Enabling Preemptive Multiprogramming on GPUs

Ivan Tanasic[1,2], Isaac Gelado[3], Javier Cabezas[1,2], Alex Ramirez[1,2], Nacho Navarro[1,2], Mateo Valero[1,2]
[1]Barcelona Supercomputing Center [2]Universitat Politecnica de Catalunya [3]NVIDIA Research
first.last@bsc.es nacho@ac.upc.edu igelado@nvidia.com

Abstract

GPUs are being increasingly adopted as compute accelerators in many domains, spanning environments from mobile systems to cloud computing. These systems are usually running multiple applications, from one or several users. However GPUs do not provide the support for resource sharing traditionally expected in these scenarios. Thus, such systems are unable to provide key multiprogrammed workload requirements, such as responsiveness, fairness or quality of service.

In this paper, we propose a set of hardware extensions that allow GPUs to efficiently support multiprogrammed GPU workloads. We argue for preemptive multitasking and design two preemption mechanisms that can be used to implement GPU scheduling policies. We extend the architecture to allow concurrent execution of GPU kernels from different user processes and implement a scheduling policy that dynamically distributes the GPU cores among concurrently running kernels, according to their priorities. We extend the NVIDIA GK110 (Kepler) like GPU architecture with our proposals and evaluate them on a set of multiprogrammed workloads with up to eight concurrent processes. Our proposals improve execution time of high-priority processes by 15.6x, the average application turnaround time between 1.5x to 2x, and system fairness up to 3.4x.

1. Introduction

Graphics Processing Units (GPUs) have become fully programmable massively parallel processors [21, 38, 4] that are able to efficiently execute both, traditional graphics workloads, and certain types of general purpose codes [26]. GPUs have been designed to maximize the performance of a single application, and thus assume exclusive access from a single process. Accesses to the GPU from different applications are serialized. However, as the number of applications ported to GPUs grows, sharing scenarios are starting to appear. Issues with GPU sharing, such as priority inversion and no fairness, have already been noticed by operating systems [30, 17, 18, 27] and real-time [16, 6] research communities. Moreover, with the integration of programmable GPUs into mobile SoCs [31, 5] and consumer CPUs [3, 15], the demand for GPU sharing is likely to increase. This leads us to believe that support for fine-grained sharing of GPUs must be implemented.

Today's GPUs contain execution and data transfer engines that receive commands from the CPU through command queues. The execution engine comprises all the GPU cores, that are treated as a whole. Commands from the same process targeting different engines can be executed concurrently (e.g., a data transfer can be performed in parallel with a GPU kernel execution). However, when a command is running, it has exclusive access to the engine and cannot be preempted (i.e., the command runs to completion). Hence, a long-latency GPU kernel can occupy the execution engine, preventing other kernels from same or different process to progress. This limitation hinders true multiprogramming in GPUs.

The latest NVIDIA GPU architecture, GK110 (Kepler), improves the concurrency of commands coming from the same process by providing several hardware command queues (often referred to as NVIDIA Hyper-Q [23]). NVIDIA also provides a software solution [24] that acts as a proxy to allow several processes to use the GPU as one, at the cost of losing process isolation. Combining these two features is especially useful for improving the utilization of GPU engines in legacy MPI applications. However they do not solve the problem of sharing the GPU among several applications.

To enable true sharing, GPUs need a hardware mechanism that can preempt the execution of GPU kernels, rather than waiting for the program to release the GPU. Such mechanism would enable system-level scheduling policies that can control the execution resources, in a similar way the multitasking operating systems do with the CPUs today. The assumed reason [1, 27] for the lack of a preemption mechanism in GPUs is the expected high overhead of saving and restoring the context of GPU cores (up to 256KB of register file and 48 KB of on-chip scratch-pad memory per GPU core), which can take up to $44\mu s$ in GK110, assuming the peak memory bandwidth. Compared to the context switch time of less than $1\mu s$ on modern CPUs, this might seem to be a prohibitively high overhead.

In this paper we show how preemptive multitasking is not only necessary, but also a feasible approach to multiprogramming on GPUs. We design two preemption mechanisms with different effectiveness and implementation costs. One is similar to the classic operating system preemption where the execution on GPU cores is stopped, and their context is saved to implement true preemptive multitasking. The other mechanism exploits the semantics of the GPU programming model and the nature of GPU applications to implement preemption by stopping the issue of new work to preempted GPU cores, and draining them from currently running work. We show that both mechanisms provide improvements in system

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

responsiveness and fairness at the expense of a small loss in throughput.

Still, exclusive access to the execution engine limits the possible sharing to time multiplexing. We propose further hardware extensions that remove the exclusive access constraint and allow the utilization of GPU cores, individually. These extensions enable different processes to concurrently execute GPU kernels on different sets of GPU cores. Furthermore, we implement Dynamic Spatial Sharing (DSS), a hardware scheduling policy that dynamically partitions the resources (GPU cores) and assigns them to different processes according to the priorities assigned by the OS.

The three main contributions of the paper are (1) the design of two preemption mechanisms that allow GPUs to implement scheduling policies, (2) extensions for concurrent execution of different processes on GPUs that allow implementing spatial sharing, and (3) a scheduling policy that dynamically assigns disjoint sets of GPU cores to different processes. Experimental evaluation shows that the hardware support for preemptive multi-tasking introduced in this paper allows scheduler implementations for multiprogrammed environments that, on average, improve the performance of high-priority applications up to 15.6x over the baseline at the cost of 12% of degradation in throughput. Our DSS scheduling policy improves normalized turnaround time up to 2x and system fairness up to 3.4x at the cost of throughput degradation up to 35%.

2. Background and Motivation

In this section we provide the background on GPU architecture and execution model that are necessary for understanding our proposals. Our base architecture is modeled after the NVIDIA GK110 chip, but we keep the discussion generic, to cover architectures from other vendors, as well as fused CPU-GPU architectures.

2.1. GPU Program Execution

Typically, GPU applications consist of repetitive bursts of 1) CPU execution, that perform control, preprocessing or I/O operations, 2) GPU execution (kernels), that performs computationally demanding tasks, and 3) data transfers between CPU and GPU, that bring input data to the GPU memory and return the outputs to the CPU memory. The GPU device driver is in charge of performing the bookkeeping tasks for the GPU, as the OS performs for the CPU (e.g., managing the GPU memory space). GPU kernel invocations (*kernel launch* in CUDA terminology), initiation of data transfers, and GPU memory allocations are typically performed in the CPU code (referred to as commands in the rest of the paper). Each kernel launch consists of a number of threads executing the same code. Threads are grouped in *thread blocks* that are independent of each other, and only threads from the same thread block can cooperate using barrier synchronization and communication through local memory (shared memory in CUDA terminology).

Programming models for GPUs provide software work queues (*streams* in CUDA terminology) that allow programmers to specify the dependences between commands. Commands in different streams are considered independent and may be executed concurrently by the hardware. Because the latency of issuing a command to the GPU is significant [17], commands are sent to the GPU as soon as possible. Each process that uses a GPU gets its own GPU *context*, which contains the page table of the GPU memory and the streams defined by the programmer.

To overcome the inefficiencies introduced by multiple processes sharing the GPU [20], NVIDIA provides a software solution called Muli-Process Service (MPS) [24]. MPS instantiates a proxy process that receives requests from client processes (e.g., processes in an MPI application) and executes them on the GPU. Such a solution has two main limitations: (1) memory allocated from any of the client processes, is accessible from any of the other client processes, thus breaking the memory isolation between processes; (2) it does not allow to enforce scheduling policies across the client processes. Although MPS provides important performance improvement in the case of MPI applications, it is not a general solution for multiprogrammed workloads.

2.2. Base GPU Architecture

The base architecture assumed in this paper is depicted in Figure 1. The GPU is connected to the rest of the system through an interconnection network (1). In the case of discrete GPUs, the interconnect is the PCI Express bus and the GPU has its own physical RAM (2). In the case of fused CPU/GPU architectures [8], the interconnect is an on-chip network and GPU and CPU share the same physical RAM (3). Current generation GPUs, including our baseline, do not support demand paging (memory swap-out). Thus, in today's GPU systems, allocations from all contexts reside in the GPU physical memory. The GPU has an execution engine (4) with access to its memory, and a data transfer engine (5) for transferring the data between CPU memory and GPU memory (in integrated designs GPUs have DMA engines for bulk memory transfers, too). All the GPU computation cores (*Streaming Multiprocessors* or SMs in CUDA terminology) belong to the execution engine and are treated as a whole, for kernel scheduling purposes.

The interface to the CPU implements several hardware queues (i.e., NVIDIA Hyper-Q) used by the CPU to issue GPU commands. The GPU device driver maps streams from the applications on the command queues. A *command dispatcher* (6) inspects the top of the command queues and *issues* the commands to the corresponding engine. Data transfer commands are issued to the data transfer engine via DMA queues while kernel launch commands are issued to the execution engine via the execution queue (7). After issuing a command, the dispatcher stops inspecting that queue. After completing a command, the corresponding engine notifies the

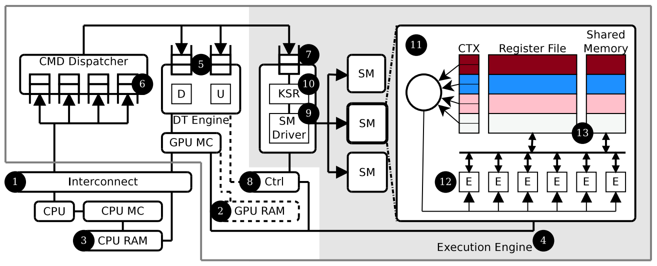


Figure 1: Baseline GPU architecture.

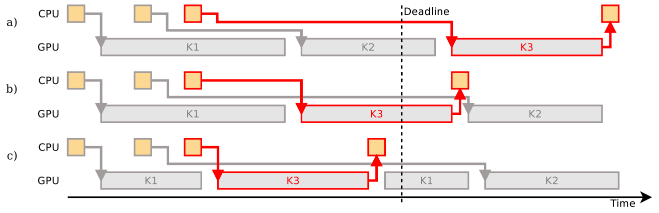


Figure 2: Execution of soft real-time application with (a) FCFS (current GPUs), (b) nonpreemptive priority and (c) preemptive priority schedulers. K1 and K2 are low-priority kernels, while K3 is high-priority.

command dispatcher so that the queue is re-enabled for inspection. Thus, commands from different command queues that target different engines can be concurrently executed. Conversely, commands coming from the same command queue are executed sequentially, following the semantics of the stream abstraction defined by the programming model. Traditionally, a single command queue was provided, but newer GPUs from NVIDIA provide several of them, often referred to as HyperQ [23]. Using several queues increases the opportunities to overlap independent commands that target different engines.

The GPU also includes a set of global control registers (8) that hold the GPU context information used by the engines. These 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 structures used by the graphics pipeline.

2.3. Base GPU Execution Engine

The base GPU *execution engine* we assume in this paper is shaded in Figure 1. The SM driver (9) gets kernel launch commands from the execution queue (7), and sets up the *Kernel Status Registers* (10) (KSR) with the control information such as number of work units to execute, kernel parameters (number and stack pointer)... The SM driver uses the contents of these registers, as well as the global GPU control registers, to setup the SMs before the execution of a kernel starts. The execution queue can contain a number of independent kernel commands coming from the same context, that are scheduled for execution in a first-come first-serve (FCFS) manner.

A kernel launch command consists of thread blocks that are independent from each other and, therefore, they are executed on SMs (11) independently. Each thread block is divided into fixed-size groups of threads that execute in a lock-step (*warps* in CUDA terminology) [21]. A reconvergence stack tracks the execution of divergent threads in a warp by storing the program counter and mask of the threads that took the divergent branch [11]. SM cycles through warps from all thread blocks assigned to it and execute (12) the next instruction of a warp with ready operands.

When a thread block is issued to an SM, it remains resident in that SM until all of its warps finish execution. An SM can execute more than one thread block in an interleaved fashion. Concurrent execution of thread blocks relies on static hardware partitioning, so the available hardware resources (13) (e.g., registers and shared memory) are split among all the thread blocks in the SM. The resource usage of all the thread blocks from a kernel is the same and it is known at kernel launch time. The number of thread blocks that can run concurrently is thus determined by the first fully used hardware resource. Static hardware partitioning implies that only thread blocks from the same kernel can be scheduled to concurrently run on the same SM.

After the setup of the SM is done, the SM driver issues thread blocks to each SM until they are fully utilized (i.e., run out of hardware resources). Whenever a SM finishes executing a thread block, the SM driver gets notified and issues a new thread block from the same kernel to that SM. This policy combined with static partitioning of hardware resources means that kernels with enough thread blocks from one kernel will occupy all the available SMs, forcing the other kernel commands in the execution queue to stall. As a result, concurrent execution among kernels is possible only if there are free resources after issuing all the work from previous kernels. This *back-to-back* execution happens when a kernel does not have enough thread blocks to occupy all SMs or the scheduled kernel is finishing its execution, and SMs are becoming free again. Today's GPUs, however, do not support concurrent execution of commands from different contexts on the same engine. That is, only kernels from the **same process** can be concurrently executed.

2.4. Arguments for Preemptive Execution

The main goal of this work is to enable fine-grained scheduling of multiprogrammed workloads running on the GPU. Figure 2 illustrates how scheduling support in current GPUs is not sufficient when, for example, a soft real-time application is competing for resources with other applications. The execution on a modern GPU is shown in Figure 2a, where the kernel with a deadline ($K3$) does not get scheduled until all previously issued kernels ($K1$ and $K2$) have finished executing. A software implementation [16] or a modification to GPU command scheduler could allow priorities to be assigned to processes, resulting in the timeline shown in Figure 2b.

A common characteristic of the previous approaches is that the execution latency of $K3$ depends on the execution time of previously launched kernels from other processes. This is an undesirable behavior from both system's and user's perspective, and limits the effectiveness of the GPU scheduler. To decouple the scheduling from the latency of kernel running on the GPU, a preemption mechanism is needed. Figure 2c illustrates how the latency of the kernel $K3$ could decrease even further if kernel $K1$ can be preempted. Allowing GPUs to be used for latency sensitive applications is the first motivation of this paper.

Preemptive execution on GPUs is not only useful to speed up high-priority tasks, it is required to guarantee forward progress of applications in multiprogrammed environments. The *persistent threads* pattern of GPU computing, for instance, uses kernels that occupy the GPU and actively wait for work to be submitted from the CPU [2, 14]. Preventing starvation when this kind of applications run in the multiprogrammed system is the second motivation of this paper.

There is a widespread assumption that preemption in GPUs is not cost-effective due to the large cost of context switching [1, 27]. Even though it is clear that in some cases it is necessary [19], it is not clear if benefits can justify the disadvantages when preemption is used by fine-grained schedulers. Comparing benefits and drawbacks of the context saving and restoring approach to preemption with an alternative approach where no context is saved or restored on preemption points is the third motivation of this paper.

3. Support for Multiprogramming in GPUs

Following the standard practice of systems design, we separate mechanisms from policies that use them. We provide two generic preemption mechanism and policies that are completely oblivious to the preemption mechanism used. To simplify the implementation of policies, we abstract the common hardware in a scheduling framework.

3.1. Concurrent Execution of Processes

To support multiprogramming, the memory hierarchy, the execution engine and the SMs all have to be aware of multiple active contexts. The memory hierarchy of the GPU needs to support concurrent accesses from different processes, using different address spaces. Modern GPUs implement two types of memory hierarchy [28]. In one, the shared levels of the memory hierarchy are accessed using virtual addresses and the address translation is performed in the memory controller. The cache lines and memory segments of such hierarchy have to be tagged with an address space identifier. The other implementation uses address translation at the private levels of the memory hierarchy, and physical memory addresses to access shared levels of the memory hierarchy. The mechanisms that we describe here are compatible with both approaches. We assume that later approach is implemented, hence no modifications are required to the memory subsystem.

If only one GPU context executes kernels, SMs can easily get the context information from the global GPU control structures. We extend the execution engine to include a *context table* with information of all active contexts. The context information is sent to the SM during the setup, before it starts receiving thread blocks to execute. The SM is extended with a *GPU context id register*, a *base page table register* and other context specific information, such as the *texture register*. The base page table register is used on a TLB miss to walk the per-process page table stored in the main memory of the GPU. This is in contrast to the base GPU architecture where the same page table was used by all SMs, since they execute kernels from the same context. Similarly, the GPU context id register is used when accessing the objects associated with the GPU context (e.g., kernels) from SM. We extend the context of the SM, rather than reading this information from the context table that would otherwise require many read ports to allow concurrent accesses from SMs.

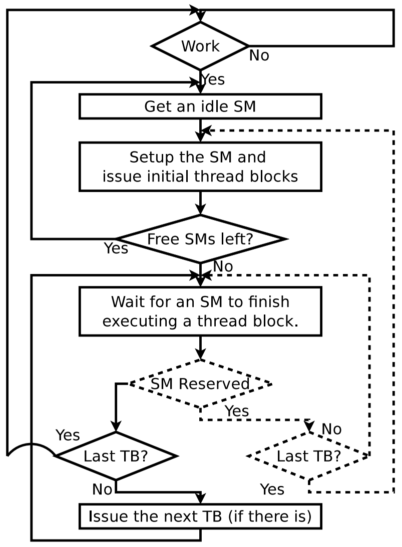

Work
No
Yes
Get an idle SM
Setup the SM and issue initial thread blocks
Free SMs left?
Yes
No
Wait for an SM to finish executing a thread block.
SM Reserved
Yes
No
Yes
Last TB?
No
Last TB?
Yes
Issue the next TB (if there is)

Figure 3: Operation of the SM driver. Dashed objects are proposed extensions.

3.2. Preemptive Kernel Execution

The scheduling policy is always in charge of figuring out when and which kernels should be scheduled to run. If there are no idle SMs in the system, it uses the preemption mechanism to free up some SMs. To provide a generic preemption support to different policies, we need to be able to preempt the execution on each SM individually. We provide this support by extending the SM driver. Figure 3 shows the operation of the SM driver, with dashed objects showing our extensions. When there are kernels to execute, the SM driver looks for an idle SM, performs the setup, and starts issuing thread blocks until the SM is fully occupied. The SM driver then repeats the procedure until there are no more idle SMs. When there are thread blocks left, the baseline SM driver issues a new thread block every time an SM notifies the driver that it finished executing a thread block.

We extend this operation and allow the scheduling policy to preempt the execution on an SM (independent of which preemption mechanism is used) by labeling it as *reserved*.

After receiving a notification of finished thread block from the SM, the SM driver checks if the SM is *reserved*. If not, it proceeds with the normal operation (issuing new thread blocks). If *reserved*, the driver waits for preemption to be done, sets up the SM for the kernel that reserved it, and continues with the normal operation. In Section 3.3 we describe the hardware extensions used by the SM driver to perform the bookkeeping of SMs and active kernels.

The first preemption mechanism that we implement, **context switch**, follows the basic principle of preemption used by operating system schedulers. The execution contexts of all the thread blocks running in the preempted SM are saved to off-chip memory, and these thread blocks are issued again, later on. Each active kernel has a preallocated memory where the context of its preempted thread blocks are kept. When a preempted thread block is issued, its execution context is first restored so the computation can continue correctly. This context consists of the architectural registers used by each thread of the thread block, the private partition of the shared memory, and other state that defines the execution of the thread block (e.g., the pointer to the reconvergence stack and state of the barrier unit). Saving and restoring the context is performed by a microprogrammed trap routine. Each thread saves all of its registers, while the shared memory of the thread block is collaboratively saved by its threads. This operation is very similar to the context save and restore performed on device-side kernel launch when using the dynamic parallelism feature of GK110 [25]. Since preemption raises an asynchronous trap, precise exceptions are needed [32]. The simplest solution is to drain the pipeline from all the on-flight instruction before jumping to the trap routine. The main drawback of the context switch mechanism is that during the context save and restore, thread blocks do not progress, leading to a complete underutilization of the SM. This underutilization could be improved by using compiler-microarchitecture co-designed context minimization techniques, such as iGPU [22].

The second mechanism that we implement, **SM draining**, tries to avoid this underutilization by preempting the execution on a thread block boundary (i.e., when a thread block finishes execution). Since thread blocks are independent and each one has its own state, no context has to be saved nor restored this way. This mechanism deals with the interleaved execution of multiple thread blocks in an SM by draining the whole SM when the preemption happens. To perform the preemption by draining, the SM driver stops issuing any new thread blocks to the given SM. When all the thread blocks issued to that SM finish, the execution on that SM is preempted.

The context switch mechanism has a relatively predictable latency that mainly depends on the amount of data that has to be moved from the SM (register file and shared memory) to the off-chip memory. The draining mechanism, on the other hand, tries to trade the predictable latency for higher utilization of the SM. Its latency depends on the execution time of currently running thread blocks, but SMs still get to do some useful work while draining. The draining mechanism naturally fits the current GPU architectures as it only requires small modifications to the SM driver. The biggest drawback is its inability to effectively preempt the execution of applications with very long running thread blocks or even preempt the execution of malicious or persistent kernels at all.

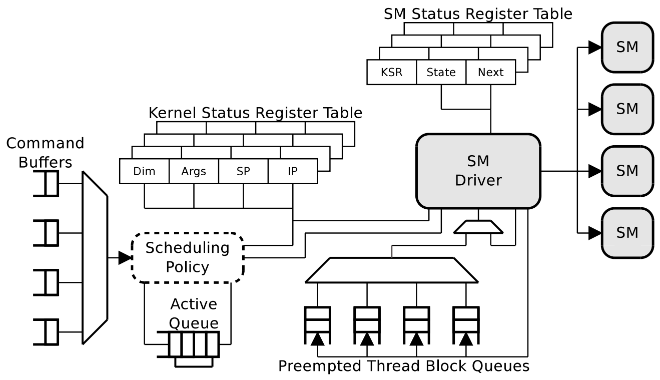


Figure 4: Scheduling framework. The rest of the execution engine (SM Driver and SMs) is shaded.

3.3. Scheduling Framework

We extract a generic set of functionalities into a scheduling framework that can be used to implement different scheduling policies. The framework provides the means to track the state of kernels and SMs and to allow the scheduling policy to trigger the preemption of any SM. The scheduling policy plugs into the framework and implements the logic of the concrete scheduling algorithm. Both the scheduling framework and scheduling policies are implement in hardware to avoid the long latency of issuing commands to the GPU [17]. Both the context switch and draining preemption mechanisms are supported by our framework. Scheduling policies performing prioritization, time multiplexing, spatial sharing or some combination of these can be implemented on top of it. The OS can tweak the priorities on the flight, but is not directly involved in the scheduling process. Thus, there is no impact on the OS noise.

Figure 4 shows the components of the scheduling framework. An example of interaction between the scheduling policy and the framework is given in Section 3.4. **Command Buffers** receive the commands from the command dispatcher and separate the execution commands from different contexts. Each command buffer can store one command. The **Active Queue** stores the identifiers of the active (running or preempted) kernels. When there are free entires in the active queue, the scheduling policy can read a command (kernel launch) from one of the command buffers and allocate an entry in the **Kernel Status Register Table (KSRT)**. KSRT is used to track active kernels and each valid entry is a KSR of one active kernel, augmented with the identifier of its GPU context. The active queue is used by the policy to search for scheduling candidates by indexing the KSRT. The **SM**

Status Table (SMST) is used to track the SMs. Each entry in the SMST contains the KSR index of the kernel being executed, the state of the SM (*Idle*, *running* or *reserved*), the number of running thread blocks, and the KSR index of the next kernel (when in *reserved* state). The SMST is accessed by the SM driver when issuing a thread block to find the KSR and the state of the SM. When setting up the SM that was *reserved*, the SM driver uses the *next* field of the SMST to find the kernel that the SM was reserved for. The **Preempted Thread Block Queues (PTBQ)** are used to store the handlers of preempted thread blocks. Each queue is associated with one KSR and its entries contain the id and stack pointer of a preempted thread block. Each time the driver is about to issue a new thread block from a kernel, it checks if there are any previously preempted thread blocks from that kernel. If there are, the thread block from the top of the associated PTBQ will be issued. Otherwise, the next thread block will be issued. We choose to issue preempted thread blocks first in order to keep their number limited, thus allowing their handlers to be stored on-chip, for quick access. Still, we allow all active thread blocks of a kernel to be preempted, not to limit the type of sharing that the scheduling policy can implement. Notice that the draining mechanism does not need PTBQs, as thread blocks run to completion.

In our implementation we limit the number of active kernels to the number of SMs, to allow spatial partitioning granularity of one kernel per SM. This also allows us to keep the PTBQs on chip, for fast access. Thus SMST, KSRT and the active queue all have N_{SMs} (the number of SMs) entries. There is also N_{SMs} PTBQs, each one with $N_{SMs} * T_{max}$ entries, where T_{max} is the maximum number of active thread blocks in an SM. This number of active kernels is also adequate for time multiplexing in large GPUs, but it could be changed (e.g., to allow time multiplexing in mobile GPUs with one SM), since the mechanisms and the policy described in Section 3.4 can also support different ratios of active kernels to SMs. Fixing the number of active kernels means that when active queue is full, new kernels submitted to the GPU will not be considered for scheduling until one of the active kernels finishes.

Hardware overheads of the framework are minor for our baseline architecture. Command buffers, KSRT, SMST, and active queue together take less than 0.5KB of on-chip SRAM. PTBQs take 21KB of on-chip SRAM and are present only if the context switch mechanism is implemented.

3.4. Dynamic Spatial Sharing Policy

Here we present the Dynamic Spatial Sharing (DSS) policy that is designed to perform dynamic spatial partitioning of the execution engine by assigning disjoint sets of SMs to different kernels. The DSS policy is based on the concept of tokens that represent the ownership of the resource (SM in this case). It allows the OS, runtime system or a user to assign a number of tokens to each kernel that represents their SM budget. One token is taken from the kernel (by decrementing its token count) when an SM is assigned to it. When an SM gets deassigned from the kernel, due to the preemption or the kernel finishing execution, the token is returned to it. To prevent the underutilization of resources that would happen when there are more SMs than tokens are assigned, kernels are allowed to occupy more SMs, by going to debt (have a negative token count). The DSS policy is formally given in Algorithm 1.

Algorithm 1 Partitioning Algorithm

```
function PARTITIONING_PROCEDURE
    repeat
        $idle\_sm \leftarrow find\_idle(SMSR.state)$
        $ksr\_max \leftarrow max(KSR.count)$
        $ksr\_min \leftarrow min(KSR.count)$
        if $KSR[ksr\_max] = KSR[ksr\_min]$ then
            return
        end if
        if $idle\_sm$ then
            $SMSR[idle\_sm].state \leftarrow running$
            $KSR[ksr\_max].count \leftarrow KSR[ksr\_max].count - 1$
        else
            $SMSR[ksr\_min].state \leftarrow reserved$
            $SMSR[ksr\_min].next \leftarrow ksr\_max$
            $KSR[ksr\_min].count \leftarrow KSR[ksr\_min].count + 1$
            $KSR[ksr\_max].count \leftarrow KSR[ksr\_max].count - 1$
        end if
    until $KSR[ksr\_max].count \leq KSR[ksr\_min].count + 1$
end function
```

The scheduling algorithm can be invoked by a periodic timer or by some events occurring in the system. We choose to execute the algorithm only on the following events: (1) a kernel is inserted in the active queue (increase in the number of active kernels) and (2) an SM becomes idle (increase in number of idle SMs). The logic that implements the policy finds the kernel with the highest token count (that has thread blocks to issue), the kernel with the lowest token count, and checks if there are any *Idle* SMs in the system. If these two kernels have the same number of tokens, no repartitioning is performed. If there are idle SMs, the token count is decremented, and the kernel is scheduled to execute on that SM. Otherwise, the policy finds the running kernel with the lowest current token count and switches the state of one of its assigned SMs from *running* to *reserved*, triggering the kernel preemption on that SM. It also increments the token count of the preempted kernel and decrements the token count of the newly assigned kernel. This procedure is repeated until the difference between the current token counts of all the active kernels is no bigger than one (to prevent a livelock) at which point the system gets into the steady state. We allow the scheduler to change the kernel for which an SM is reserved during the preemption of that SM. This optimization helps to cope with dynamic nature of the system and long latency operations.

The policy uses the contents of the SMST, KSRT and the active queue to partition the available SMs among the running kernels. Searching for the kernels with the biggest and smallest amount of tokens and searching for the idle SM can all happen in parallel. Since the operation is not on the critical path, we perform the search serially, by one counter going through the

Benchmark & Dataset	Kernel	Num. of Launches	Avg. Time (μs)	Num. TBs	Time/TB (μs)	Sh. M. /TB (B)	# Regs /TB	TBs /SM	Resour. /SM (%)	Save Time (μs)	Class 1	Class 2
lbm [short]	StreamCollide	100	2905.81	18000	2.42	0	4320	15	83.26	16.20	MEDIUM	LONG
histo [default]	final	20	70.24	42	5.02	0	19456	3	75.00	14.59	SHORT	MEDIUM
	prescan	20	20.87	64	1.30	4096	9216	4	52.63	10.24		
	intermediates	20	77.88	65	4.79	0	8964	4	46.07	8.96		
	main	20	372.58	84	4.44	24576	16896	1	29.61	5.76		
tpacf [small]	genhists	1	14615.33	201	72.71	13312	7680	1	14.14	2.75	LONG	MEDIUM
spmv [medium]	spmvjds	50	42.38	374	1.81	0	928	16	19.08	3.71	SHORT	SHORT
mri-q [large]	ComputeQ	2	3389.71	1024	26.48	0	5376	8	55.26	10.75	MEDIUM	SHORT
	ComputePhiMag	1	4.70	4	4.70	0	6144	4	31.58	6.14		
sad [large]	largersadcalc8	1	8174.21	8040	16.27	0	3328	16	68.42	13.31	LONG	LONG
	largersadcalc16	1	1529.38	8040	3.04	0	832	16	17.11	3.33		
	mbsadcalc	1	15446.02	128640	0.84	2224	2135	7	24.20	4.71		
sgemm [medium]	mysgemmNT	1	3717.18	528	98.56	512	4480	14	82.89	16.13	MEDIUM	SHORT
stencil [default]	block2Dregtiling	100	2227.30	256	8.70	0	41984	1	53.95	10.50	MEDIUM	LONG
cutcp [small]	lattice6overlap	11	1520.11	121	37.69	4116	3328	3	16.80	3.27	MEDIUM	MEDIUM
mri-gridding [small]	binning	1	2021.41	5188	1.56	0	4096	4	21.05	4.10	LONG	LONG
	scaninter1	9	7.59	29	4.14	665	1173	16	27.54	5.36		
	scanL1	8	826.12	2084	1.19	4368	9216	3	39.74	7.73		
	uniformAdd	8	127.30	2084	0.24	16	4096	4	21.07	4.10		
	reorder	1	2535.30	5188	1.95	0	8192	4	42.11	8.19		
	splitSort	7	3838.84	2594	4.44	4484	10240	3	43.79	8.52		
	griddingGPU	1	208398.47	65536	31.80	1536	3648	10	51.81	10.08		
	splitRearrange	7	1622.93	2594	1.88	4160	5888	3	26.71	5.20		
	scaninter2	9	8.81	29	4.80	665	1173	16	27.54	5.36		

Table 1: Statistics of all the kernels from benchmark applications used in the experimental evaluation.

CPU	GPU
Clock: 2.8 GHz	Clock: 706 MHz
Cores: 4	Cores: 13 (32 pipelines each)
Threading: 2-way	Memory Bandwidth: 208 GB/s
PCIe Bus	Registers (per SM): 65536
Clock: 500 MHz	Thread Blocks (per SM): 16
Lanes: 32	Threads (per SM): 2048
Burst: 4 KB	Shared memory (per SM): 16KB*/ 32KB / 48KB

Table 2: Simulation parameters used in the experimental evaluation. *Default configuration of the shared memory

SMST and KRST. This operation takes as many cycles as SMs (13 in our configuration).

4. Experimental Evaluation

4.1. Methodology

We evaluate our proposals using an in-house trace-driven simulator, based on the methodology of [9], that models a multi-core CPU connected to a discrete GPU through a PCIe bus. The simulator performs a coarse-grained modeling of the CPU, tracing the execution of our benchmarks on an Intel Core i7 930 chip. CPU traces consist of a starting and ending timestamp for each API call to the CUDA runtime library. We perform an accurate simulation of the PCIe bus and the GPU using execution traces of each GPU kernel obtained on the NVIDIA Kepler K20c discrete GPU. The simulator parameters are provided in Table 2[1]. All the benchmark applications are traced from the first CUDA call to the last CUDA call, capturing all the memory transfer, kernel execution and CPU execution phases. This approach provides more realistic execution workloads, as opposed to running kernels only.

We use ten, out of eleven, benchmarks from the Parboil benchmark suite [34] in our evaluation. We do not use the *BFS* benchmark in our evaluation since it uses the global synchronization that our trace-driven infrastructure cannot model accurately. Table 1 shows the characteristics of the benchmark applications. For each kernel we show the number of launches, the execution time of the kernel, the number of thread blocks, the average execution time of the thread blocks, the shared memory usage of each thread block, the number of registers used by each thread block, the maximal number of concurrent thread blocks in an SM, the amount of on-chip SRAM (shared memory and register file) resource utilization of an SM, and the projected context save time when preempting an SM (assuming only its share of global memory bandwidth). All the kernels are compiled for the NVIDIA compute architecture 3.5 (native for the Kepler GK110 chips) using the NVCC version 5.0 and GCC 4.6.3 for the host code.

We create multiprogrammed workloads by co-scheduling several benchmark applications chosen randomly. We run all benchmarks in the workload, replaying them once they complete until all benchmarks have been executed at least 3 times. Replaying shorter benchmarks provides even workload for the longer benchmarks. Replaying even the longest benchmarks provides different workload interleavings. We choose the input sets of the benchmarks (shown in square brackets in Table 1) in a way that minimizes the extreme differences in the execution times of the benchmarks and thus cut back on our simulation time. Still, there is plenty of variability between benchmarks. Statistics are gathered only for the completed executions and then averaged. This methodology is based on [36] and [37].

All the metrics used in our evaluation are calculated as suggested by Eyerman et al. [10]. Metrics are calculated based on the performance (execution time) of applications run in isolation and run in the multiprogrammed workload.

[1]If the default configuration does not allow the kernel to be launched because it needs more shared memory, the SM will be configured for the first bigger configuration that satisfies the shared memory requirement

- *Normalized Turnaround Time* (NTT) is the measure of application slowdown when executed as part of the multiprogrammed workload, compared to the isolated execution.
- *Average Normalized Turnaround Time* (ANTT) is calculated as the arithmetic average of turnaround times of all applications in a workload, normalized to their isolated execution.
- *System Throughput* (STP) is the measure of system's overall performance and expresses the amount of work done in a unit of time.
- *Fairness* is the measure (number between one and zero) of equal progress of applications in a multiprogrammed workload, relative to their isolated execution, and it ranges between perfect fairness (all the processes experience equal slowdown over isolated execution) and no fairness at all (some processes completely starve).

4.2. Effectiveness of the Preemption Mechanisms

To measure the performance of the mechanisms, isolated from the potential benefits and overheads of the scheduling policies implemented on top of them, we evaluate the mechanisms by implementing the simple *priority queues* scheduler. This scheduler (used in our example in Section 2.4) always schedules the kernel with the highest priority. We quantify the benefits of preemptive execution and compare the performance of the two described preemption mechanisms by comparing the priority queues schedulers that implement preemption (preemptive priority queues-PPQ) and the implementation of priority queues scheduler with no preemption (NPQ). The scheduling policy in the data transfer engine is NPQ, in all cases. We generate random workloads in which one process has higher priority than the rest of the processes in the workload. All the benchmark applications appear the same number of times as the high-priority process.

We measure the turnaround time of the prioritized application and in Figure 5 show the improvement of the application's NTT when prioritized over its nonprioritized execution. When using the nonpreemptive prioritization scheme, turnaround time improves for workloads with 4 or more processes (from 1.1x to 1.6x on average as the number of processes grows). The NPQ scheduler allows the high-priority application to start executing as soon as SMs become available, thus the high-priority kernel has to wait only for the currently running kernel to finish. The nonpreemptive scheduler does not bring any improvement for workloads with only 2 processes since in this case the scheduler actually never has any choice. The only potential scenario that improves the turnaround time over the FCFS is if a kernel from the high-priority process and a kernel from the low-priority process are both launched while the execution engine is already running another kernel of the high-priority process. The newly launched high-priority kernel has to wait in the command dispatcher until the previously launched kernel finishes. By the time it reaches the execution engine, the ready, low-priority kernel will already be scheduled.

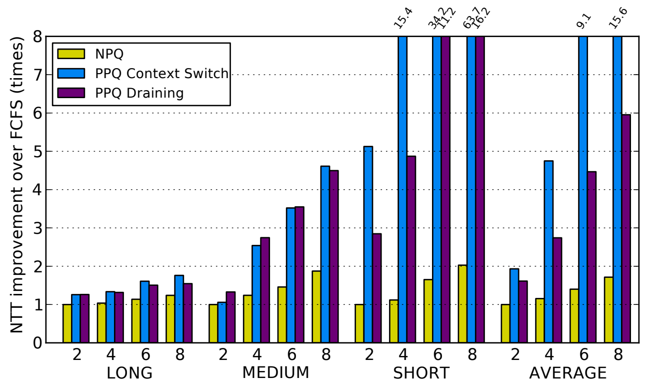
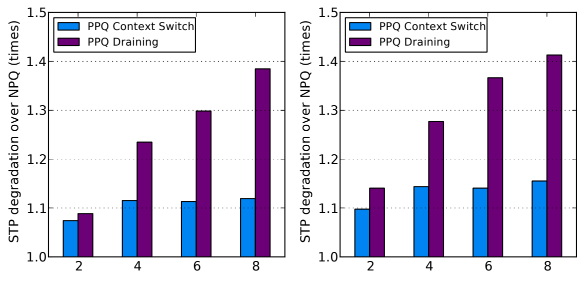


Figure 5: Turnaround time improvement of the high-priority process over its nonprioritized execution. Benchmarks in each group are listed in Table 1 as *Class 1*.

Preemptive priority queues (PPQ) scheduler shows a much higher turnaround time improvement of the high-priority process, since the high-priority kernels do not have to wait for the whole low-priority kernel to finish, just the usually much shorter preemption latency. Both preemption mechanisms improve turnaround time over the NPQ scheduler, but using the context switch mechanism the improvements, on average, are much higher (from 2x to 15.6x as number of processes grows) than the draining mechanism (from 1.6x to 6x as number of processes grows). This difference comes from the, on-average, lower preemption latency of the context switch mechanism. Only 8 of 24 kernels from our benchmark suite use more than half of the available storage resources (register file and shared memory), resulting in a longest projected context save time of 16.2μs (*StreamCollide* kernel from *lbm*). On the other hand, 6 kernels have an average thread block execution time longer than 16.2μs, with the longest one being 98.56μs (*mysgemmNT* from *sgemm*). Since the thread block execution time dictates the preemption latency when using the draining mechanism, the latency of preempting is on average smaller when using context switch.

The PPQ scheduler has variable effectiveness for different benchmark applications. In Figure 5, benchmarks are grouped by the average execution time of their kernels (Class 1 in Table 1). Both groups and execution times are listed in Table 1. Three benchmarks, from the *LONG* group, have at least one very long kernel (> 10000 μs). They observe the smallest improvement in performance (from 1.26x to 1.76x with context switch and 1.54x with the draining mechanism, as the number of processes in the workload grows) since their kernels dominate the execution. The improvements that they achieve mainly come from the workloads where they are mixed with other benchmarks from the *LONG* group. Half of the benchmarks (five of them), averaged in the *MEDIUM* group have at least one medium kernel (between 1000 μs and 3500 μs). They achieve bigger improvements (from 1.06x to 4.6x with the context switch and from 1.33x to 4.5x with the draining



(a) Exclusive access. (b) Shared access.

Figure 6: System throughput (STP) when the prioritized kernel has exclusive and shared access to the execution engine.

mechanism). The remaining two benchmarks, averaged in the *SHORT* group have only short kernels (< 350 μs). They observe very big improvements in turnaround time (from 5.1x to 63.7x with the context switch and from 2.84x to 16.2x with the draining mechanism) since the execution times of their kernels are very short compared to the other benchmarks. Preemption thus minimizes the waiting time of these kernels significantly. The benefits of the context switch mechanism accumulate with every preemption of the kernels with long running thread blocks, resulting in a big difference in the effectiveness of the two mechanisms, especially in the *SHORT* group. The shorter the kernels are, the more time will they be launched (because of the replay of benchmarks described in Section 4.1), increasing the chance of preempting a kernel with very long thread blocks.

4.3. Overheads of the Preemption Mechanisms

The preemption mechanisms on average improve the turnaround time of the high-priority process, but they come at the price of a lower utilization of the execution engine. The degradation in the STP due to the preemption mechanisms is quantified in Figure 6. We implement two slightly different PPQ schemes. The first scheme, shown in Figure 6a grants the high-priority process an exclusive access to the execution engine. Even if some resources become available, low-priority kernels will not be scheduled while high-priority kernels are still active. On average, PPQ with the context save mechanism has 1.08x to 1.12x STP overhead over NPQ while PPQ with the draining mechanism has an STP overhead between 1.09x and 1.38x. The bigger overhead of the draining mechanism comes from preemptions of kernels that can execute many (long) thread blocks concurrently. The more tread blocks per SM a kernel can run, the bigger is the chance that the variable execution times of the thread blocks will leave the SM running underutilized (i.e., running a number of thread blocks lower than its actual capacity).

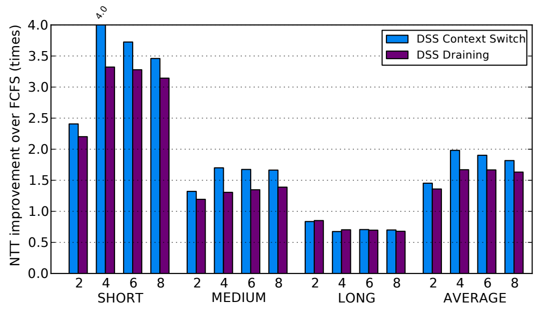
The other PPQ scheme that we implement uses the free resources to schedule low-priority kernels, even in the presence of high-priority kernels in the execution engine. It is modeled after current GPUs that try to perform back-to-back scheduling of the independent kernels (from the same process) to improve the STP. Such a technique works with the simple FCFS policy, but it is counterproductive in the case of preemptive prioritization, since some applications tend to asynchronously enqueue many kernel invocations. The back-to-back execution, described in Section 2.3, allows a low-priority kernel to start executing as soon as some SMs become free. These kernels get preempted soon after they start executing and actually waste resources, instead of saving any. Hence, this scheme, shown in Figure 6b, results in higher overheads than the exclusive-access one, from Figure 6a.

4.4. Example Policy: Equal Spatial Sharing

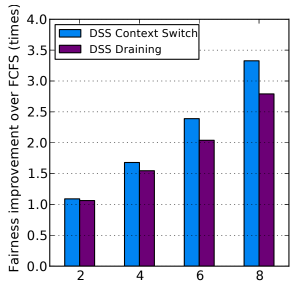
We use the DSS scheduling policy described in Section 3.4 to allow all active kernels to run concurrently. By ensuring that all the active kernels progress, the policy seeks to prevent the starvation of processes with short kernels and at the same time fairly partition the resources among all running kernels. We setup the DSS policy to perform equal sharing by assigning equal priorities (token count) to all the processes ($t_c = \lfloor N_{sm}/N_p \rfloor$). Since there is thirteen SMs in our simulated systems, but we evaluate with 2, 4, 6 and 8 process workloads, not all processes actually get the same number of SMs. The rest of the SMs that cannot be evenly distributed ($r = N_{sm} \ mod \ N_p$) are assigned to the r kernels that first reach the active queue. We use both draining and context switch mechanisms to evaluate this policy. The scheduling policy in the data transfer engine is FCFS, in all cases.

We first analyze the effects on the NTT of each benchmark application in all workloads and show their average improvements in Figure 7a. Benchmarks applications are grouped by their execution time (Class 2 in Table 1). Short applications (<5ms), averaged in Figure 7a as *SHORT*, achieve the biggest improvement in their turnaround time (2.45x to 4x with the context switch and 2.2x to 3.7x with the draining mechanism, as the number of processes in the workload grows), since their waiting time is lowered by spatially sharing the SMs. Medium long ones (between 30ms and 115ms), averaged as *MEDIUM*, achieve a significant improvement in their turnaround time (1.3x to 1.7x with the context switch and 1.2x to 1.4x with the draining mechanism). The improvements in both *SHORT* and *MEDIUM* classes come at the expense of very long (>400ms) applications, averaged as *LONG*, that get their turnaround time degraded (from around 0.9x to 0.55x with both mechanisms). On average, DSS improves the normalized turnaround time compared to FCFS with both preemption mechanisms. The improvement is bigger when using the context switch (from 1.5x to 2x) compared to the draining mechanism (from 1.4x to 1.65x).

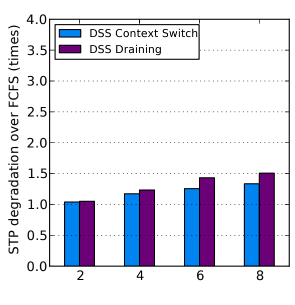
Figure 8 shows the ANTT achieved with the FCFS and DSS policies with both context switch and draining mechanisms for each workload. For workloads with 2 processes, equal sharing improves the ANTT significantly for about 20% of the simulated workloads. In the other 80% of workloads,



(a) Turnaround time improvement



(b) System fairness improvement



(c) System throughput degradation

Figure 7: Effects of equal sharing on turnaround time, fairness and system throughput. The list of benchmarks in each group of Figure 7a is given in Table 1 as *Class 2*.

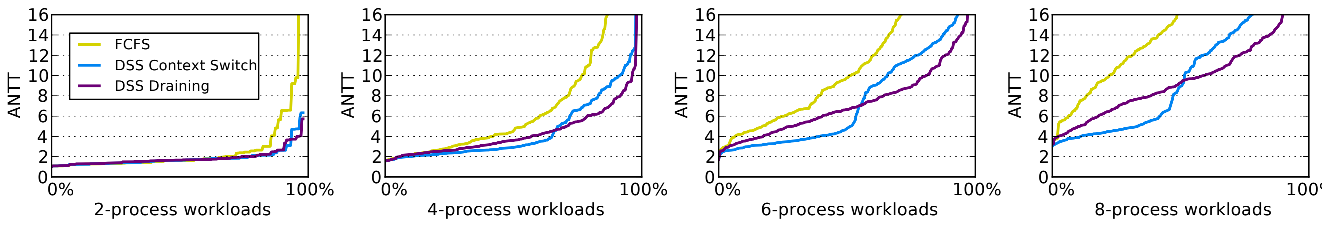


Figure 8: Average Normalized Turnaround Time (ANTT) for all the simulated workloads.

there is not much to improve because the interleaved execution phases of the benchmark applications and application's ability to partially tolerate latency by using asynchronous GPU commands keep ANTT low. The percentage of workloads with improved ANTT grows with the number of processes in the workload (70% for 4 processes), to almost all workloads (6 and 8 processes) having improved ANTT over the baseline FCFS scheduler with both preemption mechanisms.

Workloads with 4, 6, and 8 processes also show a clear cross point, after witch the policy implemented with the draining mechanism starts showing lower ANTT than the policy implemented with the context switch mechanism. In all configurations, this point is around half of the workloads that improve the ANTT over the FCFS. The crossing point appears because the two preemption mechanisms have different effects on different kernels. Contrary to the kernels with very long thread block execution times (discussed in the Setcion 4.2), some kernels have a context switch time much larger than their average thread block execution time (all of the kernels from *histo*, *StreamCollide* from *lbm*, *mbsadcalc* from *sad*, *reorder* from *mri-gridding*...). Even though DSS with the context switch mechanism achieves better average NTT, these results show that, depending on the workload at hand, the draining mechanism can also be a viable option for low latency preemption.

With equal sharing of resources, this scheduler also aims at improving the fairness among the processes. We show the relative improvement of the fairness of the DSS policy compared to the baseline FCFS policy in Figure 7b. The FCFS scheduler does not aim at optimizing the fairness, but does not cause complete starvation in our experiments because there are no persistent kernels in our benchmarks. Compared to it, the DSS policy achieves better fairness with both preemption mechanisms, thanks to its semi-equal resource allocation. The improvement is higher when using context switch (from 1.1x to 3.35x as the number of processes grows) compared to the draining mechanism (from 1.05x to 2.7x) thanks to the lower latency of preemption, as discussed in Section 4.2. Like with the ANTT in Figure 8, fairness is not improved much in the workloads with 2 processes.

Equally sharing the execution unit on the other hand achieves lower STP, mainly due to the lower utilization caused by the execution preemptions. The effects of preempting are quantified with the STP degradation, illustrated in Figure 7c. The average STP degradation compared to the FCFS scheduler is lower when using context switch (1.06x to 1.34x as the number of processes grows) compared to the draining mechanism (1.08x to 1.5x). Even though, intuitively, one might expect the context switch mechanism to achieve the lower STP than the draining mechanism, this is not the case. Analyzing the throughput of individual workloads (not shown because of the space constraint), a crosspoint similar to the one in Figure 8 can be observed. This time, however, the improvements in STP with the draining mechanism are negligible, while improvements with the context switch mechanism are significant.

Comparing the improved average normalized turnaround time and the system fairness (especially in the case of the context switch mechanism) to the degradation of STP shows that the preemptive equal sharing policy is a viable option when a little bit of overall system performance (STP) could be spared to the user perceived performance (application turnaround time) or system fairness. We thus believe that the equal sharing policy would be a good candidate for deployment in single-user multiprogrammed environments such as desktop or mobile systems, as well as multi-tenant cloud or server nodes.

5. Related Work

Li et al. [20] introduced a virtualization layer that makes all the participating processes execute kernels in the same GPU context, similar to NVIDIA MPS [24]. GERM [7] and TimeGraph [17] focus on graphics applications and provide a GPU command schedulers integrated in the device driver. RGEM [16] is a software runtime library targeted at providing responsiveness to prioritized CUDA applications by scheduling DMA transfers and kernel invocations. RGEM implements memory transfers as a series of smaller transfers and thus create the potential preemption points, lowering the stall time due to the competing memory transfers. Gdev [18] is built around these principles, but integrates the runtime support for GPUs into the OS. PTask [30] is another approach on making the OS GPU aware by using a task based data flow programming model and exposing the task graph to the OS kernel. Since the GPU execution engine does not expose its internals to software, none of these systems can control assignment of SMs to different kernels or implement preemptive scheduling policies.

Several software techniques have been proposed in the past to increase the concurrency between different kernels running in the GPU. Kernel fusion is a technique that statically transforms the code of two kernels into one that is launched with the appropriate number of thread blocks. Guevara et al. [13] proposed a runtime system for CUDA which chooses between running the fused kernel or running the kernels sequentially. Gregg et al. [12] implemented an OpenCL kernel that occupies the whole GPU and dynamically invokes kernels to be executed. Alternatively, several techniques underutilize the GPU by one application, so that concurrent execution could be achieved. Ravi et al. [29] rely on the *molding* technique (changing the dimensions of grid and thread blocks while preserving the correctness of the computation), when possible. Pai et al. [27] propose a similar technique and associated code transformation based on iterative wrapping [35] that produces an *elastic* kernel. These techniques rely on developer or compiler transformation to prepare the programs for concurrent execution.

Similarly, several software techniques have been proposed to implement time multiplexing on GPUs. The *kernel slicing* technique used in [6, 27, 39] launches the transformed kernel several times, passing the launch offset so that a slice performs only a part of the original computation. Softshell [33] programming model relies on developers explicitly declaring preemption points or preempting on the thread block boundary. Both kernel slicing and Softshell's preemption on the thread block boundary are built on the same principle as our draining mechanism, but incur extra overheads of doing it in software.

In [1], the authors make a case for spatial sharing of the GPU execution engine by simulating execution of several kernels from different applications running in parallel. They statically partition SMs among applications, since the emphasis of their work is on showing the benefits of spatial multitasking, rather then proposing the mechanisms to implement it. In contrast, we provide mechanisms that, among other things, allow implementing spatial sharing.

6. Conclusions

Current GPUs do not provide the necessary mechanisms for the OS to manage fine-grained sharing of the GPU resources. As a result, fairness, responsiveness, and quality of service of applications using GPUs cannot be controlled. As future systems continue further integration of CPUs and GPUs in the same chip, this problem will only increase. In this paper we introduce hardware extensions to modern GPUs that enable efficient sharing of GPUs among several applications and the implementation of flexible scheduling policies for multiprogrammed workloads. We propose two execution preemption mechanisms and the DSS scheduling policy that uses these mechanisms to implement dynamic spatial sharing of the GPU cores across kernels that belong to different processes. Moreover, DSS can be controlled by the OS to enforce system-wide scheduling policies. Experimental results show that hardware preemption mechanisms are necessary to obtain lower and more deterministic turnaround times for applications while having lower overheads than what was previously assumed, thus opening the possibility of the utilization of GPUs to perform computations in multiprogrammed interactive environments. We also show that a dynamic scheduling policy that assigns different GPU cores to concurrently running kernels can greatly improve system-wide metrics such as fairness. Finally, we experimentally show that the wide-spread believe that context switching in GPUs is probability expensive, does not hold when targeting multiprogrammed systems.

Acknowledgements

We would like to thank the anonymous reviewers, Alexander Veidenbaum, Carlos Villavieja, Lluis Vilanova, Lluc Alvarez, and Marc Jorda on their comments and help improving our work and this paper. This work is supported by European Commission through TERAFLUX (FP7-249013), Mont-Blanc (FP7-288777), and RoMoL (GA-321253) projects, NVIDIA through the CUDA Center of Excellence program, Spanish Government through Programa Severo Ochoa (SEV-2011-0067) and Spanish Ministry of Science and Technology through TIN2007-60625 and TIN2012-34557 projects.

References

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–12.

[2] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics 2009*. ACM, 2009, pp. 145–149.

[3] AMD, "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

[4] AMD, "AMD Graphics Cores Next (GCN) architecture white paper," 2012.

[5] ARM, "ARM Mali," 2012. [Online]. Available: www.arm.com/products/multimedia/mali-graphics-plus-gpu-compute

[6] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 287–296.

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

[8] A. Branover, D. Foley, and M. Steinman, "AMD Fusion APU: Llano," *Micro, IEEE*, vol. 32, no. 2, pp. 28–37, 2012.

[9] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 89–100.

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

[11] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 407–420.

[12] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*. USENIX Association, 2012, pp. 10–10.

[13] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the CUDA scheduler," in *Workshop on Programming Models for Emerging Architectures*, 2009, pp. 69–76.

[14] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[15] Intel, "4th generation Intel Core processors are here," 2012. [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/4th-gen-core-processor-family.html

[16] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 57–66.

[17] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "Time-Graph: GPU scheduling for real-time multi-tasking environments," in *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, 2011, p. 17.

[18] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *USENIX ATC*, vol. 12, 2012, pp. 37–37.

[19] G. Kyriazis, "Heterogenious System Architecture: a technical review," AMD, 2012.

[20] T. Li, V. K. Narayana, E. El-Araby, and T. El-Ghazawi, "GPU resource sharing and virtualization on high performance computing systems," in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 733–742.

[21] 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.

[22] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: Exception support and speculative execution on gpus," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*. IEEE, 2012, pp. 72–83.

[23] NVIDIA, "Next generation CUDA computer architecture Kepler GK110," 2012.

[24] NVIDIA, "Sharing a GPU between MPI processes: multi-process service (MPS) overview," 2013.

[25] NVIDIA, "Programming guide - CUDA toolkit documentation," 2014. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[26] 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.

[27] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. ACM, 2013, pp. 407–418.

[28] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014, pp. 743–758.

[29] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 217–228.

[30] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: operating system abstractions to manage GPUs as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

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

[32] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th annual International Symposium on Computer Architecture*, ser. ISCA '85, 1985, pp. 36–44.

[33] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg, "Softshell: dynamic scheduling on GPUs," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, p. 161, 2012.

[34] J. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. Liu, and W. Hwu, "The Parboil benchmarks," Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[35] 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.

[36] N. Tuck and D. M. Tullsen, "Initial observations of the simultaneous multithreading Pentium 4 processor," in *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 2003. IEEE, 2003, pp. 26–34.

[37] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "FAME: Fairly measuring multithreaded architectures," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. IEEE, 2007, pp. 305–316.

[38] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *Micro, IEEE*, vol. 31, no. 2, pp. 50–59, 2011.

[39] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *arXiv preprint arXiv:1303.5164*, 2013.