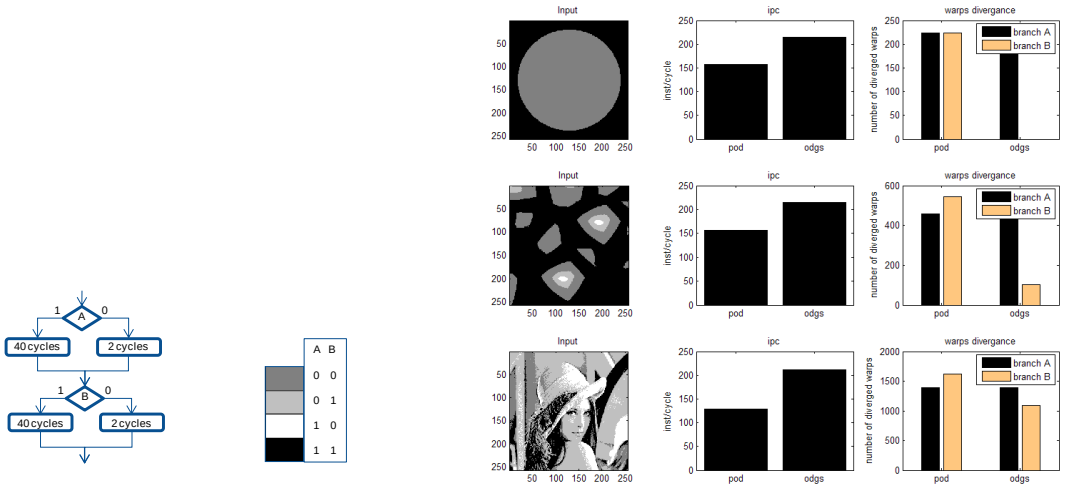


Figure 9: Synthetic benchmark example: (a) conditional kernel code with four possible execution paths (b) execution path as a function of color code (c) grid input (d) comparison of IPC for POD and ODGS scheduling (e) comparison of diverging warps number at branches A and B for POD and ODGS scheduling.

We present a unique technique to analyze the characteristics of ODGS and POD scheduling methods. We propose that input for this benchmark will be generated by consuming a 256x256 grey scale bitmap image with four possible colors, as shown in Figure 9c. Each thread will operate on a pixel and the value of the "color" will determine the outcome of the branches as shown in Figure 9b.
ODGS uses threads with identical PCs to reconstruct warps. This mechanism increases the correlation between threads within the warps and consequently lowers the overall number of

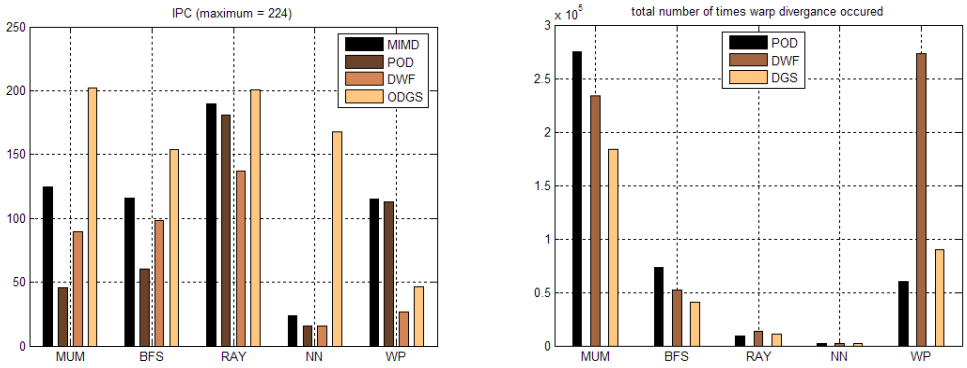
warps that diverge at later stages of the kernel execution. This phenomenon is clearly shown when using picture 1 as input. Here, there are only two colors and so threads can have only two possible execution paths.
Figure 10 shows the threads that diverged during the benchmark execution for input picture 1 with white pixels representing the diverged threads.

Figure 10: Diverging threads for input picture number 1

As shown in Figure 10, warps that contain threads of different colors will diverge at A both in POD and ODGS. However, when reaching branch B no warps diverge in ODGS scheduling. This situation is illustrated in Figure 10.c and Figure 10.d. The reason for that is that there is a perfect correlation between the threads that behave identically at branch A and later at branch B. Figure 10.b shows the IPC achieved with POD and ODGS. Results indicate a 30% performance advantage ODGS has over POD despite the fact that only relatively few warps actually diverge.
The rest of the pictures use all 4 possible colors, with each picture having increased spatial divergence between the colors, effectively increasing the number of diverged warps in each simulation. Obviously the number of warps diverging at A is the same with POD and ODGS. However, the reconstruction done by ODGS clearly succeeds to lower the number of diverging warps in branch B due to the correlation effect.

3.7 Comparing ODGS to POD, DWF and MIMD on real CUDA benchmark

Figure 11 compares the performance for MIMD, POD, DWF and ODGS scheduling algorithms running benchmarks described in chapter 3.1. Figure 11.a shows the average IPC for each of the benchmarks and Figure 11.b shows the total number of times warp divergence happened in each benchmark.



(a) (b)

Figure 11: (a) IPC comparison for various scheduling methods (b) total number of times warp divergence happened in each benchmark

Simulation results indicate that for most benchmarks ODGS outperforms other scheduling algorithms since ODGS is capable to load balance the workload and reduce the number of diverged warps more than other algorithms. Table 3 shows IPC in each benchmark relative to POD.

Table 1: Normalized IPC for different scheduling methods relative to POD

Benchmark	MIMD	POD	DWF	ODGS
MUM	2.71	1	1.95	4.4
BFS	1.92	1	1.64	2.6
RAY	1.05	1	0.76	1.10
NN	1.52	1	0.98	10.6
WP	1.02	1	0.24	0.4

Both MUM and BFS are parallel graph algorithms suffering from significant branch divergence. In those algorithms ODGS achieves much higher utilization by re-constructing the diverged warps and hence the performance gain of x4.4 and x2.6 relative to POD.

As shown in Figure 5c and 6c, in the last stages of MUM benchmark, various SM's are underutilized because there are no more new blocks that are available to be scheduled. This can explain why ODGS outperforms MIMD which achieves the best possible performance with regard to the possible local scheduler optimizations under the perfect memory model. The performance of MIMD may even become worse when the application has a high rate of branch divergence since many threads in each block finish their execution earlier and the TLP offered by the remaining threads is not enough to fully utilize the SM.

In RAY benchmark the gain relative to POD is only 10%, however compared to DWF the performance gain is almost 46%. Figure 12 shows unique insight into the dynamic behavior of various scheduling methods for the RAY benchmark. The output of the RAY benchmark is presented in Figure 12.a. Each pixel in the final picture is calculated by different thread.

(a) (b) (c) (d) (e)

Figure 12: (a) output of RAY tracing benchmark (b) execution path length (c) threads divergence number distribution for DWF benchmark (d) threads divergence number distribution for POD benchmark (e) threads divergence number distribution for ODGS benchmark

Figure 12.b shows the distribution of the runtime path length for each thread and Figure 12.(c-d) represents the number of times each thread diverged with brighter color representing higher values. As expected, the Figure 12 shows visually that there is a correlation between the final geometric shapes, the threads that worked the most and the threads that diverged the most. It's easy to see that the picture of the threads divergence distribution for ODGS method resembles the picture of execution path distribution much more than those of other methods. This indicates that ODGS have better ability to group working threads together and keep in the system only those threads that actually perform useful work.

For NN the performance gain for ODGS is very high, almost x10.6 despite the fact that this benchmark doesn't show significant branch divergence as shown by Figure 11b.

NN is an example of an application where the size of the blocks was chosen to be convenient for the programmer to better reflect the structure of the problem rather than to optimize for the particular hardware. In this benchmark a neural network consisting of 4 layers is used to recognize digits. The last 2 layers of this network consist of a group of neurons each connected to all the neurons of the previous layer. Each layer is computed by a different kernel where the last 2 kernel are running with distinct block for each neuron, each containing a single thread. Local schedulers are incapable of recovering the performance loss associated with such a choice, ODGS on the other hand can optimize much better in such situations.

WP is an example of an application where using ODGS in its current form was very inefficient despite the fact that it performs better than DWF (it's possible that DWF could perform better here

by using the PC scheduling heuristic). WP benchmark contains a relatively small number of threads with long kernel and very high number of short branches (see Table 1).
For that reason, in the last stage of grid computation there is a high number of incomplete warps remaining to be scheduled to the SMs. This leads to significant performance loss because the amount of cycles needed to compute this remaining part is very significant relative to the overall runtime. We discuss ways for improving ODGS performance in such circumstances in the next section.

3.8 Sensitivity study and discussion of ODGS implementation in hardware

Previous chapters have shown the relatively high performance of ODGS in most benchmarks in comparison with other methods under a perfect memory model and zero latency overhead for thread migration and reconstruction. In this chapter we extend the discussion and study ODGS performance using more realistic conditions (memory delays, time to reconstruct and re-schedule threads, etc).
The main issue we are facing is that on one hand, the use of fine-grained multi-threading allows GPGPU to better tolerate memory and pipeline latencies but on the other hand implementing ODGS scheduling and handling diverged warps involves increasing the latency overhead because of regrouping and copying threads context. In order to evaluate the impact of such latency on ODGS performance we perform simulations for different diverged warp handling latencies.
We model diverged warp handling latency in the following way: once a warp diverges, it is moved to the global scheduler and its "slot" in the SM is freed. No other warp can be rescheduled into this free slot until the handling latency expires. In addition, during this period threads belonging to the diverged warp cannot be used to form new warps or be rescheduled to SMs.
Figure 13 shows the impact of different diverged warps handling latency on ODGS performance in various benchmarks.

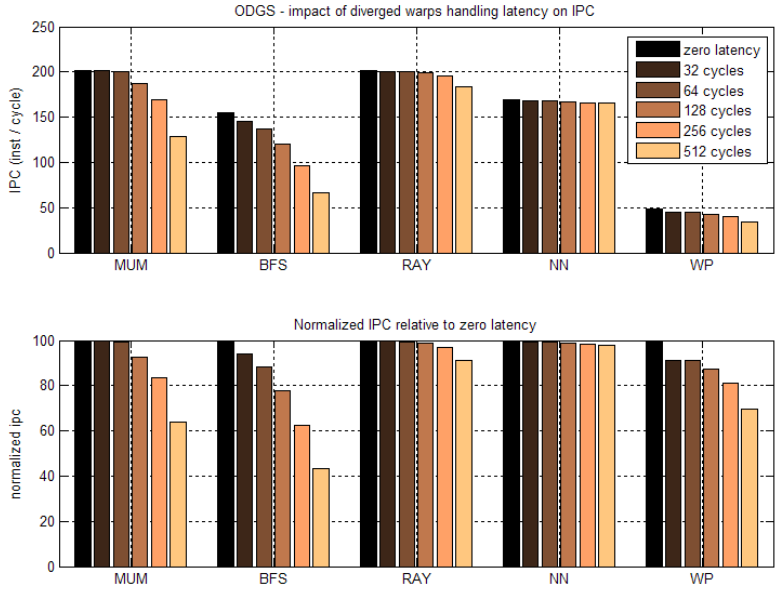


Figure 13: (a) Impact of diverged warps handling latency on ODGS performance (b) normalized IPC relative to zero handling latency

Figure 14 shows warp context size, average warp divergence rate per SM and the additional average memory bandwidth needed (per SM) for moving the diverged warps to the global scheduler when using ODGS (assuming zero handling latency overhead).

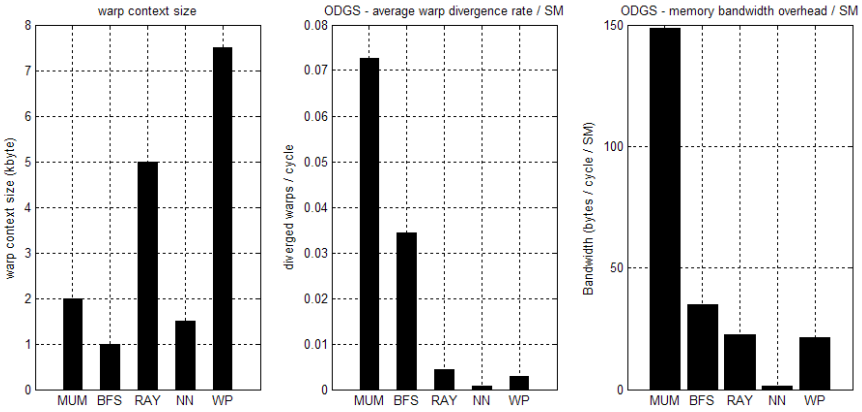
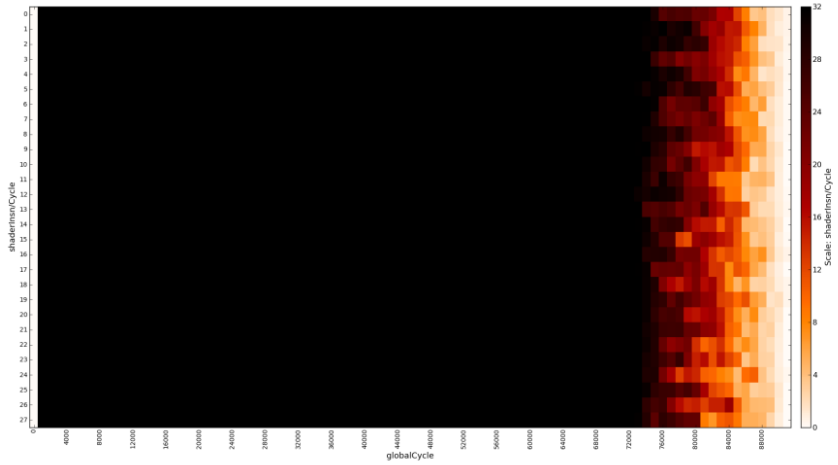


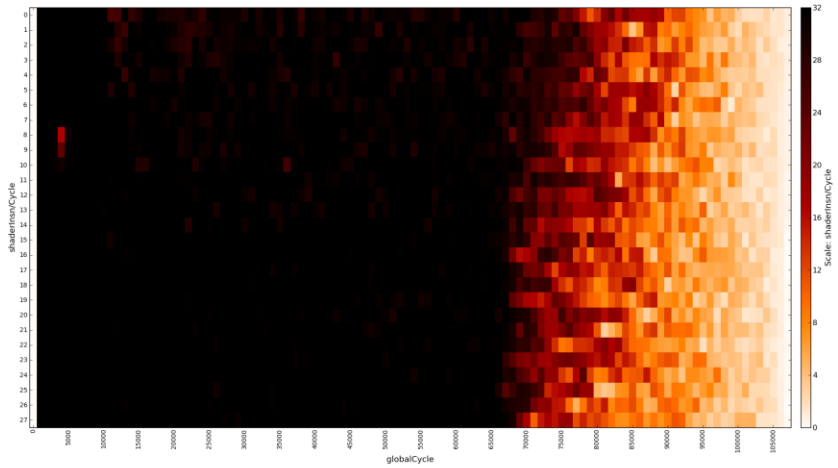
Figure 14: (a) Warp context size in each benchmark (b) Average warp divergence rate per SM when using ODGS (c) Memory bandwidth overhead per SM for moving diverged warps to global scheduler

Figure 14.b shows once again that MUM and BFS suffer from significant branch divergence rate. Figure 13 shows that in those benchmarks handling latency above 128 – 256 cycles per warp leads to significant performance drop. Figure 15 allows to analyze the effect of increasing latency on performance. The figure compares the IPC/SM during the execution of the MUM benchmark for diverged warps handling latencies of 64, 256 and 512 cycles. Each row in the figure corresponds to IPC of one of the 28 SMs during the MUM benchmark execution.

(a)



(b)



(c)

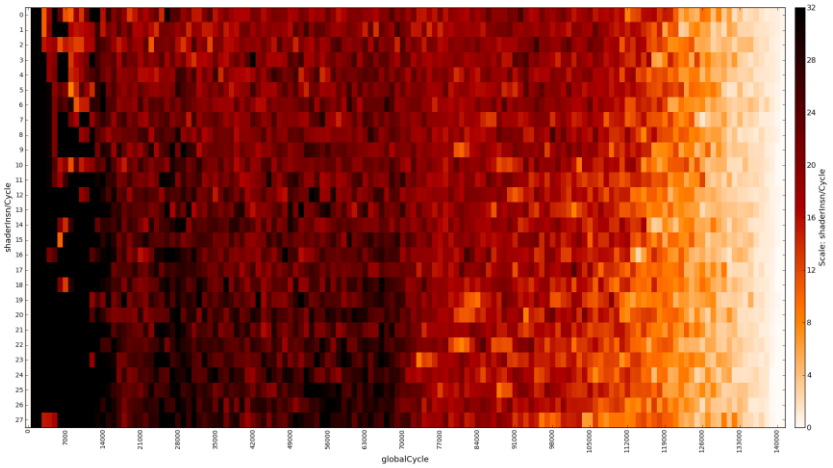


Figure 15: IPC/SM for MUM benchmark under the ODGS scheduler (a) warp handling latency is 64 cycles (b) warp handling latency is 256 cycles (c) warp handling latency is 512 cycles

Figure 15 shows that for MUM benchmark, when diverged warps handling latency is lower than 256 cycles, SMs have enough ready warps for full utilization and performance is lost mainly in the last stages of the kernel execution when the global scheduler has no more new warps to schedule. Above the 256 cycles latency threshold, high latency leads to a state where the TLP available inside the SMs is insufficient to keep the pipelines fully utilized and hence the significant performance drop, as illustrated by Figure 15.c.
In benchmarks such as RAY and NN divergence rate is very low. In those benchmarks handling latency has a relatively insignificant effect, mainly in the last stages of the kernel execution. In the WP benchmark, handling latency also influences mainly the last stages of the kernel execution, however the effect of latency is more significant here due to the high number of rare branches and their impact on ODGS as discussed in the previous section.

Figure 14.c shows the additional average memory bandwidth needed per SM to move diverged warps context to the global scheduler. In MUM the required bandwidth to move all diverged warps is relatively high: ~ 150 bytes / cycle due to the high divergence rate and 2kbyte context size for each warp. In other benchmarks the required bandwidth is much lower due to the lower divergence rate.
Results indicate that ODGS performance stays high as long as the divergence rate and handling latency are low enough to keep SMs fully utilized. The exact latency that can be tolerated depends on the size of the register file, the supported and required memory bandwidth and the divergence rate.

It's important to note that ODGS improves performance even if only part of the diverged warps is regrouped. In real implementation, ODGS will need a dynamic scheduling policy that keeps the number of warps returned to the global scheduler under the threshold that allows better performance. For example, if divergence rate and the pressure on the memory system is high, ODGS can decide that some branch points are more important to deal with than others. This decision can be based on statistics gathered during the kernel runtime or even according to ranking performed during compilation based on the compiler evaluation of the various execution path lengths. For example if the branch length is below a certain threshold ODGS can decide that it will be resolved by using predication and serialization similarly to current hardware. Such approach would allow to achieve high performance in benchmarks such as WP where the policy of trying to regroup warps after each divergence fails to achieve good performance.

ODGS fits naturally into a hierarchical scheduling model where local scheduling optimizations such as DWF or TBC[28] are employed. Combining ODGS with such optimizations has the potential to significantly lower the number of threads ODGS needs to move to the global

scheduler for reconstruction. In particular, such combination will allow a more efficient use of L1 caches and lower the pressure on the memory system.
For global reconstruction to work well, ODGS needs the ability to migrate threads between SMs. For that reason it's primarily designed to improve the performance of programs with high divergence rate that make no use of local shared memory. In programs that use shared memory ODGS can be used to handle divergence points that lie after the last shared memory access. It's important to note that global restructuring can be simply turned off in kernels where such optimizations are having a negative effect.

4. RELATED WORK

Several prior works addressed GPGPU scheduling inefficiencies. J.Meng et al [7] propose dynamic warp subdivision (DWS) which allows a single warp to occupy more than one slot in the scheduler without requiring extra register file space. Independent scheduling entities allow divergent branch paths to interleave their execution, and allow threads that hit to run ahead.
W. Fung et al. [28] proposed thread block compaction (TBC), which uses a block-wide re-convergence stack shared by all threads in a thread block to exploit their control flow locality. Warps run freely until they encounter a divergent branch, where the warps synchronize, and their threads are compacted into new warps. At the re-convergence point the compacted warps synchronize again to resume in their original arrangements before the divergence.
Those algorithms perform optimizations which are local to each SM, for that reason their ability to improve performance depends heavily on the distribution of branch targets in the grid. In contrast, ODGS performance depends only on the global availability of threads branching to the same PC, allowing better performance in problems with abundant control flow such as parallel graph algorithms.
Kapasi et al. [8] introduce conditional streams, a code transformation that creates multiple kernels from a single kernel with conditional code and connects these kernels via inter-kernel communication to increase the utilization of a SIMD pipeline. While being more efficient than predication, it may only be practical to implement on architectures with a software managed on-chip memory designed for inter-kernel communication. ODGS differs from conditional streams in that it is a hardware mechanism exploiting the dynamic conditional behavior of each scalar thread in the grid.

5. SUMMARY

We start this work by analyzing scheduling inefficiencies present in NVIDIA GPGPU architecture. We show that SM local scheduling methods have inherent limitations when dealing with control flow bound applications. To deal with those limitations we propose a new approach based on hierarchical scheduling and global warp reconstruction that allows to optimize the execution of such applications.

Our findings indicate that if future graphics will be built to allow global warps reconstruction and threads migration between SMs it may ease the optimization problem and allow to improve the performance of applications that otherwise would not be able to efficiently run in GPGPU environment, or at least will require spending long time at the software optimization phase.

We implemented and evaluated an ideal global warp scheduling algorithm (ODGS) designed to maximize the amount of effective parallelism in the system through the use of global warp reconstruction. We showed by simulations of synthetic and real benchmarks that there is a significant potential for substantial performance improvement in control flow bound applications that make no use of local shared memory.
We showed by simulations that compared to POD immediate post dominator scheduling method, ODGS achieves almost x10.6 performance boost in a neural network application. MUMmerGPU and BFS are parallel graph algorithms suffering from significant branch divergence. In those

algorithms ODGS achieves performance gain of up to x4.4 and x2.6 relative to other scheduling methods.

The sensitivity analysis we provided at the end of Section 3, indicates that implementing such a hierarchical global warp scheduling is feasible, though implementation details of the needed specific hardware support are beyond the scope of this paper. Further research on new memory models, tuning of the proposed algorithm and other extensions may lead to better understanding of the right balance between hardware complexity, power and performance benefits of using the new proposed technique.

6. ACKNOWLEDGMENTS

The authors would like to thank the development team of GPGPU-SIM for their help in the Google Groups GPGPU-SIM forum.

REFERENCES

[1] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In Micro 2007.

[2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, In proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 163-174, Boston, MA, April 26-28, 2009.

[3] NVIDIA CUDA C Programming Best Practices Guide. CUDA Toolkit 2.3. 2009

[4] Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K.Skadron. 2008, A performance study of general purpose applications on graphics processors using CUDA. JPDC.

[5] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. Department of Electrical and Computer Engineering, University of Toronto. Ispass 2010.

[6] Frank Dehne, Kumanan Yogaratnam, Exploring the Limits of GPU's With Parallel Graph Algorithms. Computer Science Carleton University, Ottawa, Canada.

[7] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance: Extended results. U.Va. Tech. Report CS-2010-5, 2010.

[8] U. J. Kapasi, J. Dally, W, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In MICRO 33, 2000.

[9] Monica Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines. Department of Computer Science, Carnegie Mellon University.

[10] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A Characterization and Analysis of PTX Kernels. School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia.

[11] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. . Center for Visual Information Technology International Institute of Information Technology Hyderabad, INDIA. In HiPC, pages 197–208, 2007.

[12] Reiter Horn Mike Houston Pat Hanrahan. ClawHMMER: A Streaming HMMer-Search Implementation. Stanford University.

[13] Michael C Schatz, Cole Trapnell, Arthur L Delcher, Amitabh Varshney. High-throughput sequence alignment using Graphics Processing Units.

[14] NVIDIA's next generation CUDA compute architecture: Fermi. NVIDIA Corporation, 2009.

[15] Rixner1, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling, Computer Systems Laboratory, Stanford University.

[16] Jung Ho Ahn, Thesis, Memory And Control Organizations Of Stream Processors. 2007, Stanford University.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Shea_er, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pages 44{54, Oct. 2009.

[18] M. Schatz, C. Trapnell, A. Delcher, and A. Varshney. High-throughput sequence alignment using Graphics Processing Units. *BMC Bioinformatics*, 8(1):474,2007.

[19] Aaftab Munshi. OpenCL, Parallel computing on GPU and CPU. SigGraph 2008.

[20] Kamran Karimi Neil G. Dickson Firas Hamze, A Performance Comparison of CUDA and OpenCL. British Columbia Canada.

[21] Billconan and Kavinguy. A Neural Network on GPU. http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[22] Maxime. Ray tracing. http://www.nvidia.com/cuda.

[23] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. IPDPS 2008: IEEE Int'l Symp. on Parallel and *Distributed Processing*, pages 1–7, April 2008.

[24] Scheduling Multithreaded Computations by Work Stealing
Robert D. Blumofe, The University of Texas at Austin. Charles E. Leiserson MIT Laboratory for Computer Science

[25] Aaron Ariel, Wilson W. L. Fung, Andrew Turner, Tor M. Aamodt, Visualizing Complex Dynamics in Many-Core Accelerator Architectures, In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 164-174, White Plains, NY, March 28-30, 2010.

[26] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi and Kunle Olukotun, Accelerating CUDA Graph Algorithms at Maximum Warp, PPoPP 2011.

[27] Sunpgack Hong, Tayo Oguntebi, and Kunle Olukotun, Efficient Parallel Graph Exploration for Multi-Core CPU and GPU, PACT 2011.

[28[Wilson W. L. Fung, Tor M. Aamodt, Thread Block Compaction for Efficient SIMT Control Flow, In proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA-17), pp. 25-36, San Antonio, Texas, February 12-16 2011.