View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

CUsched: Multiprogrammed Workload Scheduling on GPU Architectures

UPC-DAC-RR-CAP-2013-7 Technical Report, March 2013.

Ivan Tanasic[1], Isaac Gelado[1], Javier Cabezas[1],
Nacho Navarro[1,2], Alex Ramirez[1,2], and Mateo Valero[1,2]
[1]Barcelona Supercomputing Center
[2]Universitat Politècnica de Catalunya
first.last@bsc.es

***Abstract*—Graphic Processing Units (GPUs) are currently widely used in High Performance Computing (HPC) applications to speed-up the execution of massively-parallel codes. GPUs are well-suited for such HPC environments because applications share a common characteristic with the gaming codes GPUs were designed for: only one application is using the GPU at the same time. Although, minimal support for multi-programmed systems exist, modern GPUs do not allow resource sharing among different processes. This lack of support restricts the usage of GPUs in desktop and mobile environment to a small amount of applications (e.g., games and multimedia players).**

In this paper we study the multi-programming support available in current GPUs, and show how such support is not sufficient. We propose a set of hardware extensions to the current GPU architectures to efficiently support multi-programmed GPU workloads, allowing concurrent execution of codes from different user processes. We implement several hardware schedulers on top of these extensions and analyze the behaviour of different work scheduling algorithms using system wide and per process metrics.

I. Introduction

Graphics Processing Units (GPUs) used to be fixed-function circuits designed to speedup the execution of graphic manipulation tasks commonly required by gaming and video processing applications. Currently, GPUs have become fully programmable many-core processors [16] that are able to efficiently execute both, traditional graphics workloads, and general purpose codes, such as media processing, medical imaging, and High Performance Computing (HPC) applications [18].

GPU applications, both graphic and general purpose, assume exclusive access to the GPU resources. This is a reasonable assumption in most existing scenarios. Desktop and laptop users do not play video games while watching a movie. Supercomputer job schedulers are configured to grant each user process with exclusive access to each compute node. Hence, GPU designers have safely assumed that only one user process will be using the GPU at the same time. However, new scenarios where several applications concurrently access to a single GPU are starting to appear. Many MatLab users have discovered that their system becomes unresponsive when MatLab is using GPU for computations. HPC application programmers underutilize the CPU by restricting the number of MPI processes spawn on each node to match the number of attached GPUs. Cloud computing providers do not offer virtual GPU servers[1] due the inability of effectively sharing GPUs among several processes. Furthermore, with the integration of programmable GPUs into mobile Multi-Processor System on Chip (MPSoC) [4], [1], [2], GPU sharing is very likely to happen. For instance, consider a GPS application recalculating a route in the background while the user is using a video conference application. These real and hypothetical scenarios lead us to believe that the exclusive-access assumption in the GPU design needs to be reconsidered.

Traditionally, the interface between the GPU and the CPU was based on a single hardware queue where different user processes pushed commands (i.e., kernel invocations and/or DMA transfers). The Operating System (OS) GPU driver ensured exclusive access to this hardware queue: a user process cannot start issuing commands to the GPU until the hardware queue has become empty, and the OS driver has saved the GPU state of the previous user process (i.e., GPU context switch). The GPU hardware inspects this input queue, and redirects each command to the input command queue of the appropriate hardware engine (execution or DMA), if no dependencies with previously issued commands exist. Such design allows independent commands from the same user process to be executed concurrently, as long as each command is executed by a different hardware engine, e.g., DMA transfers and kernels can be executed concurrently in most modern GPUs. However, commands sent to the input queue of each engine are always executed in batches, i.e., a kernel cannot start executing until the previous kernel has finished execution or has been fully submitted[2].

Modern GPU architectures, such as the NVIDIA Kepler GK110, implement several hardware command queues, i.e., NVIDIA Hyper-Q technology [17]. This GPU interface allows different processes to issue commands to the same GPU without the need of the OS driver to context switch the GPU each time a command from a different process is issued. It enables the GPU to concurrently execute DMA transfers and kernels from different processes, but the GPU

[1]Notice that dedicated GPU servers are indeed offered which shows the demand for this type of service.

[2]NVIDIA Fermi and Kepler architectures allow work from a kernel to be scheduled for execution once all the work from the previous kernel has been issued. Please refer to Section II for detailed explanation.

still requires exclusive per-process access to each of the GPU engines. In this paper we go one step further to remove such restriction and enable concurrent multi-process execution on the GPU computational engine, completely removing the exclusive-access constraint from the GPU design. We present a novel light-weight GPU preemption scheme to enable fine grained GPU resource sharing across several user processes and show how it introduces minimal overhead in return for great flexibility in realistic workloads. We exploit the GPU programming model and hardware characteristics to preempt kernel execution at the thread-block (i.e., task) granularity, and DMA transfers at the PCI-express packet boundary. We extend a NVIDIA Kepler-like base architecture with a minimal set of hardware logic and structures to implement our proposed preemption scheme. We build upon this hardware modifications to implement several FCFS scheduling schemes, preemptive and nonpreemptive priority queues schedulers, and the token scheduler. We also show how these schedulers can accomplish GPU time sharing, spatial sharing and prioritization in multi-programmed environments and how different policies affect different metrics.

Our experimental evaluation shows that the hardware support for multi-programmed scheduling introduced in this paper allows scheduler implementations that on average for eight co-scheduled applications improve the overall system throughput by 20%, fairness by 85%, decrease normalized turnaround time by 38% and decrease the normalized turnaround time of high priority applications 3x.

This paper is organized as follows. Section II presents the necessary background and motivation for this work. Section III presents the hardware modifications required to add multi-programming support to existing GPU architectures. In this section we also present the design of a token scheduler to illustrate the trade-offs and intricacies of fine grained scheduling on GPUs. The experimental methodology and results are discussed in Section IV. Related work is presented in Section V, while Section VI concludes this paper.

II. Background

This section introduces the background material and motivates the work proposed in the rest of the paper.

A. Base GPU Architecture

The base GPU architecture we assume in this paper is depicted in Figure 1. In its simplest form, the GPU interface, similar to the one implemented by NVIDIA Fermi GPUs, has one queue which is used by the host to issue commands (i.e., DMA transfers and kernel launches) to the GPU. Commands from the same context are processed by the *command dispatcher*, which inspects the top of the queue, separates independent commands (i.e., commands from different CUDA streams or OpenCL command queues) and redirects them to the corresponding engine. Data transfer commands are sent to the data transfer engines via DMA upstream and downstream queues while kernel calls are sent to the execution engine via the execution queue. This allows overlapping of data transfers with kernel execution. The command dispatcher ensures that commands coming from the same stream are executed sequentially, following the semantics of the programming model.

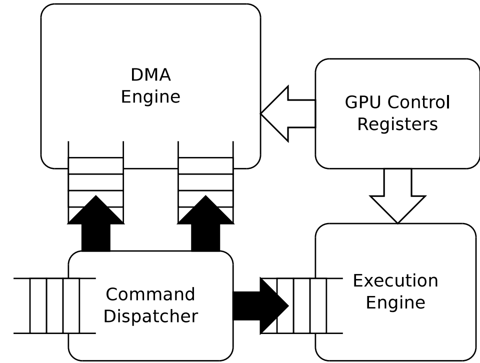


Fig. 1. Single Process GPU architecture.

The GPU also includes a set of global control registers (i.e., GPU context), used by the execution and data transfer engines. This control registers hold process-specific information, such as the location of the virtual memory structures (e.g., page table), the GPU kernels registered by the process, or the graphics buffers (e.g., OpenGL images) allocated by the process. When a single command queue is implemented by the GPU interface, there is only one active GPU context used by all execution engines in the GPU, limiting the hardware to only execute commands from a single user process. GPU sharing among user processes is done by the OS kernel GPU driver by performing a GPU context switch. First, the OS code ensures that the GPU command queue is empty. Then, the OS code saves the current GPU context into system memory, and loads the new GPU context. Finally, commands from the new process are submitted to the GPU command queue. This heavy-weight GPU context switch procedure introduces large performance overheads if several processes are contending for the GPU.

The GPU architecture we build upon in this paper assumes several GPU command queues, and thus several GPU context data structures, to mimic the Hyper-Q technology implemented in the latest GK110 NVIDIA chips [17]. Hyper-Q removes the performance overheads due to false serialization and GPU context switching, and overlaps transfers with computation of different processes. However, Hyper-Q does not support concurrent execution of commands from different processes on the same engine, i.e., only kernels from the same user process can be concurrently executed.

B. Base GPU Execution Engine

The base GPU *execution engine* assumed in this paper is shown in Figure 2. It reads one entry at a time from the input queue, and sets up the *Kernel Status Registers* (KSR), with the control information (e.g., number of work units to execute, kernel parameters, . . .). The contents of these registers, as well as the global GPU control registers, are used to setup the GPU cores (i.e., Streaming Multiprocessors in CUDA terminology). Then, the scheduler starts sending work items (thread blocks in CUDA terminology) to each GPU core (SM), until they are fully utilized (i.e., run out of hardware resources). Whenever

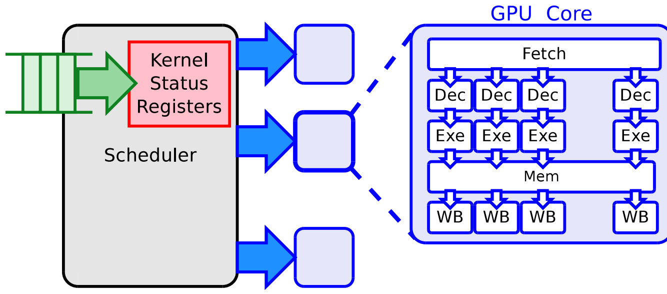


Fig. 2. Baseline execution unit.

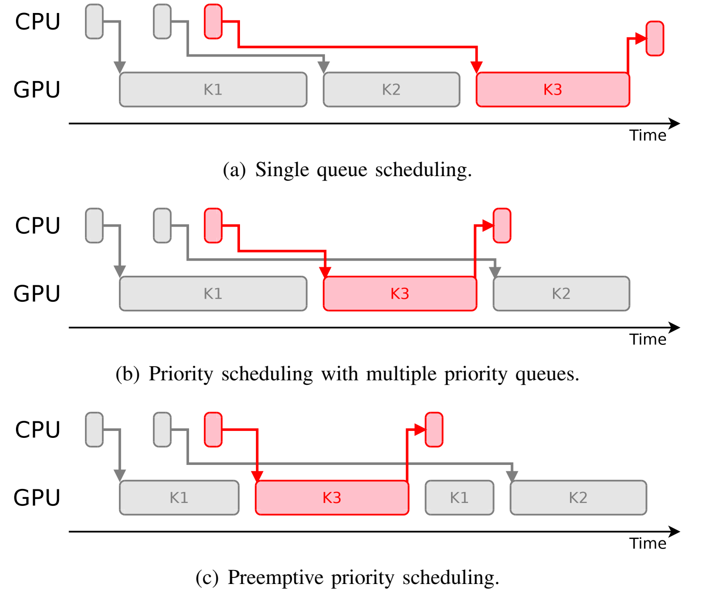
a SM finishes executing a thread block, the scheduler gets notified and schedules a new thread block to that SM.

Because of its legacy as graphics accelerator, the GPUs are designed to maximize the computation throughput of a single process: SMs are only configured just before a new kernel starts its execution and thread blocks are scheduled as soon as hardware resources become available. Depending on the number of threads in the thread block and the static usage of registers and shared memory, a limited number of thread blocks[3] can be scheduled to one SM. Once a thread block is scheduled to an SM, the SM can execute only thread blocks from the same kernel and will not be released until all the thread blocks from the running kernel have been scheduled. This policy means that thread blocks from one kernel can easily occupy all the available SMs, forcing other independent kernels (i.e., associated to a different stream) to stall. As a result, concurrent execution among kernels is possible only if the kernel that was scheduled first does not have enough thread blocks to occupy all SMs or the scheduled kernel is finishing its execution and SMs are becoming free again.

C. Motivation

GPU resource sharing can lead to better utilization of available resources (e.g., global memory bandwidth) and optimizing the system wide metrics [10], [9], [5]. Two ways of sharing a GPU are possible: time sharing, i.e., using the resources in turns, and spatial sharing, i.e., splitting the resources. Existing GPUs implement a very limited form of spatial sharing, as discussed earlier, while time sharing can be only done with kernel granularity. This time sharing granularity, however, might not be fine enough. For instance, in the NAMD molecular dynamics application [3] the application critical path requires waiting for boundary data to arrive through the network, via Message Passing Interface (MPI), executing a GPU kernel, and sending the output data through MPI. Execution on this critical path tends to get delayed because when the boundary data arrives to the node, the GPU is already busy executing the bulk computation of the simulation and, thus, the GPU kernel in the critical path needs to wait for this much larger kernel to finish. If the code is modified to delay the execution of the big GPU kernel until the MPI message arrives, the bulk computation becomes the critical path and, indeed, takes longer to execute than the previous approach.

[3]Up to sixteen in the NVIDIA Kepler GK110 architecture.



(a) Single queue scheduling.

(b) Priority scheduling with multiple priority queues.

(c) Preemptive priority scheduling.

Fig. 3. Kernel scheduling.

If the GPU would allow launching a high-priority kernel that can concurrently execute with the bulk computation, the application execution time would significantly decrease.

Figure 3 illustrates how existing GPUs are not suitable to execute soft real-time applications. In this example, a soft real-time application requires executing the kernel K3 while other applications are executing kernels K1 and K2. Figure 3(a) illustrates the case of a GPU with a single command queue, where kernels are executed in the order they were issued. In this case, the kernel $K3$ needs to wait until all previous kernels finish executing before being scheduled to run. A minor modification to the architecture with multiple queues (i.e., Hyper-Q) could easily allow to define different execution priorities for these queues. In that case, illustrated in Figure 3(b), the high-priority kernel execution latency decreases, since it can be scheduled to run as soon as the kernel $K1$ finishes execution. However, the execution latency of $K3$ is still unpredictable since it depends on the execution time of a kernel previously issued by another user process. The mechanisms we present in this paper allow decreasing this latency further, resulting in the timeline shown in Figure 3(c). In this case, as soon as $K3$ is issued, the GPU starts scheduling thread-blocks for this kernel to the GPU as resources become available. Although the execution time for $K3$ still depends on the GPU load, a maximum latency before $K3$ fully occupies the GPU can be given, which suffices for most soft real-time applications.

D. Multi-Programmed Scheduling

Many different scheduling algorithms (e.g., round-robin, priority scheduling, lottery scheduling, etc.) have been proposed for multi-programmed systems, where computational resources (e.g., CPUs) are time-shared among several user processes. However, scheduling on general purpose systems significantly differs from GPUs because of different design trade-offs.

Schedulers aim to optimize some given metric, i.e., **throughput** (number of work units finished per unit of time), **turnaround time** (total time between a submit and its completion), **response time** (time between a submit and its processing starts) and **fairness** (how time assigned is proportional to assigned priorities)[8]. In most cases, the optimization of one metric tends to harm the others (e.g., throughput vs. turnaround time) so schedulers try to find the correct trade-off for the specific system where they are deployed. For instance, turnaround time and/or response time tend to be prioritized over the other metrics on interactive systems). In this paper we focus on the scheduling mechanisms, rather than scheduling policies. However, to justify the proposed mechanisms and show their effectiveness, we compare several different scheduling policies, explain their implementation and analyze their effects on these metrics.

Three schedulers (in several variations) that are suitable for a hardware implementation on the GPU are explored in this paper: *First Come First Served*, *Preemptive Priority Queues* and *Token Based* schedulers. The first two are well known schedulers. Token scheduler assigns a number of tokens to each running kernel and every time an SM is assigned to the kernel, the token count is decremented. Thread blocks are selected from a kernel with the higher token count.

III. GPU Architecture with Multi-programming Support

The multiple queue GPU architecture (i.e., Kepler GK110) described in Section II keeps track of several GPU contexts (up to the number of queues it has), but only one can be active in each unit: one in execution unit and one per each DMA engine available in the GPU. To further support concurrency in the execution unit, it has to be extended to allow commands from multiple contexts to be issued concurrently and SMs need to be extended to keep track of the context they are assigned to.

A. Execution Engine Multi-Programming Support

With only one active context in the execution unit, all the architectures described in the Section II need an execution engine with only one command queue and one KSR. To enable concurrent execution of kernels from different contexts we extend the command dispatcher and GPU execution engine and connect them with multiple command queues (as many as there are contexts in the system). We also extend the execution engine to include as many KSRs as there are SMs in the system and augment each KSR with the identifier of the GPU context where the kernel is being executed. Anywhere between zero and all SMs can be assigned to each context. Notice that in some implementations, GPU context information still needs to be a global GPU structure since the data transfer engine also accesses this information. Besides these modification, each scheduler also has to implement the additional data structures required by the particular scheduling policy.

Any scheduling mechanism implemented in the GPU needs to deal with the ability of SMs to execute several thread blocks in an interleaved fashion. SMs are based on static hardware partitioning, so all the available registers, shared memory and threads are split among all the thread blocks in the SM[4]. The resource usage of all the thread blocks from a kernel are the same and known in advance. The number of issued thread blocks is thus determined by the first fully used resource. This parameters have to be setup in each SM to partition the hardware resources before the first thread block is scheduled to run. Static hardware partitioning implies that only thread blocks from the same kernel can be scheduled to concurrently run on the same SM. Hence, a thread block can be scheduled to a given SM only if:

1) The thread block belongs to the same kernel that is being executed on that SM and there are hardware resources available for more thread blocks.
2) There are no thread blocks being executed by the SM.

Adhering to these rules, however, gives little scheduling freedom. Like in general purpose systems, execution preemption is a third option that could create more scheduling opportunities. A potential approach to preempt the current kernel execution, in the very same way as operating systems do on the CPU. This approach, currently used in GK110 chips to implement dynamic parallelism, presents one major drawback when applied to GPU scheduling: the SMs keep a very large state (e.g., a 48KB scratch-pad memory and 512KB register file per SM in the latest GK110 chip), which would make the context switching a very long latency operation, possible longer than the kernel execution itself.

The mechanisms we propose in this paper allow a less expensive option: preemption of kernel execution on a thread block boundary. The scheduler can drain the SM by not issuing thread blocks to the given SM until all the previously issued work to that SM finishes. When the SM becomes idle, the scheduler can set it up for the new kernel and start issuing thread blocks to it. Thus, a scheduler can implement time sharing, spatial sharing, prioritization or any combination of the three. Lower utilization of the execution engine due to the SM draining and setup overhead are important trade-offs in the scheduling decision: scheduling thread blocks from the same kernel minimizes the overheads, but prevents the GPU resources to be shared across all the contexts.

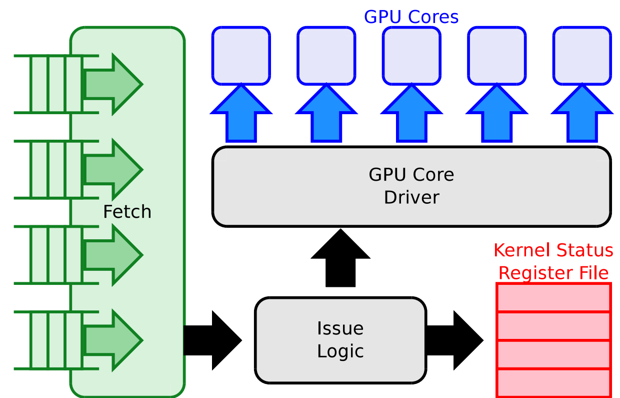


Fig. 4. Architecture of the multi-kernel execution engine.

[4]Available resources of the GK110 chip are given in the Section IV

B. Streaming Multiprocessor Multi-Programming Support

Since in the baseline GPU architecture only one GPU context is actively executing kernels, GPU context information can be easily extracted from the global GPU control structures. Now that we have enabled the execution engine to be shared by multiple contexts, we modify the baseline SM architecture by extending its state to include a *GPU context id register*, a *base page table register*, and a *texture register*. The GPU work scheduler sets the values of these registers during the SM setup, before the issue of the first thread block. The base page table register is used to configure the memory management unit (MMU) of the SM (notice that there is only one MMU per SM). The content of the base page table register is used on a TLB miss to walk the per-process page table stored in global memory. This is in contrast to the base GPU architecture, where a single page-table walker sufficed, since the same page table was used by all SMs. Similarly, the contents of the texture register are used to index the per-process texture table when textures are fetched by the texture unit. Finally, the GPU context id register is used when GPU objects (e.g., OpenGL images) are accessed by the SM. We extend the context of the SM, rather than using global GPU structures to prevent them from having many read ports (one per SM). The extra silicon area required for these modifications is negligible; on a 64-bit SM, 192 bytes of extra storage are required, which represents a 0.07% of the size of the register file. No modifications are required to the memory hierarchy because SMs in most modern GPUs implement a virtually indexed, physically tagged first-level cache memory. Hence, any memory requests coming in and out of the L1 cache use physical memory addresses.

C. Example: Token Scheduler

The GPU architecture described in this paper allows different work schedulers to be implemented on top of it. Now we discuss the implementation and design trade-offs of a token based scheduler to illustrate most of the intricacies of work scheduling on GPUs.

A token scheduler is designed to perform dynamic spatial sharing of GPU resources (SMs). This scheduler allows the OS and runtime system to assign a number of tokens to each GPU context. The token count of each GPU context is incremented when an SM is assigned to it, and decremented when an SM is deassigned. The lowest initial value for the token count is zero, so its value can reach negative numbers up to the number of SMs in the GPU. On scheduling points (details to follow), the scheduler selects the kernel with the highest token count, looks for free SMs, assigns them to the GPU context and decrements its token count accordingly. If all SMs are busy, the scheduler starts draining those SMs executing kernels with lower token counts, and schedules the kernel to them. The token count for each kernel is decremented as soon as an SM is assigned to it, and incremented as soon as an SM is deassigned. This algorithm ensures that in steady state the token count for all active kernels is the same.

Figure 5 shows the block diagram for the GPU token scheduler. This scheduler requires two additional data structures to keep the control information:

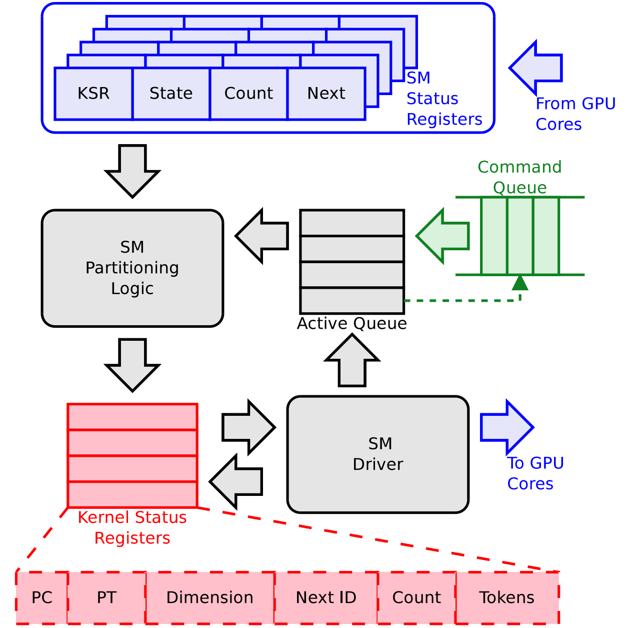


Fig. 5. Block Diagram for a Token Scheduler.

a) Active Queue: this is a circular queue where the *Kernel Status Register* indexes of the kernels being executed are stored. This queue has as many entries as KSRs are in the system and each valid entry contains a KSR index. Any other control information for the kernel (e.g., current token count) is kept in the KSR itself. When there are free entries in the active queue the fetch logic pops one entry from the execution engine input queues to the corresponding KSR. Besides the KSR index, an extra bit is needed to mark busy and free queue entries.

b) SM Status Registers (SMSR): this is a register file with as many entries as SMs in the system. The contents of each register are updated by the *SM Driver*. Each entry contains the KSR index of the kernel the SM is executing, the state of the SM (*Idle*, *Running* or *Reserved*), number of running thread blocks and the KSR index of the next kernel (when in *Reserved* state).

The scheduling logic is divided in two main components: the *SM Partitioner*, and the *SM Driver*. The SM partitioner logic uses the contents of the SMRSs, and the active queue to partition the available SMs among the running kernels. When a new kernel is inserted in the active queue, or one running kernel gets removed from the running queue (finishes the execution), the SM partitioning procedure is triggered. The SM partitioner picks the kernel with the highest current token count and checks if there are any *Idle* SMs in the system by reading the SMSRs. If there are, the token count is decremented and the SM driver can setup the SM and start issuing the thread blocks to that SM. If no idle SMs are available, the partitioner finds the running kernel with the lowest current token count and switches the state of one of its assigned SMs from *Running* to *Reserved* in the SMSR, starting the process of draining the SM. It also increments the token count of the kernel that is being drained from the SM

CPU	GPU
Clock: 2.8 GHz # Cores: 4 L1 Cache (per core): 32 KB D + 32 KB I L2 Cache (shared): 12 MB Memory Clock: 1066 MHz	Clock: 706 MHz # Cores: 13 (32 pipelines each) L1 Cache (per core): 16 KB (128 bytes line) 64-way 50 cycles L2 Cache (shared): 768 KB (32 bytes line) 64-way 150 cycles GDDR3 Memory: Clock: 1546 MHz Width: 512 bits 6 Banks Registers (per SM): 65536 Thread Blocks (per SM): 16 Threads (per SM): 2048
PCIe Bus	
Clock: 500 MHz Lanes: 32 Burst: 4 KB	

TABLE I
SIMULATION PARAMETERS USED IN THE EXPERIMENTAL EVALUATION.

Kernel Mnemonic	Number of Launches	Avgerage Exec. (us)	Thread Blocks	Thread Block Average Exec.
textbfs				
BFSmultiblkinGPU	1	35855.01	14	2561.07
BFSinGPU	2	13006.62	1	13006.62
cutcp				
cutpotentiallattice	11	1519.54	121	12.56
histo				
histofinal	20	70.37	42	1.68
histoprescan	20	22.46	64	0.35
histointermediates	20	79.77	65	1.23
histomain	20	372.47	84	4.43
lbm				
StreamCollide	100	2906.77	18000	0.16
binningReconstruction	1	2026.69	5188	0.39
mri-gridding				
scaninter1kernel	9	8.49	29	0.29
scanL1kernel	8	826.98	2084	0.40
uniformAdd	8	128.44	2084	0.06
reorderReconstruction	1	2540.83	5188	0.49
splitSort	7	3843.89	2594	1.48
griddingGPUsample	1	191356.28	65536	2.92
splitRearrange	7	1622.76	2594	0.63
scaninter2kernel	9	9.97	29	0.34
mri-q				
ComputeQGPU	2	3391.58	1024	3.31
ComputePhiMagGPU	1	5.82	4	1.46
sad				
largersadcalc8	1	8139.52	8040	1.01
largersadcalc16	1	1519.52	8040	0.19
mbsadcalc	1	15452.74	128640	0.12
sgemm				
mysgemmNT	1	3713.92	528	7.03
spmv				
spmvjds	50	290.41	765	0.38
stencil				
block2Dregtiling	100	2230.80	256	8.71
tpacf				
genhists	1	14604.03	201	72.66

TABLE II
DESCRIPTION OF THE BENCHMARKS AND KERNELS USED IN THE EXPERIMENTAL EVALUATION.

and decrements the token count of the newly assigned kernel. This procedure is repeated until all the active kernels have the same current token count.

The SM driver is in charge of issuing the thread blocks. Thread blocks are issued when the new partitioning occurs or whenever a SM notifies the driver that a thread block has been finished executing. The SM driver checks the state field of the corresponding SMSR which can be in one of the following states:

c) Running: the SM is requesting a new thread block from the running kernel. If there are pending thread blocks from the kernel, the driver schedules the next thread block. This is a fast operation which only requires the SM to add a new entry in its active thread block list and start executing. If there are no pending thread blocks the state field for the SMSR is set to *Idle* and the SM partitioner is notified that the kernel has finished execution, triggering the previously discussed scheduling procedure.

d) Reserved: if there are no thread blocks running on the GPU core (i.e., the *count* field in the SMSR is zero), the SM requests the driver to schedule a thread block from the kernel which reserved the GPU core (i.e., next field in the SMSR) in a previous scheduling round. Otherwise, no new thread blocks are scheduled.

e) Free: the SM is configured with the kernel control information stored in the KSR: page table base address, program counter, and dimensions (i.e., number of threads in the thread block, number of register used, and amount of scratch-pad memory required). After the SM is configured, the SM driver can start issuing the thread blocks to that SM.

IV. Experimental Evaluation

In this section we present the experimental results that show that extensions and ideas presented in this paper allow efficient hardware scheduler implementations that enable the programmers and the operating systems to optimize the system performance measured with both system level and user level metrics.

A. Methodology

For the experimental evaluation of the proposals presented here we use an in-house trace-driven simulator that models a multi-core CPU connected to a GPU through a PCIe bus using the parameters from Table I. This simulator performs a coarse grained modeling of the CPU, using timing measurements gathered on an Intel Core i7 930 chip. CPU traces consist of a starting and ending time stamp for each API call to the CUDA runtime library. We perform a cycle accurate simulation of the PCIe bus and the GPU using execution traces of each GPU kernel. For the GPU simulation, we used parameters from Table I which are taken from Baghsorkhi et al. [6]. Applications are simulated from the first call to the CUDA runtime until the end of the application, capturing all the memory transfer, kernel execution and CPU execution phases.

We use the full Parboil benchmark suite to evaluate our proposals. Table II shows the GPU kernels (with slightly shortened names) executed by each of the benchmark applications. Included are the number of kernel calls, average execution time of the kernel, number of thread blocks and average thread block execution time for each of the kernels.

We create multi-programmed workloads as combinations of several benchmark applications chosen randomly, which we use in all experiments. To evaluate different workload characteristics, we categorize each benchmark as short running or long running, based on the average execution time of the

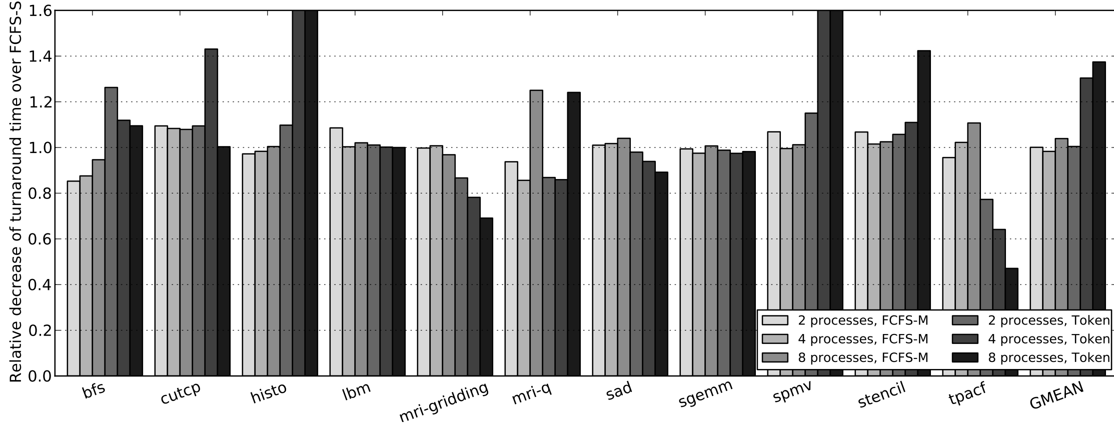


Fig. 6. Relative turnaround time improvement over *FCFS-S*.

thread blocks when running in isolation and we ensure that each experiment includes a uniform distribution of combinations for each category (short-short, long-long and short-long). We run all benchmarks in each combination until completion. Because some benchmarks end much earlier than others, we replay benchmarks as soon as they finish executing until all benchmarks have been executed once, as proposed in [22], while gathering statistics for the first execution of each benchmark only.

Three different scheduler configurations are used in most experiments. *FCFS-S* is a First-Comes-First-Served scheduler with the single active context in the execution engine, similar to current GPUs (including the Kepler GK110). *FCFS-M* is the FCFS scheduler with multiple active contexts in the execution engine enabled by the hardware extensions proposed in Section III. *Token* is the token scheduler described in detail in section III. Unless stated otherwise, all three schedulers are setup so that all contexts have the same priority. For the token scheduler, this means that the tokens are evenly distributed among applications while keeping their total number smaller or equal to the number of SMs in the GPU. When appropriate for the particular experiment, we define other scheduling schemes and describe them before using in the experiments. All the metrics used in the experiments are calculated as suggested by Eyerman et al. [8].

B. Multi-Programmed GPU Benchmarks Performance

We first evaluate the impact of co-scheduling on the execution time of applications when using different schedulers. Figure 6 shows the speedup of the turnaround time of *FCFS-M* and *token* over the default *FCFS-S*.

We identify two clear patterns in our experimental results. The *token* scheduler decreases the performance of those benchmarks with long-running kernels (i.e., *bfs*, *mri-gridding*, *sad*, and *tpacf*), while *FCFS-M* improves its performance. As the concurrency level grows, these benchmarks start sharing the GPU resources when using *token* and, thus, the kernel execution time increases. This also explains why the performance of *mri-gridding* and *tpacf*, the two benchmarks with the longest running kernels (notice the long average execution time in Table II) decreases when using the *token* scheduler. *FCFS-M* improves the performance of *bfs*, and *tpacf* as the concurrency level grows. This improvement is due to the ability of this scheduler to start executing thread-blocks for this benchmarks as soon as GPU resources become available.

The second pattern is the large benefits of the *token* scheduler for benchmarks with short-running thread-blocks (*histo*, *spmv*, and *stencil*). In this case, both *FCFS-S*, and *FCFS-M* hold the execution of these short-running kernels for relatively long periods of time if a long-running kernel has been previously scheduled to run. However, *token* starts executing thread-blocks for this benchmarks as soon as the kernel is issued. As a result, the execution time for each individual thread-block is longer than when using *FCFS* schedulers, but the wait time in the execution time is much shorter.

Cutcp presents an interesting behaviour when using the *token* scheduler. If only two processes are running, the execution time is similar for all schedulers; this behaviour is due to the lack of contention on the GPU, i.e., only one kernel is active in the GPU. When we increase the level of concurrency to four processes, *token* produces large speedups because of the short-running kernels which benefit from the shorter wait time in the GPU when there is contention. However, when *cutcp* is executed with seven other benchmarks, the contention in the GPU execution engine increases, and *cutcp* gets one or two SMs, which in this benchmark can only execute one thread-block. Moreover, the execution time of thread-blocks in *cutcp* is relatively long. Due to these two factors, the kernel execution time is heavily affected by the amount of thread-blocks that can be executed concurrently in the GPU.

In addition to the turnaround time which is an application oriented metric, it is useful to evaluate the effects of schedulers on the overall system performance. The first system oriented metric we evaluate is the fairness. Fairness is the measure of equal progress of co-scheduled applications relative to their isolated execution. It is a higher-is-better metric with fairness of one meaning that all the co-scheduled applications experience exactly the same progress while fairness of zero shows that some application completely starve. Figure 7 shows the average system fairness achieved with

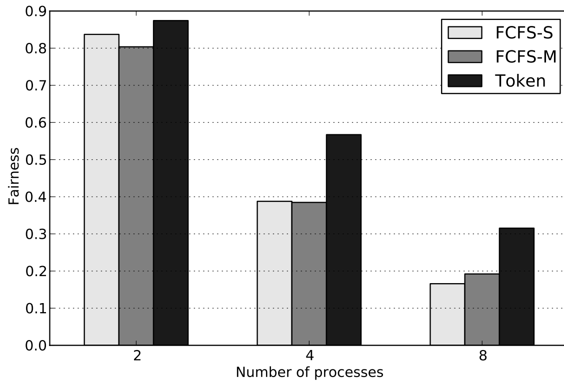


Fig. 7. Average scheduler fairness.

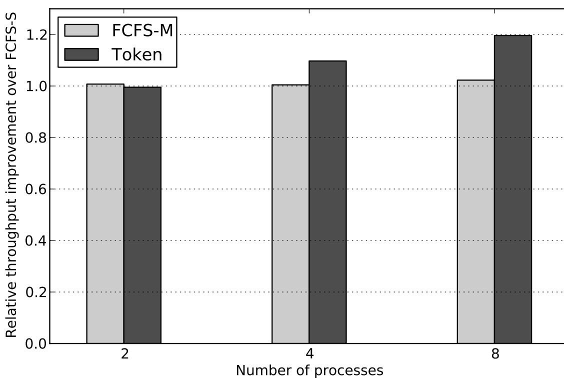


Fig. 8. System throughput improvement over *FCFS-S*.

different schedulers when all the co-scheduled applications in a workload are assigned equal priorities. When there are only two applications running in the GPU, fairness is relatively high with all schedulers due to the low contention of the execution engine. Still, the *token* scheduler has close to 5% and 9% improvement in fairness over *FCFS-S* and *FCFS-M*, respectively. As the number of co-scheduled applications grows, so does the contention and the fairness improvement of *token* over the other two schedulers. Token achieves higher fairness because equal application priorities (equal number of tokens assigned) mean that SMs are partitioned among the running kernels and that an application will always be able to progress. Still, when there are less concurrent kernels running in the execution engine than co-scheduled applications, the rest of the SMs is equally divided among the running kernels. The *token* scheduler improves the fairness around 46% when four applications are co-scheduled and 85% and 63% when eight applications are co-scheduled, compared to *FCFS-S* and *FCFS-M*.

Another important system oriented metric is the system throughput, defined as number of applications completed per unit of time. Average system throughput is shown in Figure 8.

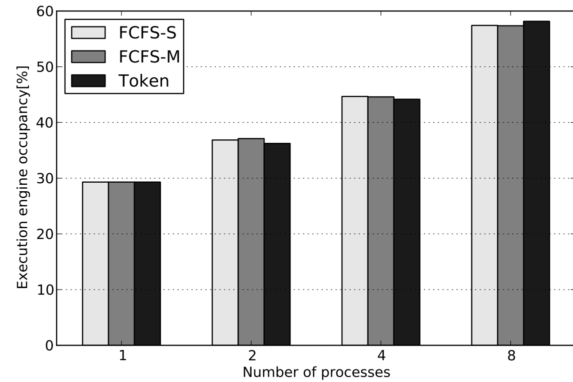


Fig. 9. Execution engine occupancy.

C. GPU Execution Analysis

A potential drawback of the design of *token* is that when a new kernel is scheduled, the logic reserves some GPU cores to schedule work groups of the new kernel. Once a GPU core becomes reserved, the scheduler logic needs to wait until all work groups being executed finish to start scheduling work groups for the new kernel. During this period of time, the GPU core is not fully utilized. This underutilization of the GPU hardware might degrade the overall system performance. To measure the impact of this effect, we analyze the occupancy of the GPU cores during the whole application execution.

Figure 9 shows the occupancy for *FCFS-S*, *FCFS-M*, and *token*. There is a variation of less than 0.1% among all three schedulers in all cases. These results show that the occupancy of the GPU cores is similar in all scheduling policies, so the effect of reserving GPU cores is negligible. This is because most benchmarks spawn a large number of work groups on each kernel invocation, so once a GPU core is assigned to a given kernel, a relatively large number of work groups are scheduled to run on it. As a result, the periods of time when GPU cores are underutilized due to reservations is very small compared to the total execution time.

We also study the effect of running multi-programmed workloads on the GPU DMA controller, which we do not show for brevity. DMA occupancy for data transfer from the CPU to the GPU increases as we increment the number of concurrent benchmarks. This is an expected result, since each benchmark requires sending its input data to the GPU memory prior calling the GPU kernels. Typically the input data for each kernel is quite large, and thus the occupancy of the downlink increases. This is in contrast to the uplink (i.e., DMA transfers from the GPU to the CPU), where the occupancy does not vary significantly as we increase the number of benchmarks being executed concurrently. This is due to the combination of two different effects. First, the output data from kernel invocations tends to be much smaller than the input data. Second, benchmarks take longer to execute when we increase the level of concurrency. As a result, the increment in the amount of data transferred from the GPU to the CPU gets compensated by the longer execution time, giving a mostly constant DMA uplink occupancy.

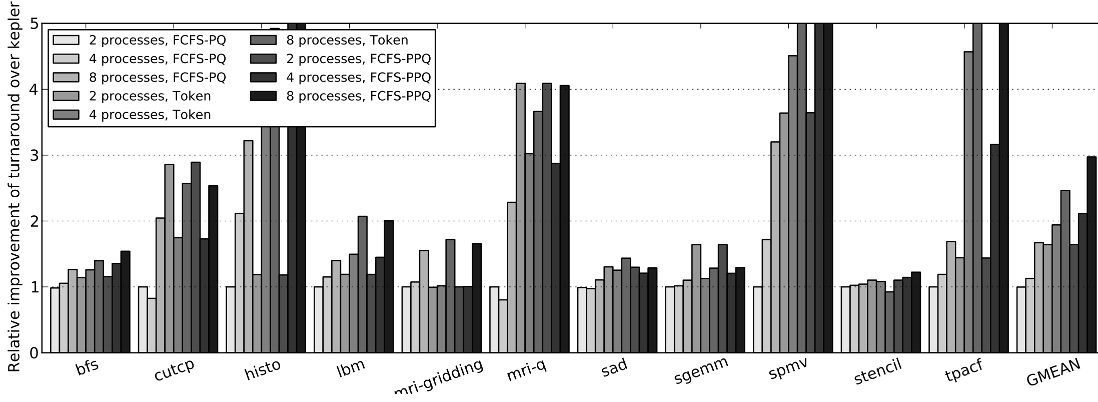


Fig. 10. Speedup of turnaround time of the high priority process over its nonprioritized execution.

D. *Enforcing Priorities*

System software uses priorities to favorize higher priority processes over the low priority ones, improves the response time of selected processes and allow soft real-time systems to achieve the given constraints. In section Section II, we illustrated on an example how being able to prioritize a task can be useful in GPU systems. Here, we evaluate the speedup when the application is prioritized compared to nonprioritized execution. We asign the high priority to one application per workload while others have equal lower priority. The token scheduler is setup so that the high priority application gets all the SMs, which means that other application fairly share the SMs when the high priority application is not running in the execution engine. In this experiment, we evaluate two additional schedulers. *Priority Queues (FCFS-PQ)* is the FCFS scheduler with one active context in the execution unit and different priorities assigned to each context. In this scheme, when the running kernel finishes its execution, a kernel from the context with the highest priority is scheduled to run in the GPU. This scheduling scheme could relatively easily be implemented on a multi-queue GPU architecture like Kepler. *Preemptive Priority Queues (FCFS-PPQ)* is also a FCFS scheduler but with multiple active contexts and preemptive kernel execution, implemented on top of the extensions proposed in the paper. As soon as the high priority kernel is launched, scheduler starts draining the SMs that execute lower priority kernels. Once the high priority kernel is executed, other, lower priority kernels can continue with the execution. Figure 10 shows the speedup of prioritized applications when using different scheduling schemes. As expected from the descriptions of the schedulers, all of them provide the improvement over FCFS-S on average. The simplest scheme with priorities (FCFS-PQ) provides the significant improvement for eight concurrent processes because only this case has relatively large number of on-flight kernels waiting in the queue. The token scheduler provides improvement even with the small number of applications because of its ability to preempt the low priority kernels once the high priority kernel is issued. However, in FCFS-PPQ in general provides the biggest speedup in turnaround time (up to 3x for eight processes) because it has the same ability to preempt the low priority running kernel but introduces the scheduling overhead smaller than the token. Note that the token scheduler, unlike the other FCFS based evaluated schedulers can also be setup so that it assigns the resources (SMs) proportional to the priorities given, which means that lower priority applications do not have to starve while the high priority application is running.

V. Related Work

Several researchers have noticed that sharing GPU resources among several applications could improve the system performance but current GPUs have limited sharing capabilities which usually limits the implementation of their proposal. Best to our knowledge, we are the first to propose mechanisms that allow transparent, fine grained GPU resource sharing and scheduling among multiple user processes.

To be able to better utilize available processing units in the existing GPUs, previous work has proposed using kernel fusion, a fusion technique that statically transforms the code of two, possible completely unrelated, kernels into one that is launched with the appropriate number of thread blocks. Fused kernel contains an *if* statement to checks which one of the original computations is to be performed. Because kernels use their thread and block *id* to find inputs and produce outputs, ids have to be recalculated to accommodate this scheme. Guevara et al. [10] proposed a runtime system for CUDA which chooses between running the fused kernel or running the kernels sequentially. Wang et al. [23] used a similar technique to reduce the energy consumption and improve power efficiency by achieving higher utilization of hardware resources. Gregg et al. [9] implemented an OpenCL scheduler that occupies the whole GPU with a scheduler kernel that dynamically invokes kernels to be executed.

Several approaches have been proposed for processes or applications to share a GPU. Pegasus [11] and the approach taken by Li et al. [15] introduce a virtualization layer. GERM [7] and TimeGraph [13] focus on graphics applications and provide a GPU command-driven scheduler integrated in the device driver. RGEM [12] is a runtime system that allows the preemption of memory transfers by splitting them into smaller chunks and thus creating the preemption points and provides separate queues to demultiplex data transfers and

computation. Gdev [14] is built around these principles, but it integrates the runtime support for GPUs into the OS to provide system level management of the GPU resources and GPU virtualization.

In [5], the authors make a case for spatial sharing of the GPU execution engine by simulating execution of several kernels from different applications in parallel on statically partitioned processing units among applications. Partitioning is chosen by the user and is performed at the beginning of the simulation, at the simultaneous launch of all benchmark applications, and cannot be changed after that. Furthermore, compute units are not reassigned to another application once the application executing on them completely finishes.

Ravi et al. [20] rely on the *molding* technique (changing the dimensions of grid and thread block while preserving the correctness of the computation) and propose a method to compute the potential improvements of co-scheduling to achieve dynamic spatial sharing of the GPU execution engine when native applications have such configurations that otherwise would not permit concurrent execution. Molding however, is not a generic technique, and can be only applied to a small number of existing kernels, without transforming the code. Pai et al. [19] propose a similar technique and associated code transformation based on iterative wrapping [21]. Their transformation produces an *elastic* kernel which can use whatever physical grid it was launched with to execute the original logical grid and is used to shape the grid in such a way that exploits the concurrent execution features of the underlying hardware. This transformation however suffers from performance issues due to the increased register usage and reduced number of thread blocks with a potential drop in throughput as the result.

Compared to all the previously published related work, our approach is the only one that is generic and transparent, and does not have the limitations of software-only approaches. Further more, our extensions allow actual implementation of some of the mechanisms and policies proposed previously.

VI. Conclusions and Future Work

In this paper we have proposed the first GPU architecture with support for multi-programmed workloads. This support allows efficient sharing of a single GPU among several user processes, enabling system software and application programmers to adapt the GPU behaviour to the characteristics of the workloads being executed, and maximize the GPU utilization by allowing several user processes to concurrently use the GPU resources.

We have also presented a novel kernel preemption technique that exploits the characteristics of the GPU programming model to efficiently hand off GPU resources from one user process to another one. This technique enables the implementation of fine grained schedulers in the GPU that allow concurrent execution and scheduling of multiple kernels on the GPU. We have explored the unique trade-offs of work scheduling on GPUs, where hardware resources are able to concurrently execute several work groups from the same GPU kernel.

The techniques and architectural support presented in this paper enable GPUs to be used in new computing environments, other than HPC and desktop systems. We plan to explore new GPU scheduling policies to bring soft and hard real time capabilities to GPUs. We also plan to study further hardware modifications to enable fair partitioning of shared resources other than computing logic, such as main memory bandwidth and shared cache memories.

References

[1] "Amd a-series processor-in-a-box," 2012. [Online]. Available: http://www.amd.com/us/products/desktop/processors/a-series/Pages/a-series-pib.aspx

[2] "Haswell microarchitecture," 2012. [Online]. Available: en.wikipedia.org/wiki/Haswell_(microarchitecture)

[3] "Namd - scalable molecular dynamics," 2012. [Online]. Available: http://www.ks.uiuc.edu/Research/namd/

[4] "Samsung exynos," 2012. [Online]. Available: www.samsung.com/exynos

[5] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for gpgpu spatial multitasking," in *HPCA 2012*, feb. 2012, pp. 1 –12.

[6] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *PPoPP 2012*. New York, NY, USA: ACM, 2012, pp. 23–34.

[7] M. Bautin, A. Dwarakinath, and T. Chiueh, "Graphic engine resource management," in *SPIE 2008*, vol. 6818, 2008, p. 68180O.

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

[9] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in *HotPar 2012*. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," 2009.

[11] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: coordinated scheduling for virtualized accelerator-based systems," in *USENIX 2011*. USENIX Association, 2011, pp. 3–3.

[12] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "Rgem: A responsive gpgpu execution model for runtime engines," in *RTSS 2011*, 29 2011-dec. 2 2011, pp. 57 –66.

[13] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX 2011*. USENIX Association, 2011, pp. 2–2.

[14] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: first-class gpu resource management in the operating system," in *USENIX 2012*. Berkeley, CA, USA: USENIX Association, 2012, pp. 37–37.

[15] T. Li, V. Narayana, E. El-Araby, and T. El-Ghazawi, "Gpu resource sharing and virtualization on high performance computing systems," in *ICPP 2011*, sept. 2011, pp. 733 –742.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[17] NVIDIA, "Next generation cuda computer architecture kepler gk110," 2012.

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[19] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *ASPLOS 2013*. ACM, 2013, p. 3x.

[20] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *HPDC 2011*. ACM, 2011, pp. 217–228.

[21] J. Stratton, S. Stone, and W.-m. Hwu, "Mcuda: An efficient implementation of cuda kernels for multi-core cpus," *LCPC 2008*, pp. 16–30, 2008.

[22] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading pentium 4 processor," in *PACT 2003*. IEEE, 2003, pp. 26–34.

[23] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *CPSCom 2012*, dec. 2010, pp. 344 –350.