Simultaneous Multikernel: Fine-Grained Sharing of GPUs

Zhenning Wang, Jun Yang, Rami Melhem,
Bruce Childers, Youtao Zhang, and Minyi Guo

Abstract—Studies show that non-graphics programs can be less optimized for the GPU hardware, leading to significant resource under-utilization. Sharing the GPU among multiple programs can effectively improve utilization, which is particularly attractive to systems (e.g., cloud computing) where many applications require access to the GPU. However, current GPUs lack proper architecture features to support sharing. Initial attempts are very preliminary in that they either provide only static sharing, which requires recompilation or code transformation, or they do not effectively improve GPU resource utilization. We propose Simultaneous Multikernel (SMK), a fine-grained dynamic sharing mechanism, that fully utilizes resources within a streaming multiprocessor by exploiting heterogeneity of different kernels. We extend the GPU hardware to support SMK, and propose several resource allocation strategies to improve system throughput while maintaining fairness. Our evaluation of 45 shared workloads shows that SMK improves GPU throughput by 34 percent over non-shared execution and 10 percent over a state-of-the-art design.

Index Terms—Context switch, GPU, multitasking

1 Introduction

Due to their superb computing power, GPUs have been widely adopted in data centers, cloud computing, and mobile/embedded computers. As GPUs become more general-purpose, there is an increasing number of applications that demand computing power from a GPU. However, non-graphics applications may be less optimized for GPUs, resulting in on-chip resource under-utilization. One kernel.[1] may use little scratchpad memory yet access the L1 cache heavily, while another kernel may have the opposite usage. This behavior arises because users may not be experienced enough to write high-quality code to fully utilize available resources. Furthermore, the GPU code may be written assuming a lower-end architecture but executed on a higher-end version of the architecture that has more resources. Also, data centers or clouds are typically exposed as services to users. Consolidation and virtualization are used to share hardware resources among multiple applications for cost-effectiveness and energy efficiency. Even the latest GPUs have minimal to no support for sharing among multiple applications or users.

There are software attempts to enable sharing a GPU [1], [2]. These approaches mainly rely on users or code transformation to statically define or fuse parallelizable kernels. Hence, these techniques cannot provide sharing among dynamically arriving GPU jobs. Also, once launched, kernels cannot be preempted and resumed later on. Kernel preemption was recently proposed with architectural extensions [3], [4]. The main mechanism is to swap the context of a kernel on one SM[2] with the context of a new kernel. These proposals allow different application kernels to execute concurrently on disjoint set of SMs, achieving spatially partitioned multitasking of a GPU [4], [5]. Unfortunately, resource under-utilization occurs mostly within an SM. Hence, context switching in unit of SMs cannot sufficiently improve GPU resource utilization.

This paper introduces a new notion of a multi-tasking GPU that improves resource utilization to boost overall system throughput. We propose Simultaneous Multikernel (SMK), drawing an analogy from simultaneous multithreading (SMT) for CPUs, to increase thread-level parallelism (TLP) of a GPU. SMK exploits kernel heterogeneity to allow concurrent execution of multiple kernels *within each SM*. SMK is enabled by a fine-grained context switch mechanism on per TB basis for low preemption overhead. Moreover, a TB dispatch strategy is proposed to maintain resource fairness among shared kernels. Our evaluation on 45 pairs of Parboil kernels shows improvement in system throughput with SMK of 10 percent on average, and up to 70.7 percent over the state-of-the-art spatial partitioning design [4].

2 Motivation—Heterogeneity of Kernels

We observed that the resource usage in each SM, such as registers and scratchpad memory, may be imbalanced within a kernel and can change significantly from kernel to kernel. Prior work reported similar findings [1]. We examined all benchmarks from Parboil and Rodinia suites and observed low resource utilization across nearly all of them, assuming a recent GPU generation Nvidia GTX980. For example, *StreamCollide* fully uses registers (92.3 percent) but leaves shared memory completely unused (0 percent). *Block2Dregtiling* is limited by hardware constraints on the number of active threads (100 percent), leaving registers underutilized (75 percent). In addition, even though GPUs are heavily multi-threaded, there are still abundant core idle cycles because of limited memory bandwidth, indicating that dynamic core computing cycles are underutilized. We observed 52.5-98.1% core idle time for 10 Parboil [6] benchmarks. If kernels with compensating resource usage and runtime behavior can be assigned to the same SM, then the resource utilization (both static and dynamic) in that SM and overall throughput can be improved. For example, a memory intensive kernel and a compute intensive kernel can co-run on the same SM: the latter kernel may may have enough compute cycles to hide the memory stall cycles from the former kernel.

3 Prior Art

The most preliminary support to kernel concurrency relies on programmer to define parallelizable *streams* of kernels, each stream being a sequence of kernels with dependencies. Hyper-Q [7] improves on this preliminary support with hardware queues for streams, allowing kernels to be launched from different queues and co-run on the same GPU. However, user-defined static parallelism is suitable for a single application with multiple kernels. Additionally, these approaches cannot enable sharing among dynamically arriving GPU jobs, as once launched, kernels cannot be preempted and resumed later.

More sophisticated software-based schemes work around hardware limitations by merging two kernels into one with compiler techniques [1], [2], [8]. The execution path is controlled by conditional statements [9], [10]. Although parallelizing different applications is possible, such static approaches still cannot enable dynamic sharing, as illustrated in Fig. 1a. Further, since kernels from different applications are fused into one, the hardware sees only one kernel, and hence, it may produce unfair scheduling among different kernel threads. Also, the kernel source code must be available prior to execution (for compilation) which often is difficult.

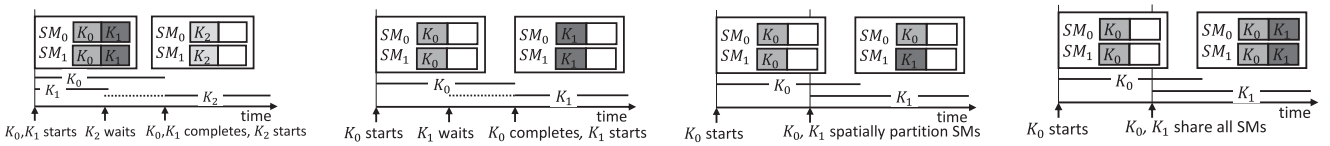
There have been many efforts for the OS or hypervisor of a virtual machine to enhance multitasking in the GPU [11], [12].

1. A GPU application consists of multiple kernels, each capable of spawning many threads that are grouped into thread blocks (TB).

2. A GPU typically consists of many streaming multiprocessors (SM), each containing many simple execution cores.

- *Z. Wang and M. Guo are with the Department of Computer Science, Shanghai Jiao Tong University, P.R. China. E-mail: znwang@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn.*
- *J. Yang is with Electrical and Computer Engineering Department, University of Pittsburgh, PA. E-mail: juy9@pitt.edu.*
- *R. Melhem, B. Childers and Y. Zhang are with the Department of Computer Science, University of Pittsburgh, PA. E-mail: {melhem, childers, zhangyt}@cs.pitt.edu.*

Manuscript received 3 Aug. 2015; revised 19 Aug. 2015; accepted 3 Sept. 2015. Date of publication 8 Sept. 2015; date of current version 5 Jan. 2017.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2015.2477405



(a) Current software's solution (b) Current OS's solution (c) Current architecture solution with preemption and spatial partitioning (d) SMK solution with preemption and per SM sharing

Fig. 1. Evolution of multitasking GPU.

A typical approach is to intercept kernel launching requests and change to another kernel as demanded. The main limitation remains—an already launched kernel cannot be preempted. Therefore, a new kernel needs to wait for the completion of the executing kernel, as illustrated in Fig. 1b. No sharing is supported and resource utilization is not improved either.

Recently, architectural extensions have been proposed to support sharing the GPU through kernel preemption. A context switch is achieved by hot-swapping kernel contexts in one SM with a new kernel via the main memory, or draining all running TBs on one SM and then loading in a new context [4]. Chimera [3] reduces swap overhead by dropping the running TBs of one SM if the kernel is idempotent [13]. These techniques allow multiple kernels to share the GPU via spatially partitioning (Spart) the SMs, as illustrated in Fig. 1c. However, each SM in Spart still executes one kernel at a time while resource under-utilization mainly occurs *within* an SM. Our proposed SMK on the contrary enables multiple kernels to share each SM (Fig. 1d), and hence, it can achieve better resource utilization and higher GPU throughput.

4 Simultaneous Multikernel Design

Consider a general scenario where the GPU is currently executing a kernel K which has exhausted at least one type of GPU resource. To allow a pending kernel, $newK$, to co-execute with K on each SM, preemption must be supported to swap *a portion* of K's context on each SM with a portion of the new context of $newK$ so that the aggregated resources of both can fit in the SM. This is in contrast to Spart where *all* context of K on a SM is swapped out so that the SM will be hosting only $newK$. Hence, the first challenge is the design of such *partial context switching* (PCS). This is immediately followed by the question of how much of K's context should be swapped with $newK$, or, how to allocate resources of an SM between K and $newK$. This decision is critical to achieving high overall GPU throughput, while being fair to co-running kernels. To this end, we discuss the proposed design of partial context switching and resource allocation with fairness in this section.

4.1 Partial Context Switching

We first describe the base GPU execution engine [4], and then discuss the essential components relevant to our design. In the base GPU, a centralized SM driver is in charge of receiving commands, such as launching kernels and memory operations, from the CPU; initializing the SM for kernel execution; and dispatching TBs of a kernel onto different SMs. In each cycle, the SM driver searches an Active Queue containing a sequence of TBs belonging to one kernel to find a candidate TB for dispatch.

When a new kernel arrives, some TBs of currently executing kernels are preempted, such that the new kernel can co-execute with the existing ones. With SMK, only a portion of TBs on a SM are preempted, depending on how much resource is needed to dispatch TBs of the new kernel. We name our preemption mechanism, *Partial Context Switching*. The main advantage of PCS is that preemption takes place without blocking the SM. The partially preempted kernel still makes forward progress during swapping. The new kernel starts execution much sooner than in Spart as the amount of swapping is much smaller. In contrast, Spart stops the execution of the preempted kernel on the chosen SM, and spends long time on saving large amount of context. We now describe the steps of PCS.

As shown in Fig. 2, we add a Preemption Engine (PreEng) per SM to perform PCS. When the SM driver decides to swap TB_x from SM_i, it sends this information to PreEng in SM_i (①). Next, PreEng stops fetching new instructions for TB_x, and drains all currently executing instructions from its pipeline, including pending memory requests (②). Once draining is complete, PreEng initiates PCS by sending memory store requests directly to the memory controller for the context of TB_x, namely for register contents, shared memory values, barrier information and SIMT stack (③). Once PCS is complete, PreEng sends the memory address, a pointer, of TB_x's context to the Preempted TB Queue (④), which maintains the TBs that the SM driver may select and dispatch in the near future. Swapping in a TB's context is symmetric to swapping out context (swapping in a TB_y to SM_j in the figure). The SM driver first reads memory locations of TB_y's context from the Preempted TB Queue and sends it to PreEng in SM_j (i). PreEng then issues memory load requests to the memory controller with the pointer to TB_y's context (ii). Once the context is fully loaded, TB_y starts executing on SM_j (iii).

As context switching tends to generate large memory traffic which could degrade the performance of non-preempted TBs, PreEng also adopts throttling to limit the number of requests sent to the memory controller. The limit is set to be no more than the number of warps in preempted TBs. Hence, in the worst case, PreEng generates no more memory traffic than a memory-intensive kernel.

The hardware overhead of PCS is mostly in logic, and is similar to [4], with low complexity. In summary, the goal of PCS is to use the least amount of context switching overhead to maximize resource utilization of each SM, while ensuring preempted kernels make forward progress.

4.2 Resource Usage

With SMK, the SM driver makes dispatch decisions with the objective of improving overall throughput while being fair to all kernels. However, there are multiple static resources to be allocated, and different kernels stress resources differently. Hence, it is not straightforward to "equally" divide resources.

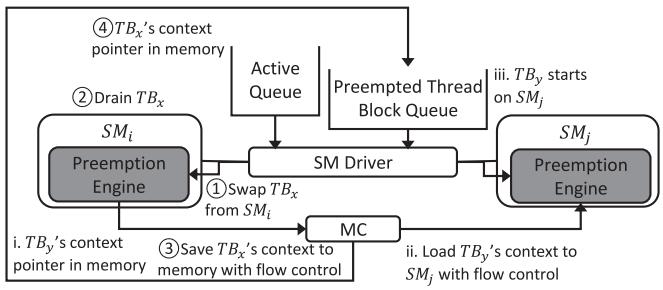


Fig. 2. SMK-supported SM. Added components are shaded. SM Driver controls the SMs and the Preemption Engine.

We adopt Dominant Resource Fairness (DRF), which is a generalized fairness metric for multiple resources [14]. The intuition behind DRF is multi-resource allocation should be determined by a kernel's dominant resource share, i.e., the maximum share that a kernel requires of any resource. In GPUs, there are four kinds of resources allocated during dispatch: registers, shared memory, number of active threads and number of TBs. These resources limit the total number of TBs that can be dispatched on the GPU. The dominant resource share of a kernel (rK) and SMs (rSM) is computed as:

$$rK(rSM) = \max\{r(Register), r(Threads), r(SharedMemory), r(TB)\},$$
$$where \quad r(x) = \frac{x_{taken}}{x_{limit}}.$$

Keeping track of kernel and SM resource usage is trivial as the resource demand of each TB in the kernel is determined at compile time. The SM driver is also fully aware of the TB status in each SM. Hence, the SM driver can use this resource information to determine a fair allocation.

4.3 Fair Resource Allocation

Once kernel and SM resource usage is defined, we can proceed to resource allocation with fairness. The objective is to equalize resource usage of each kernel (denoted as rK) as much as possible. This can be quantified as minimizing the difference between the maximum and minimum of rK of all kernels, which is termed the *range of* rK.

We first develop a naïve dispatch algorithm, termed *on-demand resource allocation*, which is a simple heuristic of the Knapsack problem. This algorithm forms the basis of the second algorithm, *resource partitioned algorithm*, which applies the same heuristic within a partition of resources.

4.3.1 On-Demand Resource Allocation

On-demand Resource Allocation described in Algorithm 1 tries to schedule a TB from the kernel, K_c, with the lowest rK onto the SM with the lowest rSM if there are enough resources on the SM for the TB. If not, resources are obtained by swapping out as many TB's as needed from the kernel with the highest rK on the SM with the highest rSM. This swapping, however, is done only if it reduces the range of rK. Otherwise, no preemption is done. Note that using the range of rK to determine when to do preemption is critical to performance. Specifically, once the range is relatively small, indicating that the resource allocation between different kernels is fairly balanced, then preemption will not be performed. This stablizes the algorithm and throttles overly frequent preemption, as observed in our study.

We use an example to illustrate the algorithm. Suppose there are only two kernels on a GPU, K_1 and K_2, with rK of 60 and 50 percent respectively. Hence, K_2 is a candidate kernel for dispatch. Further suppose there are not enough resources in the candidate SM to receive a TB from K_2. The SM driver will then search for a victim SM. When calculating rSM of all SMs, the resources taken by K_2 are excluded as if it does not exist in the entire GPU. Once a victim SM is identified, the SM driver calculates the range of rK for all kernels on the victim SM. Suppose rK for K_1 and K_2 would become 55 and 52 percent with a preemption. The range of rK is dropped from 10 percent (60-50 percent) to 3 percent (55-52 percent). Hence, preemption would improve the fairness between K_1 and K_2 and TBs from K_1 are swapped out and one TB from K_2 is swapped in.

4.3.2 Allocation with Resource Partitioning

On-demand Resource Allocation aims to achieve *global resource* balance between kernels. We observed that the algorithm cannot control the local resource allocation within each SM. To realize SMK, kernels of complementary resource usage should share resources within one SM. Consequently, the algorithm often generates an allocation with many SMs, each running only one kernel, close to Spart. Note that allocation schemes with fairness may not be unique. To achieve fairness in SMK, we propose a resource partitioned allocation strategy.

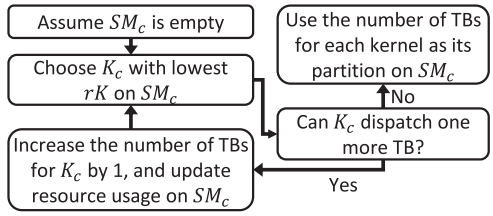


Fig. 3. The procedure of resource partitioning.

Algorithm 1. On-demand Resource Allocation

1: $K_c \leftarrow$ kernel with the lowest rK
2: $SM_c \leftarrow$ SM with the lowest rSM
3: **if** There are enough resources in SM_c to receive a TB from K_c **then**
4: **if** K_c has TBs in the Preempted Thread Block Queue **then**
5: Swap in a previously preempted TB to SM_c
6: **else**
7: Dispatch a new TB to SM_c
8: **end if**
9: **else**
10: $SM_v \leftarrow$ SM with the highest rSM excluding resources taken by K_c
11: $K_v \leftarrow$ kernel on SM with the highest rK
12: Calculate the range of rK on SM_v
13: **if** The range is reduced with preemption of K_v **then**
14: Swap out just enough TBs of K_v to hold a new TB from K_c.
15: **end if**
16: **end if**

This improved algorithm works as the follows. The SM driver first identifies a candidate kernel K_c and a candidate SM SM_c similar to On-demand Resource Allocation. Then, the SM Driver checks whether there is a partition for K_c in SM_c. If so, the driver checks whether K_c's partition has enough resources to receive another TB from K_c. With enough resources, the SM Driver will dispatch a TB. If there is no partition for K_c in SM_c, the driver initiates partitioning for SM_c. After partitioning, TBs of existing kernels on SM_c that exceed their own partition capacity are swapped out to make room for new TBs from K_c. The partitions are stored in the SM driver with several counters per SM, which add negligible cost.

To create a partition, the SM Driver applies the DRF policy [14]. It aims to equalize rK for all kernels on one SM. As charted in Fig. 3, partitioning starts with an empty SM where rK of each kernel is 0. Then the driver iteratively adds TBs from kernels with the lowest rk until no more TBs can be dispatched to this SM. That is, at least one type of resource in the SM has been exhausted. At this time, a partition of the resources between all kernels has naturally been created—the resources occupied by each kernel through this iterative procedure form the resource partition for those kernels. The procedure always attempts to reduce the range of rK: By dispatching a TB from the kernel with the lowest rK its rK is increased, contracting the range.

With partitioned resource allocation, every SM has a dedicated partition for each kernel, guaranteeing that multiple kernels can co-execute within one SM.

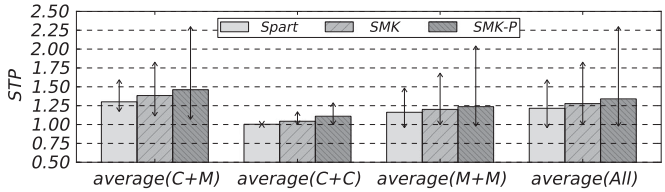


Fig. 4. System Throughput (STP).

5 Preliminary Evaluation

We evaluate our design using the latest version of GPGPU-Sim [15]. We use the parameters (core/memory frequency, the number of memory controllers, the number of SMs, the resources in each SM and the number of warp schedulers) from the specifications of NVIDIA Geforce GTX980 [16] to reflect that each SM is larger in recent architectures. We also modified GPGPU-Sim to run multiple kernels on the same SM.

We used 10 benchmarks from the Parboil benchmark set [6] and we paired the benchmarks to simulate multiprogrammed workloads. Each pair runs 2M cycles in our evaluation, and all instructions and cycles are accumulated to calculate the IPC. According to [5], the results are accurate when the simulation is longer than 1 M cycles.

Fig. 4 shows the average system throughput (STP)[17] of spatial partition (**Spart**), On-demand Resource Allocation (**SMK**) and Allocation with Resource Partitioning (**SMK-P**) for compute-intensive and memory-intensive kernels (C+M), two compute-intensive kernels (C+C), two memory-intensive kernels (M+M). The result shows that SMK improves the average STP over sequential execution by 34 percent, and over Spart for all pairs by 10 percent with the highest being 70.7 percent from a C+M workload. The average performance fairness [17] of Spart, SMK and SMK-P are 0.70, 0.64 and 0.59. This result shows that resource fairness does not guarantee performance fairness. The main reason is the warp scheduler in the SM is not aware of multiple kernels. All TBs are viewed as if they are from a single kernel. As a result, the warp scheduler may favor warps from one kernel over the other, leading to bad performance fairness. Hence, a future direction is to enhance warp scheduling with kernel awareness for better fairness under SMK.

6 Conclusion

This paper propose a fine-grained multitasking GPU design to increase resource utilization and enable dynamic resource allocation. Moreover, we propose several strategies to fully exploit the potential of this mechanism. With these strategies, we maintain the resource fairness among kernels, and improve system throughput. We will explore designs and strategies to further improve performance and fairness. We are also going to examine the scenarios with three or more kernels.

Acknowledgments

This work is supported in part by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (Nos. 61261160502, 61272099), the CSC scholarship, US National Science Foundation (NSF) grants CNS-1012070, CNS-1305220, and CCF-1422331.

References

[1] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proc. 18th Int. Conf. Archit. Support Programm. Lang. Operating Syst.*, 2013, pp. 407–418.

[2] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, Feb. 2014, pp. 260–271.

[3] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 593–606.

[4] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 193–204.

[5] J. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Proc. IEEE 18th Int. Symp. High-Perform. Comput. Archit.*, Feb 2012, pp. 1–12.

[6] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. Available: http://goo.gl/iyFqIr

[7] T. Bradley. (2012). Hyper-Q example. [Online]. Available: http://goo.gl/hh84by

[8] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-Centric program transformations," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 119–130.

[9] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber, Phys. Soc. Comput.*, Dec 2010, pp. 344–350.

[10] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proc. 4th USENIX Conf. Hot Topics Parallelism*, 2012, p. 10.

[11] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2014, pp. 301–316.

[12] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX ATC*, 2011, pp. 17–30.

[13] S. W. Kim, C. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar, "Reference idempotency analysis: A framework for optimizing speculative execution," in *Proc. SIGPLAN Symp. Principles Pract. Parallel Program.*, 2001, pp. 2–11.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, vol. 11, pp. 24–24.

[15] A. Bakhoda, G. L. Yuan, W. W. L Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2009, pp. 163–174.

[16] NVIDIA. (2014). NVIDIA Geforce GTX980 Whitepaper. [Online]. Available: http://goo.gl/ugSaEU

[17] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.