Increasing GPU Throughput using Kernel Interleaved Thread Block Scheduling

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*—The number of active threads required to achieve peak application throughput on graphics processing units (GPUs) depends largely on the ratio of time spent on computation to the time spent accessing data from memory. While compute-intensive applications can achieve peak throughput with a low number of threads, memory-intensive applications might not achieve good throughput even at the maximum supported thread count. In this paper, we study the effects of scheduling work from multiple applications on the same GPU core. We claim that interleaving workload from different applications on a GPU core can improve the utilization of computational units and reduce the load on memory subsystem. Experiments on 17 application pairs from the Rodinia benchmark suite show that overall throughput increases by 7% on average.**

I. Introduction

In recent years, graphics processing units (GPUs) have been widely adapted in high throughput computing clusters. Moreover, due to a rise in the number of data parallel consumer applications, there is a growing interest in using GPU acceleration in the desktop and mobile domains as well. Due to this trend, future GPU architectures are likely to support concurrent execution of multiple compute applications.

As GPU architectures are designed with a focus on high throughput computing, they use non-speculative in-order processor pipelines as a tradeoff for a large number of computational units. Consequently, to reduce the overhead of long latency memory operations, the GPU hardware scheduler switches between threads and tries to maximize the overlap of memory access and computation. A good balance is required between the number of threads launched on a core and the average number of computations performed per memory access (FLOP/byte ratio) within each thread. If a kernel has low FLOP/byte ratio, the amount of computation in ready threads may be insufficient to hide the memory access latency of the stalled threads leading to underutilization of the core's computational resources. On the other hand, a kernel with high FLOP/byte ratio achieves peak throughput at a lower thread count, and has threads that remain idle waiting for computational resources.

In current GPU architectures, concurrently launched kernels are executed on separate cores and suffer from the above mentioned imbalance. In this paper, we investigate whether scheduling threads from multiple kernels on the same GPU core can mitigate this problem. Our experimental results (obtained through the simulation of application pairs from the Rodinia benchmark suite [1]) indicate that compute-intensive kernels can benefit from computational resources that were previously under-utilized by the memory-intensive kernels. Moreover, memory-intensive kernels benefit from reduced load on the memory pipeline.

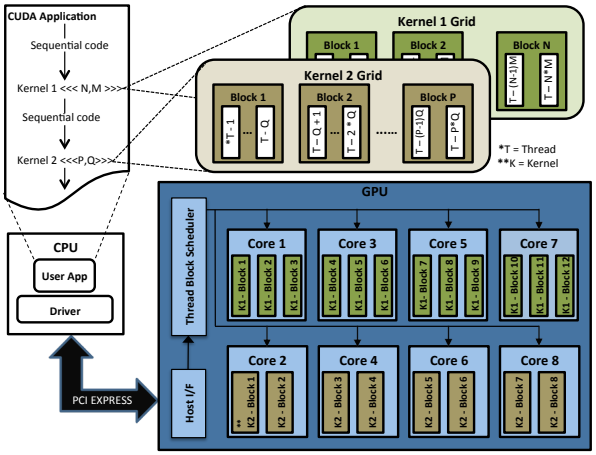

Fig. 1. A CUDA application with two kernels launched concurrently on an eight core GPU. Each set of four cores executes threads from one kernel.

II. Background and Motivation

The two most widely used programming models for general purpose computing on GPUs are CUDA and OpenCL. In this work we use CUDA and the NVIDIA Fermi architecture [2].

A. Programming Model and GPU Architecture

CUDA is an extension of C and has constructs to mark GPU-related functions. The application is compiled into a single binary and GPU-related functions are executed by the driver. A block of instructions to be executed on the GPU is referred to as a *kernel* (Fig. 1). CUDA supports a three level hierarchy to group data. Typically, a single *thread* performs work related to one data element. Threads are grouped into *thread blocks*, which can use barrier synchronization and share data using on-core memory. The *grid*, the highest level in the hierarchy, consists of several thread blocks and represents all data elements of a kernel.

The GPU has several computational cores referred to as streaming multiprocessors (SMs). To reduce the overhead of barrier synchronization work is submitted to SMs at the granularity of thread blocks. Threads within a block are executed in

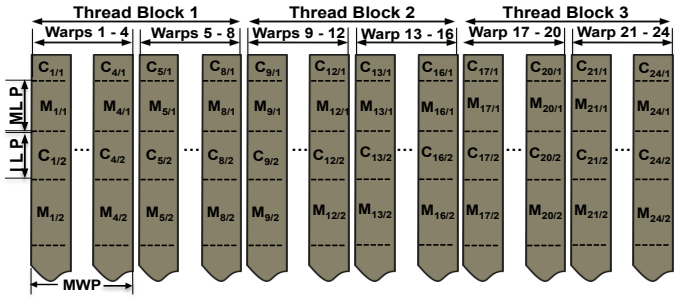

Fig. 2. Depiction of 3 thread blocks of size 8. $C_{i/j}$ and $M_{i/j}$ represent the j^{th} computation and the j^{th} memory access blocks of the i^{th} warp respectively.

fixed size groups referred to as *warps* in SIMD fashion. Once a thread block is launched, the register and on-chip memory resources are reserved until all the warps complete execution. Thus, the number of thread blocks executing simultaneously on an SM depends on the number of warps in a thread block and resource requirements of a warp.

B. Concurrent Kernel Execution

The CUDA driver assigns a separate execution context to each application. A GPU-related function is referred to as a *command*. Commands from only one context are executed at a time on the GPU [3]. CUDA *streams* can be used to launch multiple commands concurrently from within a context. Current GPUs support execution of multiple concurrent compute kernels that are launched using separate streams. When a new kernel is launched, the thread block scheduler recalculates the kernel-to-core mappings. In current GPUs, concurrent kernels are mapped to separate SMs. Figure 1 depicts two kernels launched concurrently on an eight core GPU. Each kernel is mapped to a set of four SMs. We refer to this as **spatial scheduling**. In our approach each kernel is mapped to all the SMs, and their warps are interleaved on the cores. We refer to this as **interleaved scheduling**.

C. Motivation

As previously described, GPU architectures launch a large number of warps to reduce the overhead of high latency memory operations. Figure 2 depicts three thread blocks launched on an SM. Each vertical bar represents a warp. $C_{i/j}$ and $M_{i/j}$ depict regions of compute and memory instructions. *Arithmetic density* (Table I) is the ratio of compute to memory instructions of a kernel. A group of warps that completes a memory request at the same time is referred to as a memory warp parallel (MWP) set. *Ideal warp count* is the ratio of sum of the average latencies to arithmetic density. It is a simple approximation of the number of active warps required to hide the memory latency of one MWP set and fill the arithmetic pipeline.

The applications listed as **memory** in Table I have a very high ideal warp count. These application become bandwidth limited when all the ready warps complete instructions from the j^{th} compute block before warps waiting on the j^{th} memory block become ready. The functional units remain idle when all the warps are waiting for data from memory. On the other hand, **compute** applications have a low ideal warp count. Consequently, the ready warps will still be executing instruction from the j^{th} compute block when the stalled warps receive their data, and become ready to execute instructions of the $(j+1)^{th}$ block. If the number of active warps is more than the arithmetic pipeline depth, the excess warps remain idle until compute resources become available.

If we launch these two kinds of applications on the same SM, the warps of the compute-intensive application will benefit from the otherwise under-utilized computational resources. Additionally, warps of the memory-intensive application will benefit from the reduced load on the memory pipeline.

TABLE I
CHARACTERIZING APPLICATIONS USING ARITHMETIC DENSITY

Application	ALU Inst	MEM Inst	Arithmetic Density	Avg. ALU Latency	Avg. MEM Latency	Ideal warp count	Type
Pathfinder (Path)	3822528	692648	5.52	10.60	25.95	6.63	Compute
Backprop-1 (BP-1)	4177920	675840	6.18	13.75	26.83	6.56	Compute
Hotspot (Hot)	3459423	510461	6.77	12.60	20.59	4.90	Compute
Backprop-2 (BP-2)	1802258	458759	3.93	14.45	52.23	16.97	Memory
Heartwall (HW)	150144	25398	5.91	15.30	58.93	12.55	Memory
Comp. Fluid Dynamics (CFD)	933521	78023	11.95	13.37	249.35	21.96	Memory
LU Decomposition (LUD)	109800	73800	1.49	11.13	53.28	43.30	Memory
Needleman-Wunsch (NW)	117888	33152	3.56	9.43	80.71	25.35	Memory

III. THREAD BLOCK SCHEDULER

The thread block scheduler performs two primary tasks: calculating the kernel-to-core mappings, and scheduling thread blocks from active kernels on the mapped cores. In this section, we describe how the spatial scheduler and the interleaved scheduler perform these tasks.

A. Kernel to Core Mapping

Kernel to core mapping is calculated each time a new kernel is launched or a kernel completes execution. When a new kernel is launched, its information is added in the **kernel status block** (Fig. 3) at a location that has the valid bit reset. If all the entries have their valid bit set to 1, the GPU is executing the maximum number of kernels supported and the kernel launch fails.

The spatial scheduler maps kernels to cores in a round robin order. The index of the mapped kernel is stored in the assigned kernel field of the core in the **core status block** (Fig. 4). Launch of new thread blocks is stalled on cores for which the active kernel is different than the assigned kernel. When threads of the active kernel complete, the assigned kernel is made active and launching of thread blocks is resumed.

The interleaved scheduler maps each kernel to all the cores. The assigned kernel field is replaced by a queue with an entry for each active kernel. When the mapping changes, the maximum thread block limit of each kernel in the queue is adjusted. Interleaved scheduling changes the number of thread blocks active per core. Hence, the register and shared memory requirements of each kernel are stored and per-core occupancy for each kernel is calculated in hardware. Interleaved scheduling is used only if the total number of

Valid	Kernel Index	Grid Dimension	Next Block Dimension	Resources		Num. Active Cores
				Reg	SMEM	
1	9	(128,128,1)	(19,18,1)	4096	128	9
1	10	(256,1,1)	(182,1,1)	2048	256	7
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	0	0	0	0	0	0

Fig. 3. The kernel status block: Kernel 9 was launched first. 7 cores have started executing kernel 10. Core 16 is still executing kernel 9 (Refer Fig. 4).

Full	Core ID	Active Kernel	Assigned Kernel	Maximum Blocks	Num. Active Blocks
1	1	9	9	8	8
1	2	10	10	6	6
⋮	⋮	⋮	⋮	⋮	⋮
0	16	9	10	8	2

Fig. 4. Core status block: Cores 1 and 2 are executing kernels 9 and 10 respectively. 16 is mapped to kernel 10, but is finishing threads of kernel 9.

blocks for each kernel across the GPU is greater than or equal to spatial scheduling. We refer to this as the *occupancy test*.

B. Thread Block Scheduling

The spatial scheduler performs two checks before launching a thread block. The active and assigned kernel fields are compared to verify that mapping has not changed. Secondly, the number of active blocks and maximum blocks of the kernel are compared to verify that core has sufficient resources (Fig. 4). If the two checks pass, the thread block is launched and the active blocks field is incremented.

The interleaved scheduler performs the occupancy test each time a new kernel is launched or a kernel exits. Depending on the outcome, the maximum blocks field of all kernels within each core is updated. Only kernels having fewer active thread blocks than the maximum are scheduled. This technique encompasses both the checks performed in the spatial scheduler. It also handles cases where the mapping changes at runtime from spatial to interleaved or vice versa.

IV. Results

The applications used for our work (Table I) were chosen from the Rodinia benchmark suite [1] and simulated on GPGPUSim version 3.1 [4]. The GPU configuration used has 16 SMs, with each core executing a maximum of 48 warps.

A. Effect of Occupancy on Throughput

Interleaved scheduling reduces the number of active warps per core. As the applications execute on twice the number of cores, the total warps active across the GPU remain the same. To verify that reducing per-core occupancy does not reduce throughput significantly, we simulated the applications with different thread block counts by tuning the core resources. Figure 5 shows the incremental change in speedup obtained

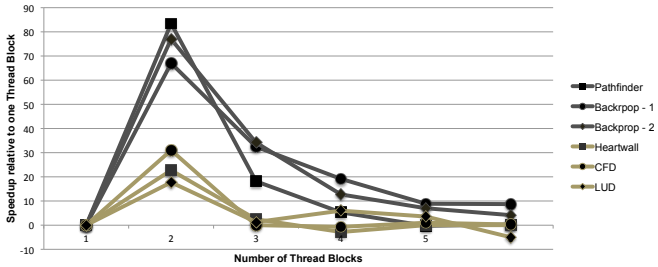

Fig. 5. Relative increase in speedup with increase in number of thread blocks

by increasing number of thread blocks by one at each data point.

All applications benefit when number of blocks is increased from one to two. This is because warps within a thread block have good data locality and do not benefit from inter-warp memory latency tolerance. Increasing the blocks to two significantly increases overlap of memory access and computation. As expected, memory applications show lower speedup (20-30%) as compared to compute applications (65-80%).

Compute intensive applications continue to improve gradually. The speedup saturates when number of warps is high enough to hide the latencies completely. We observe that memory-intensive applications do not improve significantly beyond two thread blocks. Interestingly, they achieve similar performance at half the occupancy as the maximum.

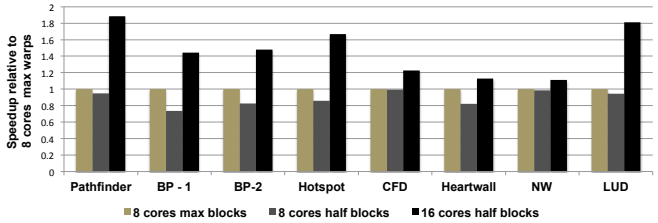

Fig. 6. Relative increase in speedup with increase in computation resources

B. Effect of Computational Resources

Figure 5 shows that compute intensive applications continue to improve gradually with increase in occupancy. It is important to verify that the overhead of reduced throughput per core is not greater than the improvement that would be achieved from additional computational resources. We observe in Fig. 6 that compute intensive applications suffer by 15 to 30% when executed at half the number of blocks. However, when total blocks across the GPU are kept constant by executing on 16 cores, they achieve a significant speedup. Memory intensive applications do not suffer significantly from reduced occupancy and show little benefit when executed on 16 cores.

The behavior of LUD is particularly interesting. We suspect that it has long regions of memory and compute instructions. If a highly compute-intensive region follows a memory-intensive region, having more than two thread active thread blocks does not improve performance significantly. However, when executed on twice the number of cores the compute regions execute in parallel, leading to significant increase in speedup.

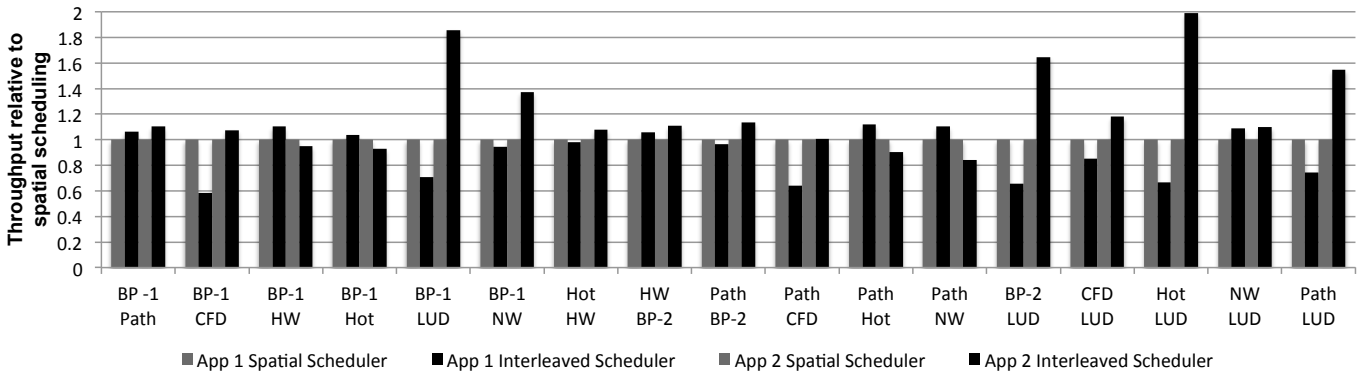

Fig. 7. Percentage change in throughput for each kernel in application pair

C. Effect of Thread Block Interleaving on Throughput

Figure 7 compares the throughput of each application executed with interleaved and spatial thread block scheduling schemes. Three trends can be observed. One, overall throughput improves when compute intensive applications like Pathfinder and Backprop are interleaved with memory intensive applications like Heartwall. Second, when two applications of the same type are interleaved, one with higher arithmetic density shows improvement. For example Pathfinder and Backprop show improvement when executed with Hotspot.

A third trend is that applications with high L1 data cache miss rates significantly reduce throughput of the other application. In our experiments, CFD and LUD significantly reduced throughput of all the applications executed with them. Their cache miss rates are 73% and 78% respectively. The higher improvements for LUD as compared to CFD are due to its compute intensive regions benefiting from the additional computational resources (Section IV.B). The overall throughput increased by 7% on average for 13 application pairs. The average increase in throughput across all 17 application pairs was 2.36%.

V. Related Work

The benefits of concurrent kernel execution were first demonstrated by Guevara et al. [5]. Their objective was to merge kernels that do not have enough data parallelism to utilize the GPU completely. Chen et al. introduced "persistent" kernel that executes throughout the runtime of the application [6]. New compute work is pushed via GPU memory and executed using thread blocks of the persistent kernel. The motivation behind these works was to enable concurrent kernel execution, a feature currently supported by the hardware.

Another body of work focuses on efficient sharing of GPU by multiple CPU threads. Wang et al. manually manage the synchronization between OpenMP threads that launch work on the GPU [7]. This capability was implemented in CUDA runtime v4.0. Wende et al. developed a software layer, which intercepts kernel launch calls from multiple host threads and reorders the kernels to maximize concurrency [8].

Our work is closest to the work of Gregg et al. in [9]. Their goal was also to improve throughput by concurrently executing compute and memory bound applications. They use the persistent kernel approach [6] and launch thread blocks from separate applications into the persistent kernel. However, they do not discuss how the GPU hardware schedules these blocks on the cores.

VI. Conclusion

In this work we study the effects of scheduling threads from multiple applications on the same GPU core. We show that interleaving threads of certain application types can improve the overall throughput. Contention in L1 data cache can reduce throughput for certain application pairs. Runtime detection of application characteristics to aid scheduling decisions can help in such scenarios and is one of the next steps for this project.

Acknowledgments

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

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. on Workload Characterization*, October 2009.

[2] Nvidia, "Next Generation CUDA Compute Architecture: Fermi," 2010. [Online]. Available: www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_architecture_Whitepaper.pdf

[3] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU Resource Management in the Operating System," in *USENIX ATC*, vol. 12, June 2012.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[5] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling Task Parallelism in the CUDA Scheduler," in *Workshop on Programming Models for Emerging Architectures*, September 2009.

[6] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," in *2010 IEEE Int. Symp. on Parallel Distributed Processing (IPDPS)*, April 2010.

[7] L. Wang, M. Huang, and T. El-Ghazawi, "Exploiting concurrent kernel execution on graphic processing units," in *2011 Int. Conf. on High Performance Computing and Simulation (HPCS)*. IEEE, July 2011.

[8] F. Wende, F. Cordes, and T. Steinke, "On Improving the Performance of Multi-threaded CUDA Applications with Concurrent Kernel Execution by Kernel Reordering," in *IEEE 2012 Symp. on Application Accelerators in High Performance Computing (SAAHPC)*, July 2012.

[9] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained Resource Sharing for Concurrent GPGPU Kernels," in *Proc. of the 4th USENIX Conf. on Hot Topics in Parallelism*, October 2012.