Perf-Sat: Runtime Detection of Performance Saturation for GPGPU Applications

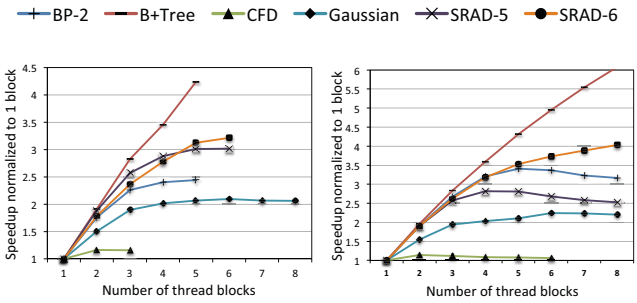
Mihir Awatramani, Joseph Zambreno, Diane Rover
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
Email: {mihir, zambreno, drover}@iastate.edu

***Abstract*—Graphic Processing Units (GPUs) achieve latency tolerance by exploiting massive amounts of thread level parallelism. Each core executes several hundred to a few thousand simultaneously active threads. The work scheduler tries to maximize the number of active threads on each core by launching threads until at least one of the required resources is completely utilized. The rationale is, more threads would give the thread scheduler more opportunities to hide memory latency and thus would result in better performance. In this work, we show that launching the maximum number of threads is not always necessary to achieve the best performance. Applications have an optimal thread count value at which the performance saturates. Increasing the number of threads beyond this value results in no better and sometimes worse performance. To this end, we develop Perf-Sat: a mechanism to detect the optimal number of threads required on each core at runtime. Perf-Sat is integrated into the hardware work scheduler and guides it to either increase or decrease the number of active threads. We evaluate the performance impact of our scheduler on two GPU generations and show that Perf-Sat scales well to different applications as well as architectures. With performance loss of less than 1%, Perf-Sat is able to achieve core resource savings of 18.32% on average.**

***Index Terms*-GPGPU, Resource Utilization, Workload Scheduling**

I. Introduction

Graphics processing units (GPUs) have proven to be excellent accelerators for general purpose computing. As they are designed for high throughput, GPU architectures incorporate several cores, each of which have hundreds of arithmetic units and support thousands of active threads. Due to transistor scaling, each new generation of GPU provides a higher computational power compared to the previous one. For example, the NVIDIA Kepler architecture [1] has six times the number of single precision and four times the number of double precision floating point units per core as compared to NVIDIA Fermi [2]. GPGPU programming models, e.g. CUDA [3] and OpenCL [4], use the *thread block* abstraction to enable applications written for a previous generation GPU to scale to the increased computation resources provided by future generation GPUs. The number of blocks that can be active simultaneously on a core depends on the resources available. Newer generation GPUs provide more resources per core, thereby supporting a higher number of concurrently active thread blocks. Thus, by executing a higher number of thread blocks (and hence threads) on a newer generation GPU, the same application is able to utilize the increased number of computation units and thereby achieve a higher throughput.



(a) M2090 (Fermi) (b) K20X (Kepler)

Fig. 1. Impact of thread count on performance when simulated on configurations matching the NVIDIA Tesla M2090 (a) and K20X (b) GPUs.

However, we observed that increasing the number of threads does not always result in increased throughput. Each application has an *optimal* thread count value, beyond which increasing the number of threads does not result in further improvement in performance. To observe the effect of number of threads on application performance, we simulated various application kernels[1] with an increasing number of thread blocks on GPGPU-Sim, a cycle-accurate GPU simulator [5]. The configurations were chosen to match the NVIDIA Tesla M2090 (Fermi) and K20X (Kepler) GPUs, as they are designed specifically for high performance general purpose computing. Fig. 1 depicts the results for six out of the 16 kernels that we used for this work. Three types of workloads are shown in Fig. 1(a). One type of workload (SRAD-6 and B+Tree) exhibits continuous improvement in performance as the number of blocks is increased. For the second type (SRAD-5 and BP-2) performance improves initially and then saturates. Behavior of the third type (CFD and Gaussian) is similar through the saturation point, except the performance drops off as the number of blocks is increased beyond that. Interestingly, the kernels for which performance saturated on the M2090 (SRAD and BP-2) behave like the third type of workload on the K20X (Fig. 1(b)) as now they execute a higher number of thread blocks.

Prior studies have made similar observations [6]–[8]. Cache Conscious Wavefront Scheduling [6] proposes a warp

[1]Kernels are the functions of an application that are offloaded to the GPU.

1530-2016/14 $31.00 © 2014 IEEE
DOI 10.1109/ICPP.Workshops.2014.14

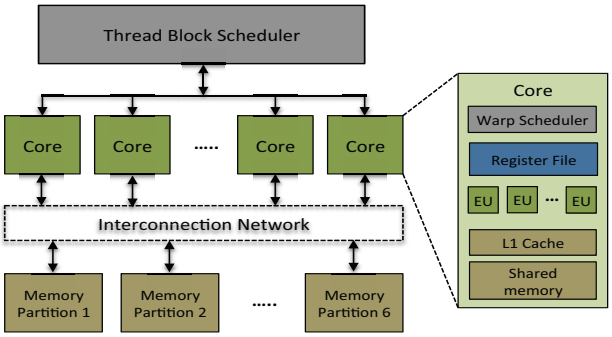


Fig. 2. Block diagram of GPU hardware

scheduling policy (see Section II-B for description of the warp scheduler) that reduces the number of active warps, when threads from different warps contend for the same L1 data cache set and increase evictions of possibly useful cache lines. DYNCTA [7] tries to find the optimal number of thread blocks at the thread block scheduler by monitoring the core idle and memory wait cycles. However, both techniques use thresholds that are determined heuristically and hence require executing the applications for different architectures.

We believe that it is not possible to use one threshold for different applications, even if they are being executed on the same architecture. Thus, we propose Perf-Sat: a hardware mechanism that detects the optimal thread block count at runtime by using techniques similar to control systems. Per-Sat uses information about core activity at the previous thread block count along with feedback from the cores and makes decisions based on relative performance. To reduce the effect of not using thresholds, it uses history bits to record previous decisions. The thread block count detected by Perf-Sat is used by the thread block scheduler to limit the number of active blocks on each core. This work is a first step towards a hybrid scheduler that would use Perf-Sat at the chip level to determine cores with unused resources. These resources would then be used to execute threads at a lower priority from a concurrently executing kernel, in an interleaved manner [9], thereby increasing throughput whenever possible without sacrificing performance of the first kernel.

The remainder of this paper is organized as follows: Section 2 gives an overview of the GPU hardware architecture and work scheduling. The motivation of our work is described in Section 3. Section 4 provides details of Perf-Sat. Experimental results are discussed in Section 5. An overview of related work is provided in Section 6. Section 7 summarizes contributions of this work and provides directions for future work.

II. Background

This section gives an overview of the GPU architecture and describes the workload scheduling techniques that are used for general purpose computing on GPUs.

A. GPU Hardware Architecture

Fig. 2 shows a high-level block diagram of the GPU chip and main components within a core. GPUs have a many

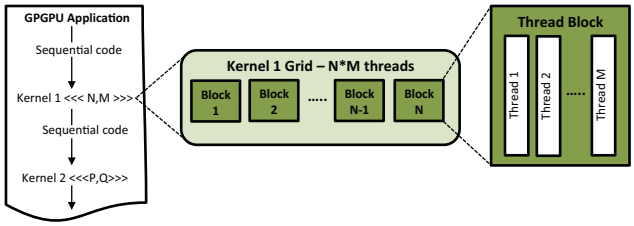


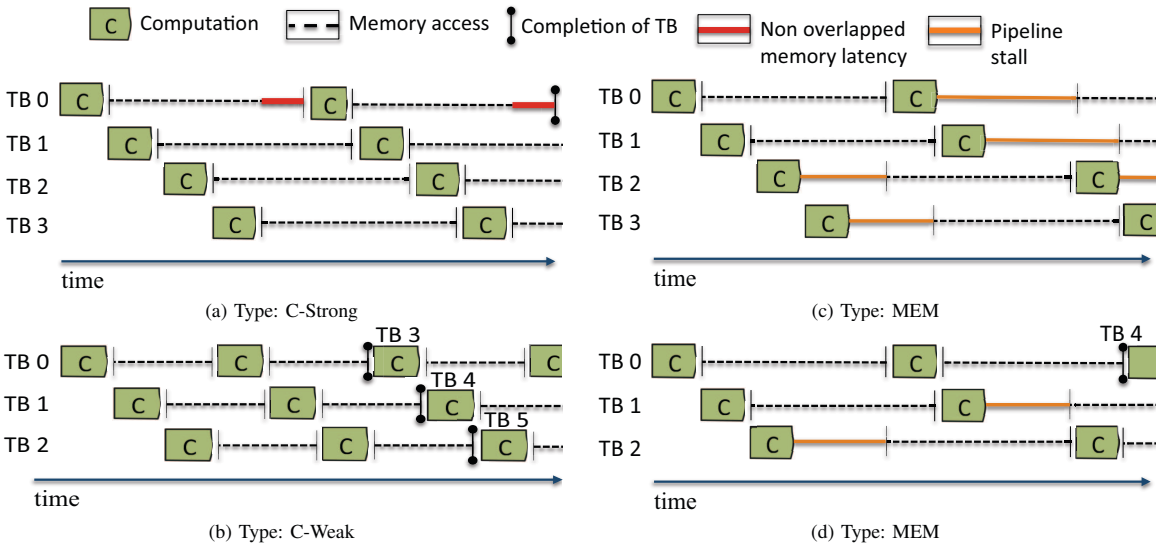
Fig. 3. Thread hierarchy used by GPU programming models

core architecture consisting of several in-order cores called Streaming Multiprocessors (SMs). Each SM consists of several arithmetic units denoted as EU (execution units) in the figure. These units can be single or double precision floating point units, load/store units or special function units used for transcendental operations like sine and cosine. To store the context of a large number of active threads, each SM has a very large register file. Additionally, each SM has a private L1 data cache and a high-bandwidth on-chip memory that can be shared by threads from the same thread block. If a memory request misses the L1 data cache or shared memory, it is routed via the interconnection network to a one of the memory partitions. Each memory partition has a L2 cache bank and a memory controller. The Fermi and Kepler configurations used in our work have six 64-bit-wide memory controllers providing a peak bandwidth of 177 GB/s and 250 GB/s respectively. Further details of the configurations are provided in Table I.

B. Workload Scheduling

GPGPU programming models (eg. CUDA and OpenCL) divide the computation using a three level hierarchy as shown in Fig. 3. Usually, a thread performs work corresponding to a single input/output data point. Thread blocks are groups of threads that can synchronize using barriers and share data through the on-chip shared memory. A grid, the highest level in the hierarchy consists of several thread blocks and represents all threads of a particular kernel.

Thread block scheduling: GPUs use a Single Program Multiple Data (SPMD) model of computation. Work is launched from an application via a *kernel* function call. The kernel, is the single set of instructions that is executed by all threads in the grid corresponding to that kernel. The thread block scheduler is the hardware module that issues work to the SMs. As threads within a block are allowed to use synchronization barriers, work is issued at the granularity of thread blocks. The publicly available information on the implementation of the thread block scheduler is sparse. Generally, it is assumed that thread blocks are issued to cores in a round-robin fashion until a core has resources available for an entire block. Four resources are checked: number of registers used per thread, shared memory used per thread, number of available thread slots and number of available thread block slots. Thus, the maximum number of thread blocks (and hence threads) that can be issued simultaneously on an SM depends on the size of the thread block as well



(a) Type: C-Strong

(b) Type: C-Weak

(c) Type: MEM

(d) Type: MEM

Fig. 4. Depiction of memory latency tolerance for different kernel categories.

as the resource requirement of each thread. We refer to this maximum block count as N_{max}.

Warp scheduling: On the SM, threads are grouped into fixed-size groups called warps, usually having 32 threads. A warp scheduler issues warps that are ready on the array of execution units in Single Instruction Multiple Data (SIMD) fashion. Although instructions from threads that belong to a single warp are executed in lock-step, execution of different warps is interleaved to achieve an overlap of memory access latency with computation.

III. Motivation

A. The Need for Thread Level Parallelism

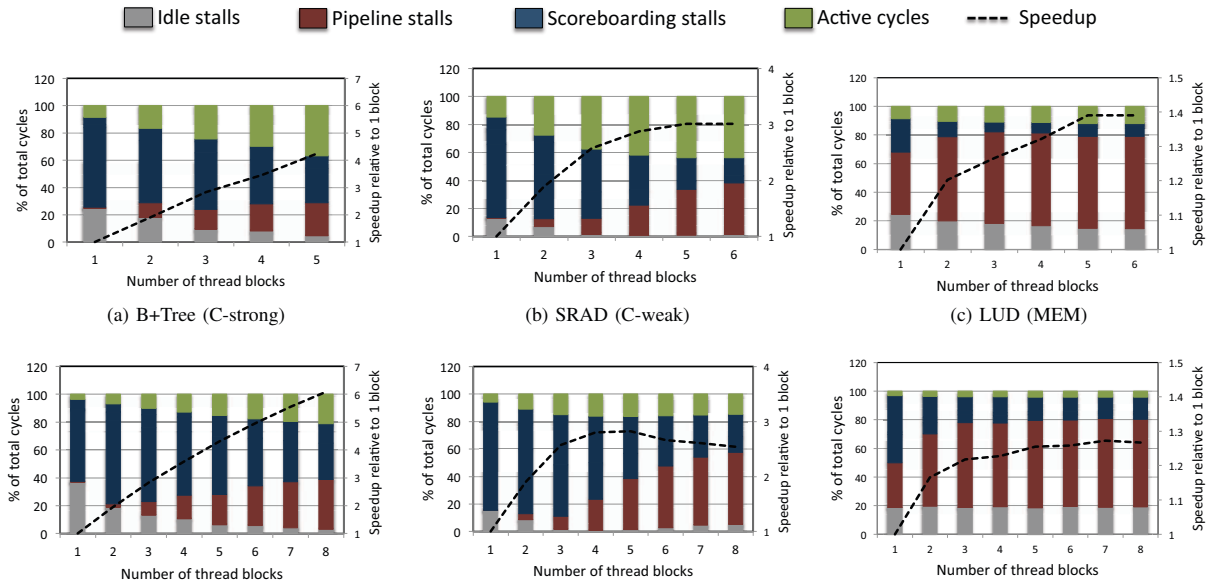
Graphics processing units are designed for high throughput rather than low latency. Hence, they trade-off the optimizations used in CPUs for reducing latency, such as large caches, out of order execution and branch prediction, to provide more die area to computational units. To account for the overhead of increased latency of memory accesses, GPU cores use a technique called Single Instruction Multiple Threads (SIMT) execution [2]. SIMT interleaves execution of multiple SIMD groups, each of which can be at a different instruction.

As mentioned in section II-B, a SIMD group of threads is called a warp. Warps within one thread block are executed in round-robin order, until all of them stall due to memory access. At this point, warps from the oldest thread block that was scheduled are selected for execution and so on. In this manner, the warp scheduler tries overlap memory access latency of the stalled threads with computation of the active threads. This scheduling technique is known as Greedy Then Oldest. Many other warp scheduling policies exist such as Round Robin, Two-level, etc. We do not describe them in detail as the work reported in this paper is neutral with respect to the warp scheduling policy. Further details regarding different policies can be found in [5], [6], [10], [11] and [12].

B. Optimal Thread Block Count

The warp scheduling approach described above is widely known in the GPGPU community. Hence, application developers tune the thread block size and resource requirement per thread to maximize the number of active threads that can be supported on a GPU core. Furthermore, the thread block scheduler launches thread blocks on a core until this maximum count is reached (N_{max}). The rationale is, more threads would provide the warp scheduler with more opportunities to overlap memory access latency with computation. However, as shown in Fig. 1, launching the maximum number of thread blocks (and hence threads) does not result in the best performance for all applications. For some applications, increasing the number of threads beyond some point does not result in further performance improvement, and for others might also reduce performance. We call this point, the ***optimal*** thread block count.

In our experiments, we observed three categories of applications. The effect of thread block count on each category is illustrated in Fig. 4. Let us assume that the applications have N_{max} equal to four. In Fig. 4(a), memory requests of warps from the first block are not completed even after all the other blocks have completed their computations. The performance of this category of applications would scale if additional thread blocks could be issued. They are referred to as **C-strong**. The optimal count for these applications is equal to N_{max}. Fig. 4(b), depicts the execution of the second type of application: **C-weak**. Notice that three blocks are sufficient to overlap all the memory latency. Launching the fourth thread block would not result in a better performance and hence the optimal count is three. Performance of such applications saturates once the optimal block count is reached. For the third category of workloads (refer Fig. 4(c) and 4(d)), performance starts to decrease if the number of active blocks is more than the optimal count. In the figures, we can observe



(a) B+Tree (C-strong) (b) SRAD (C-weak) (c) LUD (MEM)

(d) B+Tree (C-strong) (e) SRAD (MEM) (f) LUD (MEM)

Fig. 5. Breakdown of core activity for three types of workloads on the NVIDA Fermi (above) and Kepler (below) architectures.

that the memory system can support requests from only two blocks. Issuing more blocks, starts increasing pipeline stalls. The optimal count is the number of blocks that can issue memory requests simultaneously, plus one. In this example, it would be three. Such applications are categorized as **MEM**. In the next section, we describe the breakdown of core activity during execution, which explains the cause of this behavior.

C. Effect of Workload Characteristics on Optimal Thread Block Count

To understand the effect of number of thread blocks on application performance, we simulated a wide range of applications from the Rodinia benchmark suite [13] on a cycle accurate GPU microarchitecture simulator [5]. The thread block scheduler was modified to limit the number of blocks it launches on each core. Applications were executed from one to N_{max} active blocks and the following architectural parameters were monitored at each core:

Scoreboarding stalls: Cycles when all the warps are stalled due to dependency from a previous instruction. A stall is categorized as either ALU or memory depending on whether the previous instruction is an arithmetic or a memory operation.

Pipeline stalls: Cycles when all the warps are stalled because of insufficient hardware resources. Again, a stall is categorized as ALU if the resource is a computational unit and memory if the resource is required for a memory operation (e.g. MSHR entry, memory access queue, etc.).

Idle stalls: Cycles when all the ready warps are waiting on a synchronization barrier. The warp scheduler does not have any warps to issue.

Active: Cycles when the core is performing computations.

Fig. 5 illustrates the core activity profiles for an application from each of the three types. Data plotted for scoreboarding and pipeline stalls are attributed to both ALU and memory. For simplicity, we do not distinguish between these in the figure.

C-strong: We can observe in Fig. 5(a) that the number of scoreboarding stalls is very high with one block. This is because there are not enough threads to overlap the memory access and ALU latencies. As the number of thread blocks increases, the scoreboarding stalls decrease while the pipeline stalls increase. Notice that the decrease in scoreboarding stalls is always greater than the increase in pipeline stalls, which results in an overall increase in the number of active cycles. This trend continues until the thread block limit is reached. Hence, the optimal count is the same as N_{max} for these workloads. Similar behavior is observed on the Kepler architecture (Fig. 5(d)). It should be noted that, while the applications had a lesser execution time on the Kepler architecture, the ratio for which the core was active also reduced. This motivates additional investigation into execution throughput, which is a component of our planned future work.

C-weak: Fig. 5(b) shows that the performance of SRAD continues to improve and starts to plateau at five blocks (optimal block count). Observe that after five blocks, the decrease in scoreboarding stalls is comparable to the increase in pipeline stalls. This is because, as the number of active threads increases, the warp scheduler has a higher number of ready threads to choose from, which decreases the number of scoreboarding stalls. However, this also increases the contention for hardware resources, which results in an increase in the number of pipeline stalls. Due to this contradictory effect, the number of active cycles does not increase. Thus, the performance saturates after the optimal count value.

MEM: Similar to the C-weak category of workloads, the performance of workloads in this category also saturates when

the decrease in scoreboarding stalls becomes comparable to the increase in pipeline stalls. However, beyond this point the performance starts to degrade. Observe in Fig. 5(c) that performance of LUD drops off when the number of blocks is increased beyond five. We observed that this is due to an increase in L1 data cache contention among warps, which causes an increase in the L1 data cache miss rates. The effect is more prominent on the Kepler configuration, as the L1 cache size is the same and the number of warps executing simultaneously increases. SRAD, which is a C-weak type of workload on Fermi, becomes a MEM workload on the Kepler configuration due to the same reason (Fig. 5(e)).

Our work is motivated by these observations. Detecting the optimal thread block count for C-weak and MEM types of workloads can reduce the amount of core resources required by a kernel, without sacrificing performance. These resources can either be power-gated or utilized for issuing threads from other workloads in concurrent kernel execution scenarios. In the next section, we explain how Perf-Sat utilizes the information regarding core stalls to detect the optimal thread block count.

IV. PERF-SAT

A. Hypothesis

The design of Perf-Sat is based on the behavior presented in the previous section. The key observation is that, performance improves until the increase in the number of pipeline stalls is lesser than the decrease in the number of scoreboard related stalls. Lets consider an application that is executing with N active blocks. Using N+1 active blocks would be better if,

$$Pipeline_{N+1} - Pipeline_N < Scoreboard_N - Scoreboard_{N+1}$$

Thus, N+1 active blocks are a better choice, if the sum of the pipeline and scoreboarding stalls is lower than if N blocks were active. This sum is referred to as stalled cycle count in the rest of the paper. Our thread block scheduler issues $\lceil N_{max}/2 \rceil$ number of blocks on each core at initialization. The number of active blocks is then adjusted until the stalled cycle count is greater than at the previous thread block count.

B. Detection Algorithm

The detection algorithm has three phases:

1) Sample rate detection: As we make decisions based on samples of the core activity during program execution, our algorithm is sensitive to the sampling rate. For example, a sample of the core activity when the workload was very compute intensive should not be compared against one that had many memory access instructions. We observed that applications usually exhibit such phase behavior within warps. The effect of phase behavior gets averaged across thread blocks. Thus, we set the sampling period to approximately the number of cycles it would take for N_{max} thread blocks to complete. When work from a new kernel is started on a core, the number of cycles that elapsed until the first thread block completes (One_TB_{cycles}) are recorded. The sampling period is then set as $One\_TB_{cycles} * N_{max}$.

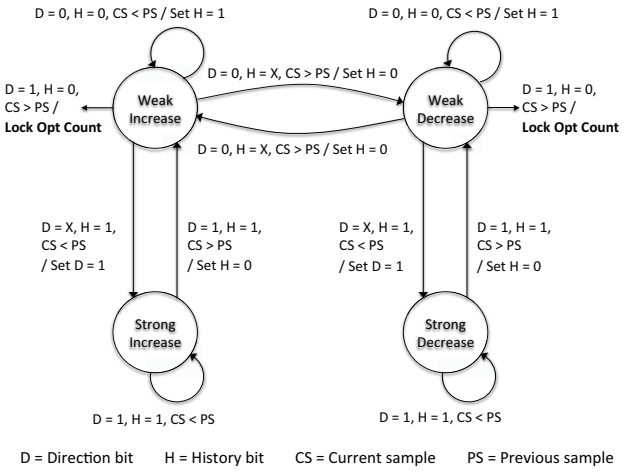


Fig. 6. State machine used by Perf-Sat

Decisions in the next two phases are made according to the state machine shown in Fig. 6. The state machine has four states: weak-increase, strong-increase, weak-decrease and strong-decrease. The stalled cycle count for the previous block count is stored (previous sample). At the end of a sample period, Perf-Sat compares the stalled cycle count for the current thread block count (current sample) with the previous sample. A direction bit is used to indicate whether a particular direction of throttle has been decided. A history bit is used to distinguish between the weak and strong states.

2) Direction of throttle: The optimal thread block count can be either higher or lower than the initial value of $\lceil N_{max}/2 \rceil$. Hence, the direction in which we should change the number of active blocks is decided in the second phase. At initialization, the direction bit is set to 0. At the end of the first sample period, the stalled cycle count for $\lceil N_{max}/2 \rceil$ is stored as the previous sample and the number of blocks is increased to $\lceil N_{max}/2 \rceil + 1$. Perf-Sat is in the weak-increase state.

At the end of the second sample period, if the stalled cycle count is lesser than the previous sample, it is considered that the decision to increase was correct. Perf-Sat remains in the weak-increase state and the history bit is set to 1. The thread block count remains as $\lceil N_{max}/2 \rceil + 1$. On the other hand, if the stalled cycle count is greater than the first sample, Perf-Sat goes into the weak-decrease state. The sample corresponding to $\lceil N_{max}/2 + 1 \rceil$ is stored and number of active blocks is reduced to $\lceil N_{max}/2 \rceil$.

At the next sample, if the number of stalled cycles is coherent with the decision taken, Perf-Stat goes into the strong-increase (or strong-decrease) state and the direction bit is set. If not, the history bit is reset again. Thus, the history bit ensures that a direction of throttle is decided only if two consecutive decisions favor the same direction. If the optimal thread count value is close to $\lceil N_{max}/2 \rceil$, Perf-Sat can toggle several times between the weak-increase and weak-decrease states. The optimal count is set as $\lceil N_{max}/2 + 1 \rceil$, if the state machine toggles more than three times.

TABLE I
DETAILS OF THE GPU CONFIGURATIONS.

Resource	M2090(Fermi)	K20X (Kepler)
Register file capacity per SM	128 KB	256 KB
Maximum threads supported per SM	1536	2048
Maximum thread blocks supported per SM	8	16
SP floating point units per SM (total)	32(512)	192(2688)
DP floating point units per SM (total)	16(256)	64(896)
SP floating point performance (peak)	1330 GFLOPS	3935 GFLOPS
DP floating point performance (peak)	665 GFLOPS	1311 GFLOPS
DRAM clock frequency (MHz)	1850	2600
DRAM peak bandwidth (GB/s)	177	250

TABLE II
DETAILS OF CUDA KERNELS USED FOR OUR WORK.

Kernel	Resource per block		M2090(Fermi)			K20X (Kepler)		
	Reg (Bytes)	Threads	Max	Opt	Type	Max	Opt	Type
Backprop - 1 (BP-1)	12228	256	6	6	C-strong	8	8	C-strong
Backprop - 2 (BP-2)	24576	256	5	5	C-weak	8	5	MEM
B+Tree - 1 (B+-1)	24576	256	5	5	C-strong	8	8	C-strong
B+Tree - 2 (B+-2)	20480	256	6	6	C-strong	8	8	C-strong
Comp. Fluid Dyn. (CFD)	39936	192	3	3	MEM	6	5	MEM
Gaussian (Gaus)	1536	16	8	6	MEM	16	6	MEM
LU Decomposition (LUD)	16384	256	6	5	MEM	8	5	MEM
Hotspot (Hot)	36864	256	3	3	C-strong	6	6	C-strong
Pathfinder (Path)	16384	256	6	5	C-weak	8	5	C-weak
Nearest neighbor (NN)	16384	256	6	5	C-weak	8	7	C-weak
SRAD - 1	12288	256	6	6	C-strong	8	6	MEM
SRAD - 2	12288	256	6	5	C-weak	8	5	MEM
SRAD - 3	16384	256	6	6	C-strong	8	3	MEM
SRAD - 4	20480	256	6	6	C-strong	8	7	C-weak
SRAD - 5	20480	256	6	5	C-weak	8	4	MEM
SRAD - 6	12288	256	6	6	C-strong	8	7	MEM

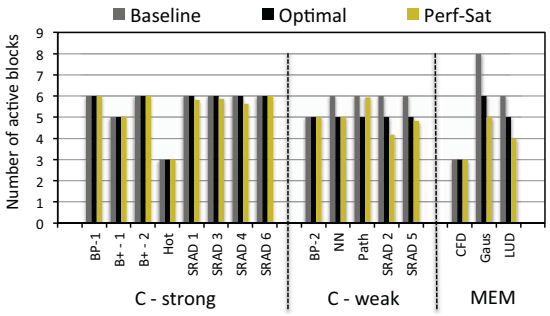
3) Detection of optimal value: Once the state machine is in the strong-increase (or strong-decrease) state, Perf-Sat keeps increasing (or decreasing) the number of active blocks at the end of each sample period. The stored sample is now updated at each sample point. This continues until the stalled cycle count of the current sample is more than the previous sample. For example, lets say an application has $N_{max} = 15$ and $N_{optimal} = 12$. Perf-Sat starts with N = 8. It takes two samples at N = 9 to set the direction bit and goes into the strong-increase state. After N = 9, the number of active blocks is increased at every sample point until 12.

The number of stalled cycles at N = 13 would be more than at N = 12. At this point; the N_{13} sample is discarded, Perf-Sat goes into the weak-increase state and number of active blocks are kept as 13. The history bit is set to 0. At the next sample, if the stalled cycle count is again more than the N_{12} sample, the optimal block count is detected as 12. Similar to the problem mentioned in the previous section, just before converging at $N_{optimal} = 12$, Perf-Sat can toggle several times between the weak-increase and strong-increase states. The optimal block count is decided as the previous value (12 in this example) if it toggles more than 3 times.

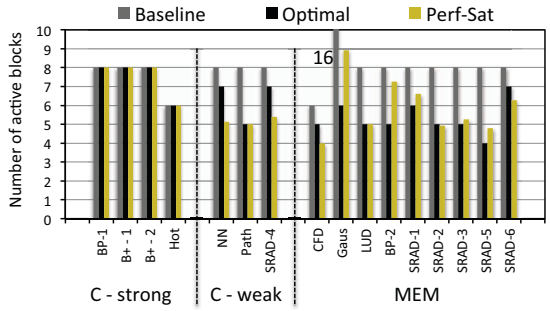
V. EXPERIMENTAL RESULTS

A. Experimental Setup

Our experiments were performed on GPGPU-Sim v3.1 [5], a cycle accurate GPU microarchitecture simulator. As mentioned in section I, we configured the simulator to closely match the NVIDIA Tesla M2090, which is based on the Fermi architecture, and the Tesla K20X, which is based on the Kepler architecture. Details of both the configurations are provided in Table I.



(a) Tesla M2090



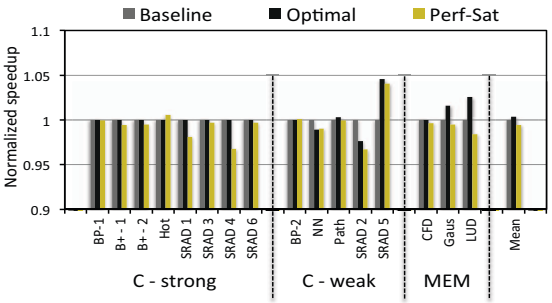
(b) Tesla K20X

Fig. 7. Comparison of active thread blocks detected by Perf-Sat, with baseline and empirically found optimal count.

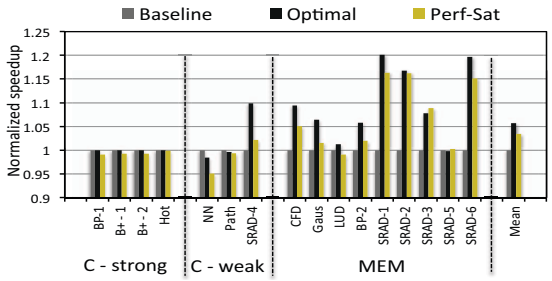
We simulated a wide range of CUDA kernels from the Rodinia benchmark suite [13]. We show results for 16 kernels representing a good mix of the three types of applications: C-strong, C-weak and MEM. Table II provides information about the kernels. Kernels that are numbered, are from applications which have multiple kernels.

B. Detection Accuracy

To find the optimal block count, the kernels were executed separately for each thread block count by modifying the thread block limit per core in the simulator. The count after which increasing the number of blocks resulted in less than 2% performance improvement was decided as optimal. Fig. 7 compares this empirically determined optimal count with the count detected by Perf-Sat. The baseline thread block scheduler issues the maximum number of blocks supported. As our mechanism operates separately for each core, the number reported in the figure is the average across all cores. Perf-Sat finds the optimal count with 94.25% accuracy on the Fermi configuration and with 85.12% on the Kepler configuration. There are two reasons for the difference between the detected and the empirically found optimal values. Kernels for which the optimal count is close to the maximum, Perf-Sat detects the optimal count as maximum on some cores and lower than the empirically found optimal on other cores. Secondly, for the cores where the optimal count is close to $N_{max}/2$, Perf-Sat conservatively chooses the higher block count value.

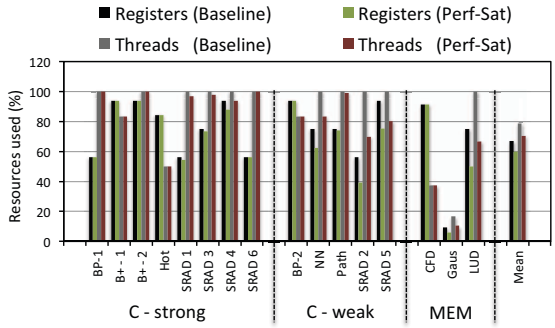


(a) Tesla M2090

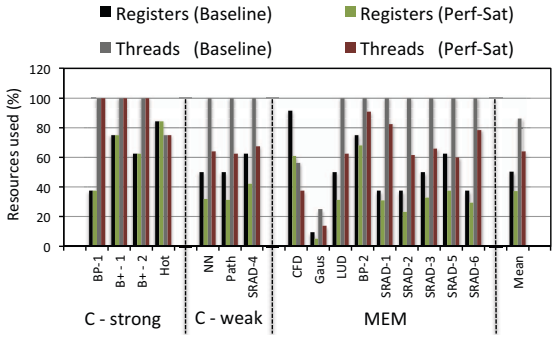


(b) Tesla K20X

Fig. 8. Performance comparison between baseline, optimal and Perf-Sat.



(a) Tesla M2090



(b) Tesla K20X

Fig. 9. Comparison of resource utilization per core.

C. Performance Impact

Fig. 8 compares the performance of Perf-Sat with baseline. The difference between Perf-Sat and the baseline scheduling policy is that Perf-Sat throttles the number of active blocks in the beginning to detect the optimal count, while the baseline scheduler always issues the maximum number of blocks possible. The speedup reported under Optimal is using the execution time when the kernel is executed with the empirically found number of thread blocks. As expected, we incur some loss in performance for the C-strong and C-weak category of kernels. This is because, their optimal block count is close to the maximum. Perf-Sat starts with $N_{max}/2$ number of blocks and takes a few iterations to reach the optimal count. However, the performance loss is only 0.51% on the Fermi configuration and 0.88% on the Kepler configuration. The performance of the MEM category of workloads increases because the optimum count is much less than the maximum. There is a 4.95% performance improvement on average for these workloads across the two configurations.

D. Resource Utilization

Fig. 9 shows the percentage of resources utilized by each kernel with the baseline and Perf-Sat scheduling policies. As the maximum block count for all of our kernels was constrained due to either thread slots or registers, results are shown for only these two resources. For the C-strong category kernels, Perf-Sat utilizes only 0.97% lesser resources on average compared to the baseline scheduler. This is expected as the optimal count value is the same as maximum for these workloads. The resource savings are more significant for the other two categories of workloads. Perf-Sat utilizes 14.32% and 35.35% less resources compared to the baseline for the C-weak workloads on the M2090 and K20X configurations respectively. For the MEM category, the reduction in resource usage is 25.31% on the M2090 and 31.48% on the K20X configuration. Across all kernels, the savings are 10.45% on the M2090 and 25.51% on the K20X. Average resource savings across the configurations is 18.32%.

VI. Related Work

This section gives on overview of related work in GPGPU scheduling techniques and workload throttling.

A. Scheduling in GPGPU

There have been several prior works that have proposed scheduling techniques at the warp level. To increase the overlap of memory latency with computation, Narasiman et al. [10] proposed the two-level warp scheduling scheme. They group a fixed number of warps into fetch groups. Warps are scheduled in round-robin order from one fetch unit until all the warps stall on a memory request, and then the next fetch group is brought into the scheduler. Fetch groups are selected in round robin order as well. Gebhart et al. [11] proposed a similar hierarchical scheduling policy, along with a register file cache. The objective of their work was to improve the energy efficiency of the warp scheduler and instruction dispatch. Rogers et al. [6] suggested that using a greedy then oldest (GTO) policy for selecting fetch groups as well as for

selecting warps within a fetch group, performs better than round robin. Jog et al. [12] used the two level scheduler to improve the efficiency of prefetching for GPUs. They group non-consecutive warps into one fetch group, which then prefetch for the next group. As warps for which data is being prefetched are scheduled in the next group, this reduces the amount of time they are blocked due to memory access.

B. Workload Throttling

It has been suggested in a few prior works that increasing number of threads beyond a point can degrade performance for some workloads. Bakhoda et al. [5] varied the amount of core resources from 25% to 200% and observed that two of the workloads they used showed a decrease in performance. In their work on cache conscious warp scheduling [6], Rogers et al. showed that high thread count can degrade performance for cache sensitive workloads. They track L1 data cache lines to detect warps that have lost intra-warp data locality, possibly because of other warps evicting data lines that would have been used by them. A scoring system increases the priority of such warps. Warps that have a score below a certain threshold are not scheduled, thereby reducing the number of active threads. For cache-sensitive workloads, their warp scheduler effectively reduces the active thread count close to the optimal count. We attempt to find this count at the block scheduler.

Our work is closest to [7] and [8]. Kayiran et al. [7], monitor the core idle cycles (C_{idle}) and cycles when all threads are waiting for memory access (C_{mem}). They start by issuing $\lceil N_{max} \rceil$ thread blocks. They increase the number of blocks if C_{idle} is higher than a threshold. Once, C_{idle} is less than the threshold, they further increase (or decrease) the block count if C_{mem} is lower (or higher) than another threshold. Values for both the thresholds, as well as the sampling period are found empirically.

The work presented by Lee et al. in [8] is similar to ours, although using a different methodology. They launch N_{max} blocks at initialization and use a greedy warp scheduling policy to issue work. The number of instructions executed until the first thread block completes are counted. The optimal thread block count is estimated as $\lfloor N\_max * (instr.\ in\ 1\ block)/instr.\ executed \rfloor$. The idea is the amount of computation done to overlap latency of the first block should be enough as now a new block can be issued in its slot. Thus, the number of thread blocks that encompassed those computations (calculated by the earlier equation) should be optimal.

VII. Conclusion

In this work, we showed that launching maximum number of thread blocks is not always optimal for GPGPU applications. Three categories of applications were identified and the effect of number of thread blocks on their behavior was thoroughly evaluated. We presented Perf-Sat, a hardware mechanism that detects the optimal thread count on each core at runtime. Its performance was evaluated against the baseline and the empirically found optimal thread count values.

With less than 1% loss in performance, Perf-Sat can result in significant resource savings for two out of the three workload categories. In our future work, we would use Perf-Sat in an integrated thread block and warp scheduler. The thread block scheduler would use Perf-Sat to detect cores which have unused resources. It would then issue thread blocks from a concurrently active kernel. The warp scheduler on those cores, would then schedule threads from the second kernel at a lower priority. Priority based access to hardware units would be required to ensure that performance of first kernel is not degraded due to the contention from threads of the second kernel.

Acknowledgments

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

References

[1] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[2] P. Glaskowsky, "Next Generation CUDA Compute Architecture: Fermi," 2010.

[3] NVIDIA, "CUDA C Programming Guide," 2013.

[4] A. Munsi, "The OpenCL C 2.0 Specification," 2013.

[5] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. of the IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2009, pp. 163–174.

[6] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proc. of the 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*. IEEE Computer Society, 2012, pp. 72–83.

[7] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proc. of the 22nd Int. Conf. on Parallel Architectures and Compilation Techniques*. IEEE Press, 2013, pp. 157–166.

[8] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization and Performance Through Alternative Thread Block Scheduling," in *Proceedings of the Int. Symp. on High Performance Computer Architecture*, 2014.

[9] M. Awatramani, J. Zambreno, and D. T. Rover, "Increasing GPU Throughput using Kernel Interleaved Thread Block Scheduling." in *Proc. of the 31st Int. Conf. on Computer Design*. IEEE, 2013, pp. 503–506.

[10] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *Proc. of the 44th Annual IEEE/ACM Int. Symp. on Microarchitecture*. ACM, 2011, pp. 308–317.

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A Hierarchical Thread Scheduler and Register File for Energy-Efficient Throughput Processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 2, p. 8, 2012.

[12] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proc. of the Int. Symp. on Computer Architecture*. ACM, 2013.

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