Towards multiprogrammed GPUs

by

Ivan Tanasić

Towards Multiprogrammed GPUs

by
Ivan Tanasić

Advisor: Prof. Nacho Navarro
Co-advisors: Dr. Isaac Gelado, Prof. Eduard Ayguade

DISSERTATION
Submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy (Doctor per la UPC)
in the Department of Computer Architecture

Universitat Politècnica de Catalunya
Barcelona, Spain
Fall 2016

Abstract

Programmable Graphics Processing Units (GPUs) have recently become the most pervasive massively parallel processors. They have come a long way, from fixed function ASICs designed to accelerate graphics tasks to a programmable architecture that can also execute general-purpose computations. Because of their performance and efficiency, an increasing amount of software is relying on them to accelerate data parallel and computationally intensive sections of code. They have earned a place in many systems, from low power mobile devices to the biggest data centers in the world. However, GPUs are still plagued by the fact that they essentially have no multiprogramming support, resulting in low system performance if the GPU is shared among multiple programs. In this dissertation we set to provide the rich GPU multiprogramming support by improving the multitasking capabilities and increasing the virtual memory functionality and performance.

The main issue hindering the multitasking support in GPUs is the non-preemptive execution of GPU kernels. Here we propose two preemption mechanisms with different design philosophies, that can be used by a scheduler to preempt execution on GPU cores and make room for some other process. We also argue for the spatial sharing of the GPU and propose a concrete hardware scheduler implementation that dynamically partitions the GPU cores among running kernels, according to their set priorities. Opposing the assumptions made in the related work, we demonstrate that preemptive execution is feasible and the desired approach to GPU multitasking. We further show improved system fairness and responsiveness with our scheduling policy.

We also pinpoint that at the core of the insufficient virtual memory support lies the exceptions handling mechanism used by modern GPUs. Currently, GPUs offload the actual exception handling work to the CPU, while the faulting instruction is stalled in the GPU core. This stall-on-fault model prevents some of the virtual memory features and optimizations and is especially harmful in multiprogrammed environments because it prevents context switching the GPU unless all the in-flight faults are resolved. In this disser-

tation, we propose three GPU core organizations with varying performance-complexity trade-off that get rid of the stall-on-fault execution and enable preemptible exceptions on the GPU (i.e., the faulting instruction can be squashed and restarted later). Building on this support, we implement two use cases and demonstrate their utility. One is a scheme that performs context switch of the faulted threads and tries to find some other useful work to do in the meantime, hiding the latency of the fault and improving the system performance. The other enables the fault handling code to run locally, on the GPU, instead of relying on the CPU offloading and show that the local fault handling can also improve performance.

Acknowledgements

This thesis has been a great journey which would not be possible without many great people that helped me along the way. For that, I wish to thank them all.

First and foremost, I would like to thank my advisors for their guidance and continuous support during my time in Barcelona. They have thought me so many things and helped me so many times that I will not be able to repay them, ever. Nacho Navarro always made sure that I have resources and freedom to pursue my research. I thank him for inspiring confidence in my research and his readiness to help with any issue I faced. It is a great loss that he is not here with us anymore. I know how proud and happy he would be to see me finish this thesis. Isaac Gelado has been another great role model who set the bar high with his strong work ethic. I am grateful for his dedication, patience, and the ability to still find time for me, years after leaving Barcelona. Eduard Ayguade has stepped in after Nacho passed away and made sure I actually finish this thesis. I appreciate all the help he provided during this last year.

I would like to thank Mateo Valero for providing the opportunity and a great environment to do my PhD, as well as helping with my ISCA 2014 paper. My thanks also go to Alexander Veidenbaum for suggestions that improved that paper, Alex Ramirez for his generous help and advice during my PhD, Wen-mei Hwu for being a great teacher and helping with the GPGPU 2013 paper, and Veljko Milutinovic for helping me get here in the first place.

Two internships that I had during my PhD were remarkable learning opportunities. I would like to thank Santosh Abraham and the rest of the Advanced Processor Lab for the fantastic four months at Samsung Research America. It was an eye-opening experience during which I learned so much about GPUs and computer architecture in general. I would also like to thank Steve Keckler, Arslan Zulfiqar and the rest of the Architecture Research Group for the great three months at NVIDIA. The work and discussions done during this internship have inspired me to pivot the second half of the thesis towards exception handling mechanisms, rather than use-cases.

I was very fortunate to be a member of GSO, an extraordinary group of people that made my PhD studies so memorable and special. My sincere thanks go to all of the GSO members with whom I had the pleasure to work with and who were unfortunate enough to sit through my numerous presentation rehearsals and provided endless paper reviews. I overlapped with Carlos Villavieja the least, yet he is one of the guys that helped me the most. From helping me get internship opportunities to practicing job interviews, Carlos still keeps his title of an eminent GSO member, five years after leaving Barcelona. Javier Cabezas and Lluis Vilanova have been like a father and a mother to me and other younger students. Their knowledge, technical ability and selflessness never cease to amaze me. Ramon Bertran served as a proof that hard work, focus and positive attitude can get you far. Lluc Alvarez kept me on my toes and provided his expertise in microarchitecture. Marc Jorda was a true brother in arms, always ready to get down in the trenches with me. Victor Garcia, Diego Marron and Pau Farre helped me finish my PhD by requiring way less support and attention than I required at the beginning. I would go as far as saying that I cannot imagine a better research group, except that I do not see GSO as just a research group, but rather something more special. Comradeship during both work time and playtime was astonishing. And there was plenty of both.

A great group of students at Barcelona Supercomputing Center made my time here more enjoyable. Many discussion about life or work were held over coffee, lunch and beers with Nikola Rajovic, Milovan Duric, Michalis Alvanos, Thomas Grass, Alex Rico, Karthikeyan Palavedu Saravanan, Milan Pavlovic, Ugljesa Milic, Darko Zivanovic, Branimir Dickov and many others.

Last, but certainly not least, I want to thank my family for providing their endless love, support, and inspiration to start and finish this journey. My father Branislav with his dedication to everything that he started, my mother Miroslava with her confidence in better tomorrow and my wife Dorotea with her endless patience for me. I am not done learning from you.

The work on this thesis was performed at Barcelona Supercomputing Center and was financially supported by the European Commission through *TERAFLUX: Exploiting Dataflow Parallelism in Teradevice Computing* project (contract FP7-249013), Spanish Ministry of Science and Technology through projects *Computación de Altas Prestaciones V* (contract TIN2007-60625) and *Computación de Altas Prestaciones VI* (contract TIN2012-34557), and NVIDIA Corporation through the *GPU Center of Excellence* grant.

Contents

List of Figures

List of Tables

Glossary

ASIC Application Specific Integrated Circuit
CMP Chip Multi-Processing
CMT Chip Multi-Threading
CPU Central Processing Unit
DRAM Dynamic Random Access Memory
DSM Distributed Shared Memory
FCFS First Come First Served
GPU Graphics Processing Unit
HPC High Performance Computing
ILP Instruction Level Parallelism
I/O Input / Output
IPC Inter Process Communication
IR Intermediate Representation
ISA Instruction Set Architecture
NUMA Non-Uniform Memory Access
MMU Memory Management Unit
OoO Out of Order
OS Operating System
RaW Read after Write
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Threads
SM Streaming Multiprocessor
SMT Simultaneous Multithreading
SoC System on Chip
SRAM Static Random Access Memory
TB Thread Block
TLB Translation Lookaside Buffer
TLP Thread Level Parallelism
VLIW Very Long Instruction Word
WaR Write after Read

Chapter 1

Introduction

Over the course of the last several decades, an increasing amount of challenging and computationally intensive tasks in 2D and 3D rendering has been migrated from software into specialized hardware accelerators. Due to the evolution of integrated circuit technology a single-chip solution emerged, today known as the Graphics Processing Unit (GPU).

In order to enhance the quality of the rendered images and make them more realistic, GPUs have been improving flexibility and performance with each new generation. Greater flexibility was achieved through increased programmability, that is moving from the rendering pipeline with configurable stages [142] to the rendering pipeline with programmable stages [101, 111, 102, 172]. Greater performance was achieved through increasing the memory bandwidth and the number of cores to exploit the inherent data parallelism of graphics workloads [111, 19]. Increasing parallelism resulted in GPUs with higher peak performance than general-purpose CPUs, and this performance gap keeps growing[1].

The trend of increasing performance gap was noticed by researchers and developers who started leveraging GPUs to improve the performance of other, non-graphics related computations [123, 122]. Ultimately, NVIDIA introduced the Tesla G80 [102] architecture with hardware and software improvements geared specifically towards general-purpose computing. Other vendors, like AMD [67, 105] and Intel in the desktop market, as well as others in the mobile market (Qualcomm, Imagination, ARM, etc.) followed closely. These improvements have a goal of abstracting away the graphics pipeline and its limitations. They include the unified shader core capable of executing vertex, fragment and compute shaders (i.e., procedures) [102] on the

[1] Currently the gap is an order of magnitude increase in double precision arithmetic performance and memory bandwidth [120].

hardware side, and dedicated programming languages like CUDA [120] and OpenCL [57] on the software side.

Today, GPUs are the most pervasive massively parallel processors and can be found in wide range of systems. On one end of the spectrum are the mobile SoCs that power smartphones and embedded systems. Their GPUs are used for computational photography [131, 130, 12, 64], augmented reality [107] or other computer vision tasks [168] such as face recognition [29] or road sign detection [114] in autonomous vehicles.

In the middle of the spectrum are traditional GPU markets like desktop (running 3D games) and workstation (running CAD and modeling tools) that have also benefited from general-purpose features. Games today perform physical based simulation (e.g., cloth [136], particle [85] and fluid [61] simulations) on the GPU to achieve complex visual effects. Similarly, CAD and other engineering tools can use the GPU for computation (e.g., photo realistic rendering with ray tracing [132, 127]).

On the other end of the spectrum are large scale machines used in HPC [46, 84, 165] and warehouse scale computers [13]. Recent advances in machine learning [16, 91, 75] have created a surge of intelligent services that rely on speech recognition, natural language processing and image classification. GPUs are playing a central role in this revolution, speeding up their adoption in data centers and public clouds [63, 62].

Even though a major effort was invested in making them useful beyond graphics applications, GPUs are still missing features that prevent them from being considered as truly general-purpose processors. The reason for holding back the evolution of GPU architectures is to avoid hurting the performance and efficiency of interactive 3D rendering (e.g., computer games), which is still the primary GPU market. One notable consequence is the lack of multiprogramming support typically found in general-purpose systems.

The concept of multiprogramming was originally introduced with the goal of improving the system *throughput* by filling the execution gaps (e.g., during I/O operations) of one process with work from another process. However, system throughput is not the only metric that matters in multiprogrammed systems. For example, in interactive systems it is equally important to provide *responsiveness* to prevent sluggish performance which would otherwise frustrate the users. Similarly, in multi-tenant system it is also important to provide *fairness*, so that the customers get the resources that they payed for.

GPUs, designed first and foremost as graphics accelerators, were intended to be used by a single application at a time. This is progressively becoming an issue, as GPUs are finding their way into inherently multiprogrammed environments, such are desktop and cloud. Traditionally, the OS is at the heart of multiprogrammed system, providing resource virtualization and, through

it, numerous benefits like protection (i.e., security), performance isolation, programmability, etc. Because GPUs are missing most of the multiprogramming support, GPU-accelerated systems are unable to provide key multiprogrammed workload requirements.

In this dissertation we focus on solving the two fundamental obstacles that are hindering the multiprogramming support in GPUs. The first is the *non-preemptive execution* of GPU kernels which means that once a GPU kernel starts its execution, it has to run until completion. The non-preemptive execution makes it impossible for the system to perform resource allocation (i.e., assign GPU cores to different processes) at will and through it enforce system properties (e.g., responsiveness). The second is the *non-preemptible exceptions* on the GPU which means that the instruction that caused the exception is stalled in the middle of the pipeline, while servicing the exception has to be offloaded to the CPU. This has become a problem because the latest generation of GPUs implement on-demand page migration (i.e., page swapping) between CPU and GPU memories using the page fault mechanism (a type of exceptions) [175]. Migrating pages is costly and can leave GPU cores underutilized for significant amount of time during which a context switch of the GPU core cannot be performed.

1.1 Contributions

The objective in this dissertation is to show that, with moderate amount of added complexity, multiprogramming support can be introduced to current GPU architectures without harming the performance of single-application execution.

We argue and provide experimental data to support the claims that preemptive multitasking and preemptible exceptions can be implemented in such a way that accomplishes the objective. We further study, analyze and evaluate the implementation and performance issues that arise when enabling multiprogramming support in GPUs. Following is the list of contributions in the field of computer architecture made in this dissertation.

1.1.1 Enabling Preemptive Multitasking

In this dissertation we demonstrate that preemptive multitasking is not only desired but also a feasible approach to multiprogramming on GPUs. We introduce two preemption mechanisms with different effectiveness and discuss their implementations. One is inspired by the classic operating system preemption mechanism where the execution on GPU cores is stopped and

their context is saved to memory, implementing true preemptive multitasking. The other mechanism exploits the semantics of the GPU programming model and the nature of GPU applications to implement a form of cooperative multitasking. We study the performance of the two mechanisms in terms of preemption latency and introduced overheads, as well as how they translate to final system performance. Finally, we show that both mechanisms provide improvements in multiprogrammed systems, increasing responsiveness and fairness at the expense of a small loss in throughput.

1.1.2 Design of a GPU Kernel Scheduler

We further propose hardware extensions that remove the constraint of one process having exclusive access to the GPU and allow the utilization of GPU cores individually. These extensions enable different processes to concurrently execute GPU kernels on different sets of GPU cores and provide a framework for implementing kernel schedulers that use the previously introduced preemption mechanisms. Here we also present Dynamic Spatial Sharing (DSS), a concrete scheduler implementation that dynamically partitions the GPU cores and allocates them for different processes according to the priorities assigned by the OS. Finally, we demonstrate the improved responsiveness and system fairness of DSS over the baseline scheduling policy.

1.1.3 Enabling Preemptible Exceptions

In this dissertation we also propose novel approaches to supporting preemptible exceptions, without resorting to a full-on precise exceptions model. Avoiding the precise exceptions is necessary in order to keep the high performance and efficiency of the GPU core pipeline. Instead, we impose the minimal amount of execution constraints and track the minimal amount of additional state so that a well-defined restart point is provided, at which context switch can be performed. We present three low-overhead design choices with varying performance-complexity trade-offs. The simplest approach introduces limitations to the execution model and requires minimal changes to the GPU pipeline at the cost of decreased performance. The most comprehensive solution introduces additional hardware structures resulting in a small increase in the area of the GPU core due to extra storage, but completely preserves the performance of the baseline GPU pipeline.

1.1.4 Page Fault Latency Hiding Scheme

We introduce a scheme for hiding the latency of page faults by performing a context switch on a fault in this dissertation. Servicing a page fault can take a significant amount of time, which in turn leads to severe system underutilization and performance loss. Instead of waiting for the fault to be resolved, this scheme switches out the faulted threads and frees the resources that can be used to execute some other useful work in the meantime. We present scheduler extensions that allow oversubscribing the GPU core with threads and selecting non-faulted threads for execution, exploiting the nature of GPU applications to improve their performance in the presence of page faults. Previously introduced preemptible exception support enables the restartability of the context switched threads, once the fault is resolved. We demonstrate that this technique can provide performance improvement for applications that are neither execution nor data transfer bound.

1.1.5 Handling Page Faults on the GPU Cores

This dissertation also proposes handling certain type of GPU-triggered page faults on the GPU itself, instead of offloading all the handling work to the CPU as the current GPUs do. This enables another standard feature of general-purpose systems, the lazy allocation of memory, by locally handling the page faults caused by writing to pages with no physical memory assigned (i.e., on-demand allocation of physical memory). The GPU code runs its own physical memory allocator, which reserves the required memory and fixes the GPU page table without interrupting the CPU and occupying the system interconnect. Only page faults that do not need to transfer data from the CPU (i.e., accesses to the GPU heap and output buffers) can be handled locally. Preemptible exceptions support clears the pipeline of faulted requests and thus guarantees that the fault handler can be safely executed on the faulted GPU core. We demonstrate improved performance on kernels with significant number of concurrent faults, which is the case for a considerable amount of benchmarks.

1.2 Organization

The rest of this dissertation is organized as follows:

Chapter 2: GPU Systems Background introduces all the technical details of modern GPU systems that are necessary for understanding the

proposals made in this dissertation. The chapter introduces the baseline architecture in a top-down fashion, going through the architecture of a GPU-accelerated system, GPU itself, its execution engine, and finishing with the GPU core microarchitecture.

Chapter 3: Multiprogrammed Systems discusses the state of the art in multiprogramming. It briefly summarizes the mechanisms and policies used to provide multitasking on CPUs, and gives a detailed review of related work in GPU multitasking. It further describes some common CPU virtual memory features and discusses the recent proposals found in the literature, made to circumvent the lack of these features in GPUs. Finally, it surveys the exception handling approaches previously proposed in the literature, focusing on both precise exceptions and alternative approaches to implementing preemptible exceptions.

Chapter 4: Methodology describes the workloads used for the evaluation and provides the basic information on the two simulation methodologies used to validate and evaluate the proposals in this dissertation.

Chapter 5: Enabling Preemptive Multitasking covers contributions 1 and 2, providing the motivation, detailed architecture of the proposals, performance evaluation and details of the evaluation methodology.

Chapter 6: Enabling Preemptive Exceptions covers contributions 3, 4, and 5, providing the motivation, detailed architecture of the proposals, performance evaluation and details of the evaluation methodology.

Chapter 7: Conclusions and Future Work concludes the dissertation. It reviews the key contributions and insights presented in this dissertation and presents potential future work and open research lines.

Chapter 2

GPU Systems Background

In this chapter we present the architecture of the baseline system used throughout this dissertation in a top-down manner. The focus is only on the details necessary for understanding our proposals.

2.1 GPU Accelerated Systems

Originally, GPUs have started life as system expansion components, residing on the add-in-boards (here referred to as the *discrete* GPUs). Increasing commoditization resulted in inclusion of GPUs into the motherboard chip-set for desktop and laptop computers (the *integrated* GPU mode, mostly abandoned today). Finally, the raise of graphics requirements in mobile markets resulted GPUs in becoming part of the system-on-chip (SoC) (referred to as the *fused* GPU model). The two common form factors (discrete and fused) are shown in Figure 2.1.

In the discrete GPU model, a GPU card is connected to a system through the system expansion bus (e.g., PCI express or NVLink). In its most basic form, the system has one CPU attached to its own system memory, while the expansion card has one GPU attached to the graphics memory. Traditionally, the two memories have been incoherent, and it was the job of the programmer to keep them coherent by performing explicit data transfers [86, 120]. Configurations with multiple CPUs and/or GPUs are common in high performance environments, and further increase the complexity of the system [27].

Alternatively, the fused GPU model puts the CPU cores and GPU cores on the same die. Both GPU and CPU in this case use the same physical memory (system memory). Implementations range from physically partitioned memory (in which case the explicit data transfers between two partitions are

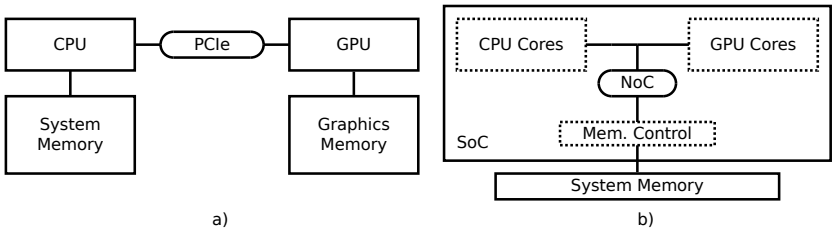


Figure 2.1: GPU accelerated systems in their basic forms: a) descrete GPU model with GPU and graphics memory on the expansion card, and b) fused GPU model with CPU and GPU integrated in the same chip.

still necessary) [22] to the latest designs that keep the CPU and GPU cache coherent [73]. Despite the improved programmability of the cache coherent design, the discrete model is likely to stay dominant, especially in the HPC and data center domains, because of the much higher performance[1].

In general case a GPU application consists of (usually in repetitive bursts):

- CPU execution that performs control, preprocessing or I/O operations, as well as serial portion of the algorithm;
- GPU execution of kernels that performs computationally demanding tasks (parallel portion of the algorithm); and
- data transfers between CPU and GPU that bring input data to the GPU memory and return the outputs to the CPU memory.

```
float *x, *y, *dev_x, *dev_y;

... // 2 x malloc + 2 x cudaMalloc
read_from_file(N, x, y, ...);                    <- I/O

cudaMemcpy(dev_x, x, N, cuadMemcpyHostToDevice);
cudaMemcpy(dev_y, y, N, cudaMemcpyHostToDevice);   <- Data Transfers

saxpy<<<GDIM, BDIM>>>(N, 2.0, x, y);             <- GPU kernel

cudaMemcpy(y, dev_y, N, cuadMemcpyDeviceToHost);   <- Data Transfers

write_to_file(N, y, ...);                        <- I/O
... // 2 x free + 2 x cudaFree
```

Figure 2.2: Simple SAXPY GPU program with explicit data transfers (programmer managed).

An example program performing the SAXPY linear algebra operation on the GPU is shown in Figure 2.2. It first allocates the buffers on CPU and

[1]Discrete GPUs usually have higher core count (they are on a separate chip) and memory bandwidth (graphics memory uses wider buses and runs at higher frequency than CPU memory).

GPU and loads the data from a file into the CPU buffers. Data is then copied to the GPU memory, before the *saxpy* kernel is launched to compute the result. The output is then copied back to the CPU memory, and written to a file.

CUDA is a Single Program Multiple Data (SPMD) style programming model [38] and a kernel launch consists of a number of threads executing the same code. Threads are grouped into *thread blocks* that are independent of each other. Only threads from the same thread block can cooperate using barrier synchronization and communication through local memory (shared memory in CUDA terminology).

The GPU device driver (executing in the CPU, as part of the OS) is in charge of performing the bookkeeping tasks for the GPU (e.g., managing the GPU memory space). GPU kernel invocations (*kernel launch* in CUDA terminology), initiation of data transfers, and GPU memory allocations are performed in the CPU code (referred to as commands in the rest of this text).

GPU programming models 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 [77], commands are sent to the GPU as soon as possible. Once commands are issued to the GPU, the software has no control over them anymore. 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.

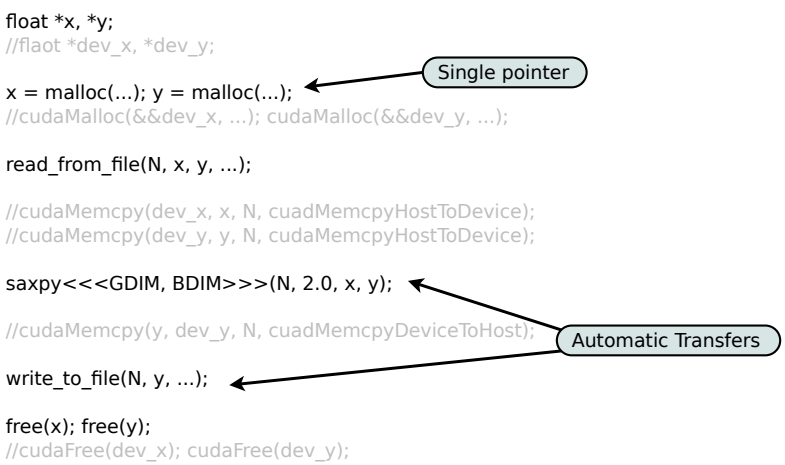


Figure 2.3: Simple SAXPY GPU program with automatic data transfers (demand paging).

The latest generation of NVIDIA GPUs, GP100 Pascal, supports the demand paging between CPU and GPU memories [121, 37]. On demand page

migration eliminates the need for explicit programmer data transfers, significantly improving the programmability of the system. Furthermore, it also allows oversubscription of the GPU memory, increasing the number of applications that can benefit from GPU acceleration. Figure 2.3 shows the version of the SAXPY program from Figure 2.2, rewritten to rely on the new demand paging feature, instead of programmer controlled data transfers. Noticeable is the lack of double pointers (host and device), associated allocations and deallocations, as well as data transfer calls.

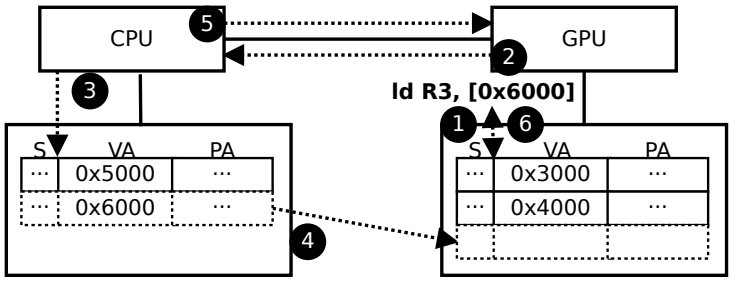


Figure 2.4: On demand page migration.

Figure 2.4 illustrates the steps taken by the baseline system while performing demand paging. When the kernel running on the GPU performs an access to a page owned by the CPU, after walking through the GPU page table the MMU determines that the page is not present in the GPU memory (1) and sends an interrupt to the CPU (2). The CPU interrupt handler, implemented by the GPU OS device driver (3), transfers the contents of the faulting page from the CPU memory to the GPU memory, updates both CPU and GPU page tables to reflect the new location of the page (4), and notifies the GPU that the page has been successfully migrated (5). The MMU broadcasts this information to GPU cores, in order to resend the faulted requests [175]. Note that this only replays the memory request (from the microarchitectural state) and does not replay the instruction itself. On the retried page table walk, the MMU finds a valid translation and sends it back to the core which now can let the faulting instruction continue execution (6). Effectively, this scheme treats GPU page faults as very long latency TLB misses which stall the execution of the instructions causing the page fault.

2.2 GPU Architecture

The base architecture assumed in this paper is depicted in Figure 2.5. The GPU has an execution engine and a data transfer engine used to copy the data between CPU memory and GPU memory. Even though it typically has

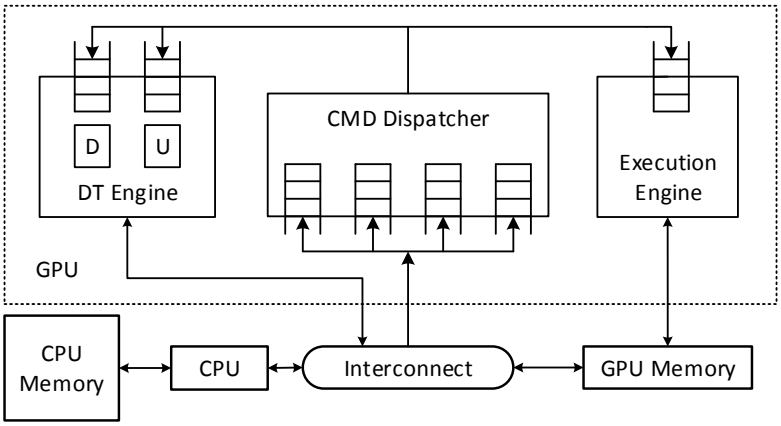


Figure 2.5: GPU architecture with its main components: Command (CMD) Dispatcher, Data Transfer (DT) Engine, and Execution Engine.

multiple cores, the execution engine is treated as one unit 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 [117, 23]. The GPU device driver maps streams from the applications on the command queues. The command dispatcher is in charge of inspecting the top of the command queues and issuing 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. After issuing a command, the dispatcher stops inspecting that queue until the command completes. 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 use several queues to remove the problem of false dependencies introduced by the design with one command queue [171], and further increase the opportunities to overlap independent commands.

The GPU also includes a set of global control registers 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.

At any given moment, the GPU is executing commands from one context only[2]. This constraint, coupled with non-preemptive execution leaves room

[2]Note that the GPU memory however can hold data from multiple contexts at the same time.

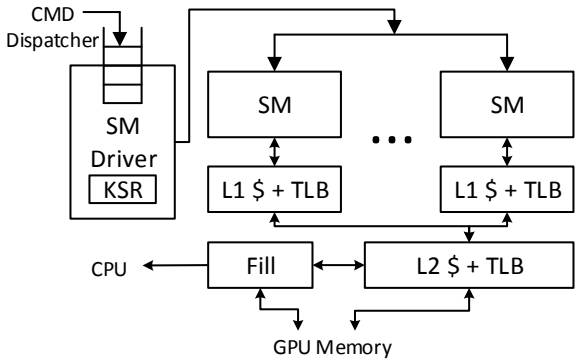


Figure 2.6: Execution engine architecture.

for very coarse grained sharing, only. All commands submitted to the GPU must complete before switching to another context.

2.3 Base GPU Execution Engine

The base GPU execution engine we assume in this paper is shown in Figure 2.6. It has a number of computation cores (Streaming Multiprocessors, or SMs, in CUDA terminology) that varies, depending on the deployment environment, from one (targeting mobile SoCs) to several few dozens (targeting HPC). Each SM has a private L1 cache and L1 TLB, while the L2 cache and TLB are shared between all SMs. A shared fill unit implements the hardware logic of page table walkers and communication with the CPU in case of page faults.

The SM driver sits at the front of the execution engine and distributes the work to the SMs. It receives kernel launch commands via the execution engine queue, and sets up the Kernel Status Registers (KSR) with the control information such as number of work units to execute, kernel parameters (number and stack pointer), etc. 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.

When a thread block is issued to an SM, it remains resident in that SM until all of its threads 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 (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.

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, active, kernel to that SM. This policy combined with static partitioning of hardware resources means that kernels with enough thread blocks 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 Core Architecture

We illustrate the detailed SM pipeline in Figure 2.7. Each thread block executed on the SM is further divided into warps, groups of 32 threads adhering to the Single Instruction Multiple Threads (SIMT) execution model [102]. Threads in a warp execute in a lock-step, acting as a single thread of the Single Instruction Multiple Data (SIMD) execution model [52], until threads take different paths on a conditional branch. At that point a compiler controlled [95] mechanism in the form of a stack (usually referred to as the reconvergence stack [53]) is used to execute both paths of a branch, one after the other [97, 53]. Our baseline uses the immediate post-dominator basic block as the reconvergence point [53]. Threads of the same thread block can be synchronized with each other using barriers. A hardware barrier unit in each SM keeps track of blocked and active threads of a thread block and reactivates them all when each one has executed the barrier instruction.

Because the SM is a throughput oriented processor, designers have opted for multithreaded execution of warps as a mean of hiding the pipeline latencies [146, 170]. Thus each clock cycle, the warp scheduler (WS) picks one ready warp, for which an instruction line will be fetched from the instruction cache (IC) in the next cycle and sent to the issue queue. The unified issue queue contains instructions from multiple warps and can issue multiple instructions per cycle, from one or different warps. A score-boarding (SB)

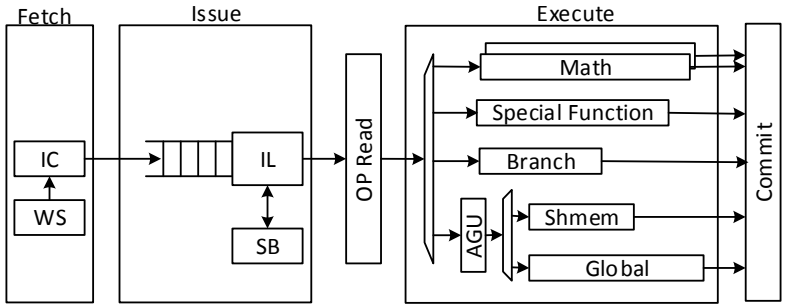


Figure 2.7: SM microarchitecture.

mechanism [157] is at the heart of instruction scheduling, making sure that dependencies among instructions from the same warp are enforced. The issue logic (IL) thus implements out-of-order issue for most instruction types, as long as dependencies are honored. However, unlike typical out of order processors, SM does not support speculation nor register renaming. Both control flow and memory dependence speculation are avoided by briefly disabling the fetch for a warp upon fetching a control flow or memory instruction. Memory instructions re-enable the warp at issue time, forcing the in-order issue with respect to other memory instructions of the same warp. Control flow instructions re-enable the warp during commit.

Once issued, instructions go through the operand read stage, in which the register file is accessed and the data is dispatched to the execution units. The SM has a very large unified (integer and floating point) register file that is partitioned among all resident warps. The execution stage consists of several math units, a branch unit, a special functions unit and different memory pipelines. The shared memory pipeline accesses the on-chip scratch-pad memory that holds CUDA *shared address space* objects, which are not subject to memory translation (each SM is used by only one user process at a time). The global memory pipeline performs the accesses to the off-chip memory that go through the cache hierarchy and memory translation (global memory can hold allocations from different processes at the same time).

Upon completion, each instruction is sent to the commit stage, resulting in out-of-order commit. Manipulating the score-boards is split between the operand read stage, in which source operands are released (after reading), and the commit stage, where the destination operand is released (after writing). Early score-board release for source operands mitigates the WAR (Write After Read) hazard in absence of register renaming.

Several features discussed in this dissertation require running system software (i.e., trap handling routine and page fault handler) on the GPU. To prevent the user code from directly modifying system data structures, such

as the GPU page table and the physical memory allocator, the privileged execution mode is required. There is no public evidence that GPUs support privileged execution, but it is easy to see how sufficient support for our use cases is easy to achieve. We can assume two different privilege (user and system) execution levels which first requires keeping additional bit per warp to distinguish the privilege level and using it to detect violations. The bit is checked in the decode stage to protect the user level code from executing privileged instructions. The GPU page table entires also need to be extended with one bit that distinguishes pages from user and supervisor (i.e., system), similar to what modern CPUs do [72]. This bit is then checked during memory translation, together with other access violations checks (e.g., write protected pages) to protect the system memory.

Chapter 3

Multiprogrammed Systems

Today most computers, spanning the full spectrum from a portable media player to the fastest supercomputer in the world, are multiprogrammed, allowing several programs (potentially from different users) to run concurrently. Essential in enabling the multiprogramming capability is the operating system, who's role is to provide the management of the machine resources and isolation of applications running on the machine. The OS achieves this through the concept of virtualization, creating an illusion that the program is running on the system alone.

Virtualization of the processor is typically achieved through OS controlled time-multiplexing of programs on the processor. Periodically, the execution of a program gets suspended by the timer interrupt and the OS gains control of the processor. The OS can then choose to run another program for some time and perform a context switch by saving the context of the preempted process to memory, and restoring the context of another process. Different processor scheduling techniques have been proposed in effort to improve the performance on different metrics such as system fairness, improved responsiveness, or overall system throughput.

Virtualization of the memory is achieved through a set of mechanisms collectively referred to as virtual memory. Virtual memory provides isolation between the programs, so that they cannot access each other's data (intentionally or by accident), unless explicitly allowed. Isolation is maintained by using one virtual address space per process and translating (mapping) memory accesses from the virtual address space to the actual memory present in the system (i.e., physical address space). Memory translation also enables another great feature of the virtual memory, allowing the virtual address space to be much bigger than the physical memory installed in the system. If the program uses more memory than available, the OS can transparently

move the data (i.e., demand paging) between the DRAM and slower but bigger store such as hard drives, using the page fault mechanism to detect accesses to data residing in the backing store.

Some of the mechanisms used for processor and memory virtualization are built on the same low-level mechanisms. Timer interrupts and page faults are both built on top of the exceptions mechanism that is used to signal events which require special attention. When the exception is raised, the processor stops executing the current process and invokes the exception handler implemented in the OS (or at least registered with it). If the cause of the exception is resolved by the handler (e.g., the exception was not caused by an illegal memory access), the interrupted process can be restarted.

The reminder of this chapter is a survey of the state of the art in process scheduling, virtual memory and exceptions handling.

3.1 Process Scheduling

The process scheduler plays a part in sustaining the illusion of each process having the system to itself. To do that, the scheduler usually needs to balance few, sometimes contrary requirements. That is, it needs to preserve *fairness* between processes while keeping the system *responsive* (more important in interactive systems) and not let any process starve. It also needs to make sure that the machine resources are well used, keeping the *system throughput* as high as possible. Since balancing all these metrics is quite hard (or even impossible) to achieve, schedulers usually focus on few of them, depending on the environment the system is deployed in.

3.1.1 CPU Scheduling

Some of the classical approaches to process scheduling include the *first come first serve* (FCFS), *round robin* (RR) [88], and *shortest job first* (SJF) [32] and its preemptive sibling *shortest time to completion first* (STCF) [33] algorithms. Of these three, perhaps round robin only deserves to be called a scheduler, as SJF and STCF are mostly impractical (in general case the length of each job is unknown) and FCFS only performs basic queueing. *Multi-level feedback queues* (MLFQ) [35] is a very common scheduling paradigm, used in Solaris, BSD derivatives, and Windows NT based systems [106, 96, 150, 11]. In essence, the MLFQ schedulers tries to predict the behavior of processes based on their past (length of the CPU burst) and prioritize the interactive and I/O bound processes (short CPU burst).

Another common category are the proportional share (or fair-share) schedulers [79], that aim to guarantee a certain percentage of CPU time to a process (usually proportional to their priority). Stochastic implementations such is the *lottery scheduler* [166] or deterministic such is the *borrowed virtual time scheduler* (BVT) [41] have been proposed. Proportional share schedulers are most commonly used for the processors scheduling at the hypervisor level [30]. In the past few years, Linux has transitioned from the MLFQ-like *O(1)* scheduler [20] to the proportional share *completely fair scheduler* (CFS) [110, 124].

As the processors became more concurrent and complex, so did the schedulers. Increased levels of concurrency are making current implementations ineffective [89, 103]. Furthermore, contention of shared resources (e.g., caches and memory controllers) have elevated the problem of performance interference [70, 83, 155, 133, 115, 113, 116, 34], which makes the current SMP schedulers even less effective. Several approaches to improve performance through OS level scheduling have been proposed for SMT [26, 48], CMT [47], CMP [49, 177], and NUMA [18].

3.1.2 GPU Scheduling

Improving the multiprogramming (or multitasking, in general) capabilities of the GPU was one of the early goals set by the research community. GERM [15] and TimeGraph [77] focus on improving responsiveness of the graphics applications running on the GPU. Both provide a GPU command scheduler integrated in the device driver, that chooses the context from which commands should be submitted to the GPU next. Once the command is submitted to the GPU, their scheduling cannot be controlled any more. Because of the high cost of submitting the command to the GPU commands are submitted in batches, requiring the scheduler to balance the cost of command submission versus the longer execution of the command batch.

RGEM [76] is a software runtime library targeted at providing responsiveness to prioritized CUDA applications by scheduling DMA transfers and kernel invocations. One of the problems that RGEM tries to solve are the non-preemptive DMA transfers. Their approach is to implement memory transfers as a series of smaller transfers and thus create the potential scheduling points, lowering the stall time due to the competing memory transfers.

Gdev [78] is built around these principles, but integrates the runtime support for GPUs into the OS. Disengaged scheduling proposed in [109] aims at providing the safe and fair scheduling conducted by the OS, while minimizing the overhead introduced by crossing the user/kernel boundary on each command submission. PTask [137] 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.

Li et al. [100] introduced a software virtualization layer that makes all the participating processes execute kernels in the same GPU context. They instantiate a proxy process that receives requests from client processes and executes them on the GPU. This approach, later adopted by NVIDIA's own Multi-Process Service (MPS) runtime system [118], eliminates the overheads imposed by the constraint that only one context can be active at a time, and is especially beneficial for the GPU accelerated MPI applications [118]. Running one MPI rank per core is the common approach in HPC, especially with legacy applications, and since a typical GPU accelerated node has a ratio of half a dozen or so CPU cores to each GPU, sharing eliminates the overheads and improves utilization of the GPU [21, 171]. However, sharing the context breaks the memory isolation between the participating processes and thus it is not a general solution. Furthermore, 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. Fused kernel contains an *if* statement to check 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. [58] proposed a runtime system for CUDA which chooses between running the fused kernel or running the kernels sequentially. Wang et al. [167] used a similar technique, and observed the improved energy efficiency as a result of the better GPU utilization. Extending this approach, Gregg et al. [56] implemented an OpenCL kernel that occupies the whole GPU and dynamically invokes kernels to be executed, acting as a scheduler. Kernel fusion approaches were the first take on improving the concurrency of the GPU execution engine. As such, their contribution is mostly in demonstrating the benefits rather than proposing a concrete implementation, since they are highly impractical.

Alternatively, several software approaches have been proposed to implement spatial partitioning of the GPU. The common idea is to underutilize the GPU by one application, and rely on the back-to-back execution so that concurrent execution could be achieved. Ravi et al. [135] rely on the *molding* technique, that is changing the dimensions of grid and thread blocks while preserving the correctness of the computation (if possible). Molding is not

a generic technique and can be only applied to a small number of existing kernels, without developer having to transform the code to comply to requirements. Pai et al. [125] propose a similar technique and associated code transformation based on iterative wrapping [153] that produces an *elastic* kernel. These techniques rely on developer or compiler transformation to prepare the programs for concurrent execution. Furthermore, the back-to-back execution, which is the key enabler of these proposals, only works for kernels from the same context.

Similarly, several software techniques have been proposed to implement time multiplexing on GPUs. The *kernel slicing* technique used in [14, 125, 176] transforms the kernel so that the code is oblivious to the kernel launch parameters (e.g., the number of thread blocks). Such transformed kernel is then launched multiple times, passing the launch offset so that each slice performs only a part of the original computation, but all launches together perform the full computation. Static slice size [125, 14], or the smallest slice size in case of dynamic slicing schemes [176], are chosen a priori for the specific system configuration, by the user. Softshell [151] is a GPU programming model with multitasking support. It relies on developers explicitly declaring preemption points (similar to the *yield()* function in collaborative multitasking) or preempting on the thread block boundary to enable the scheduling of tasks.

In [3], 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. 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.

3.2 Virtual Memory

Memory can easily become a scarce resource in multiprogrammed systems due to the additional pressure from multiple processes. Hence, it is important that the OS manages it fairly and prevents starvation of any process. Similarly, the demand paging can cause performance bottlenecks and the OS must try to minimize the impact and maintain the high system throughput. Some of the classic solutions include scheduling other ready processes while on a page fault, and page prefetching and page stealing optimizations.

3.2.1 CPU Virtual Memory

Operating systems have advanced the use of virtual memory beyond providing the basic features of process isolation and memory swapping. Several techniques that rely on the virtual memory mechanisms have been adopted to improve performance or programmability of the system. Lazy allocation of physical memory is a commonly employed technique where no physical memory is assigned to a page until that page is written to for the first time. Lazy memory allocation thus improves the utilization of physical memory and avoids performing functions such as physical frame lookups, page table updates and page zeroing, unless it is really necessary. Copy-on-write (COW) is another related optimization, used to optimize the process cloning (i.e., fork() system call) [20] and IPC [1], among other things. COW defers copying the shared data, unless one of the copies needs to be changed. Linux kernel also implements an inverse operation to COW, called kernel same-page merging (KSM) [9]. The KSM finds duplicate pages and maps them to the same physical memory, which proved to be very useful for increasing the number of guest OS instances in virtualized systems.

Operating systems today expose protection violation handles to user level applications [50, 8], facilitating further innovative uses of virtual memory mechanisms. Examples include garbage collection [7], program checkpointing [99], transaction locking [134], and transactional memory [31].

Another common use of virtual memory mechanisms is providing a shared memory abstraction on a distributed memory machine. Some Distributed Shared Memory (DSM) use memory protection hardware to detect accesses to the shared data. Coherence operations, such are page replication and invalidation, are then performed to migrate the page to the requesting node and allow accessing the data locally. Both OS level [51, 39] and user level library [98, 28, 17, 81] implementations have been proposed in the past.

3.2.2 GPU Virtual Memory

Pichai et al. [128] and Power et al. [129] have recently conducted simulation studies of GPU MMUs. Typical of modern general purpose processors [72], they studied TLBs accessed before or in parallel with the L1 cache, and concluded that a certain degree of concurrency in both TLB access and page table walking needs to be maintained, or the performance will suffer due to large working set and massive concurrency of GPU cores. Vesely et al. [164] studied the performance of virtual memory on the AMD APU, a fused CPU-GPU SoC, and proposed directions for further improvements. Among other things, they observed that faults caused by the GPU have much higher la-

tency and much lower scalability of handling than faults caused by the CPU.

Saha et al. [140] have proposed a programming model for heterogeneous x86 based systems which have a host Intel x86 CPU and the accelerator card carrying the Intel Larrabee x86 based processor. As part of the implementation, they had a software DSM between the host and accelerator memories, using page faults on both host and accelerator to trigger data transfers between the two memories. Such implementation is possible, because Larrabee is effectively x86 CMP with in-order cores derived from Intel Pentium core [143]. Thus it comes with all the features expected from an x86 core, including exceptions (page faults) handling.

In order to circumvent the lack of page faulting capabilities on older GPU architectures Gelado et al. [55] have proposed to implement an asymmetric DSM (ADSM). It uses only CPU page faulting capabilities to track the state of shared pages (annotated by the programmer), but had to be conservative since no page faults could be used to detect data accesses on the GPU. Jablin et al. [71] extended this work with compiler support, so that regular *malloc* function can be used instead of the special allocator for the shared allocations.

Several solutions have been proposed to allow oversubscription of the GPU memory (i.e., allowing an application to use a data set larger than the GPU physical memory). Ji et al. [74] have proposed a software scheme that keeps track of used data via CPU runtime software and a GPU runtime software that acts as a software translation layer. Lee et al. [94] have proposed another software solution that uses compiler generated *inspector kernel* to extract the working set required for the workload partitions (thread blocks) and iteratively schedule as much blocks as can fit in the GPU memory, until all blocks have executed. These techniques that aim at enabling on-demand page migration and memory oversubscription for the GPU have been outdated, since the latest generation of GPUs (i.e., NVIDIA Pascal) supports them natively.

Shahar et al. [145] argue for page fault handling on GPUs, in order to enable their paging techniques and demonstrate performance improvement with applications that oversubscribe the GPU memory. Because of limitations of the current hardware, they had to introduce a software layer that performs address translation. Additional complexity and runtime cost of the software address translation can be avoided by introducing the exception handling mechanism on the GPU.

3.3 Exception Handling

High performance processor implementations that utilize pipelining or out-of-order execution (or both, in most cases) need to take special care, in order to provide the correct exception recovery and handling. The problem arises because the processor is overlapping execution of multiple instructions (i.e., pipelining) or executing instructions out of their program order (i.e., out-of-order execution). An older instruction in the program order might signal an exception after a younger instruction has finished and updated the program state. Thus, the processor needs to make sure that the execution can be correctly restarted after the exception cause has been fixed.

Additional complexity is introduced by the fact that all general purpose processors (following the von Neumann architecture) promise the sequential instruction execution interface to the programmer. This has the biggest impact on debugging capabilities, since it can make the state of the processor at the exception point different from what programmer expects (sequential execution in the program order), making it hard to debug. For this reason, modern general purpose CPUs support precise exceptions, which entails bringing the processor to the precise sate, before the exception is handled. This state is consistent with the state as if instructions would have been executed sequentially in program order, and execution stopped right before the faulting instruction. This way, it is enough to set the program counter to that of the faulting instruction and the execution can be restarted correctly.

3.3.1 Precise Exceptions

Early pipelined processors did not fully support precise exceptions. IBM System/360 model 91 [6] had a floating point unit (scheduled using Tomasulo's algorithm [158]) that was causing imprecise exceptions. Other high performance computers like the CDC 6600 [157] and Cray Research Cray-1 [139] vector machine implemented out-of-order commit, thus they did not support precise interrupts, and virtual memory for that matter.

In their seminal paper Smith and Pleszkun [147] discussed three mechanisms to recover from an exception in a precise manner: reorder buffer, history file, and future file. Another classical mechanism, the checkpoint-repair, was proposed by Hwu and Patt [68]. Sohi [149] and Moudgill et al. [112] discuss the use of the unified register file that holds architectural and speculative state. All of these proposals are focused on supporting exception recovery to precise architectural state by significantly increasing the storage to include architectural and speculative state (effectively register renaming). They can also be used as a misspredict recovery mechanism,

which is crucial because most of the modern general-purpose processors (the high performance ones, at least) employ speculation and dynamic instruction scheduling (and already doing renaming to improve performance). As a result, some variation of these mechanisms is usually used in commercial high performance processors [174, 82, 65, 156]

Implementing exceptions in vector processors has proven to be a challenging task, also. For that reason, many vector machines omitted the support for virtual memory [139, 162, 87]. IBM System/370 allowed only one vector instruction to be in-flight [25], which simplifies the support for exceptions, but limits the performance. Several vector processors / extensions have been proposed in academia which support precise [43] (using the reorder buffer approach) or restartable exceptions [90] (using the history file approach). Tarantula was a vector extension for the canceled Alpha EV8 processor [42]. It also supported precise exceptions, piggybacking on the host EV8 renaming capabilities.

There have been several proposals for exception handling in multi threaded processors [80, 178], that stall the faulting thread, while handling the exception in another execution thread. The goal of this approach is to start executing the handler code as soon as possible by avoiding the instruction flushing, state repairing and context switching latencies. These approaches still need the hardware support for precise exceptions to allow process restart, in case that the handler decides that context switch is needed, after all.

Implementing precise exceptions on the GPU through these classical methods is not deemed very practical [108]. Because of their throughput oriented nature, GPUs use very large register files already, and increasing this state significantly is not feasible. Furthermore, such increased complexity is justified in latency oriented machines (i.e., general-purpose processors) because it enables speculative execution. With GPUs relying on massive multi-threading for latency hiding, it is expected that additional performance achieved through speculation would not justify the high complexity.

3.3.2 Other Exception Handling Approaches

As a way of handling exceptions in exposed pipeline processors, Rudd [138] proposed redirecting the output of the pipelines into a replay buffer instead of feeding them to the write-back stage. After the exception is resolved, the replay is performed by draining the replay buffer (which now acts as the execute stage) to the write-back stage. Sentinel scheduling [104] is a compiler technique for detecting exceptions of speculatively scheduled instructions in VLIW processors. It is focused on correct signaling only, i.e., not restarting the process after the exception is handled. To tackle the restartability issue,

Bringmann et al. [24] has proposed the write-back suppression scheme as a method of recovery for speculatively scheduled instructions in VLIW processors. Both replay queue and write-back suppression schemes perform result buffering akin to that of the reorder buffer from [147].

Snapshotting the microarchitectural state of the machine, saving, and restoring it during the context switch is one of the ways to implement restartable exceptions. Reportedly, the CDC CYBER 200 designers chose this path, with the *invisible exchange* package [147], while the DEC VAX designers took a similar approach for the vector unit [60]. In case of GPUs this would entail directly manipulating the state of the warp scheduler, score-boards, SIMT divergence hardware, load/store unit that holds dozens of coalesced memory requests, etc. Besides the increased context, another drawback is that most of these units would need to be extended to support such state read/write operations. Torng et al. [159] proposed a scheme that does a snapshot of the issue queue and treats it as a part of the context. Their solution was developed for a single threaded processor, and as such, it is not suitable for our baseline multithreaded SM.

Hampton and Asanovic have proposed the software restart markers [60] as a way to support virtual memory in vector processors without implementing precise exceptions. Their architecture executes idempotent regions of the program (regions that can be executed multiple times with the same result) constructed by the compiler. Execution of the regions does not overlap, so in case of an exception, all the instructions can be squashed, and replayed later, from the start of the region. Kruijf et al. [40] proposed using idempotent region execution to provide speculation recovery and precise interrupts for scalar processors, while Menon et al. [108] applied idempotent processing to GPUs. Idempotent processing approach on GPUs introduces runtime overheads due to the additional instructions (generated by the compiler) that perform register spills that perform state preservation [108].

Chapter 4

Methodology

In this chapter we present the experimental methodology used in the dissertation. First, we describe our workloads and provide the reasoning behind our choice of benchmarks. We then describe the simulation infrastructure developed for the evaluation of our proposals.

4.1 Benchmarks

To evaluate most of our proposals we use the CUDA implementation of Parboil benchmarks [154] developed by the IMPACT research group at University of Illinois at Urbana Champaign. Table 4.1 lists all Parboil benchmarks. The Parboil suite is chosen over other benchmark options for several reasons. First, we find that this suite covers more diverse computational patterns and has more heterogeneous behavior compared to other GPGPU benchmark suites. Furthermore, Parboil is now being standardized as part of the SPEC ACCEL benchmark [36]. Finally, our group teaches CUDA programming using those benchmarks, and thus we have a fairly good understanding of their behavior.

None of the standard CUDA benchmarks, including Parboil benchmarks, perform dynamic memory allocations from the kernel (i.e., device side malloc). In order to evaluate our proposals on local GPU page fault handling, we use the benchmarks distributed with *Halloc* CUDA dynamic memory allocator [2]. Table 4.2 lists halloc benchmarks.

Mnemonic	Benchmark
sgemm	Single precission dense matrix multiplication
cutcp	Coulombic potential calculation
histo	Histogram
sad	Sum of absolute differences
tpacf	Two-point angular correlation function
bfs	Breadth first graph search
mri-griddin	MRI cartesian gridding
stencil	3D stencil code
spmv	Sparse matrix vector multiplication
lbm	Lattice-Boltzman method simulation
mri-q	MRI non-cartesian Q matrix calculation

Table 4.1: Parboil benchmarks used for the evaluation.

Mnemonic	Benchmark
add-strings	Performs string concatenation
random-graph	Constructs a random graph
grid-points	Inserts points into a grid

Table 4.2: Halloc benchmarks used for the evaluation.

4.2 Simulators

For the purpose of this thesis, we have developed two simulators with different levels of abstractions, each one geared towards the task at hand. We evaluate our multitasking proposals using a higher level system simulator that simulates full execution of multiple CUDA applications. Our proposals on exception handling are evaluated using a detailed microarchitectural simulator that simulates the execution of only one CUDA kernel.

4.2.1 Full System

The full system simulator used in this dissertation is a trace-driven simulator based on the methodology of [44]. It models a multi-core CPU connected to a discrete GPU through a PCIe bus. The simulator performs a coarse-grained modeling of the CPU and fine grained modeling of the GPU, tracing the execution of our benchmarks on the real machine. We perform parametrized simulation of the PCIe bus.

Figure 4.1 shows the work workflow with our system simulator. Benchmarks are compiled with NVIDIA nvcc compiler, and run two times on a real machine, to gather CPU and GPU traces. 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. CPU

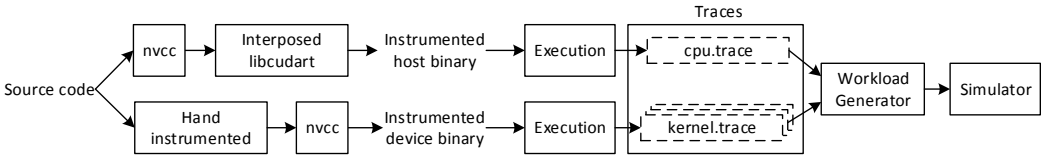


Figure 4.1: Full system simulation workflow.

traces consist of a starting and ending timestamp for each API call to CUDA runtime library, obtained using interposed libcudart. GPU traces consist of warp start and finish timestamps, obtained using hand instrumented kernels to gather the timing. Workload generator creates multiprogrammed workloads by randomly picking benchmarks that will be run at the same time.

4.2.2 Microarchitectural

The microarchitectural simulator used in this dissertation is a trace-driven cycle-level timing simulator. It consumes dynamic instruction and memory traces generated by an execution-driven functional simulator that emulates and traces all kernel invocations of a benchmark. The simulator models detailed SMs (except for the ideal instruction cache), cache hierarchy and MMU (TLBs and fill unit) attached to a constant latency and bandwidth DRAM model. We do not simulate texture caches and constant caches, thus benchmarks that use texture or constant memory spaces are modified to use global memory instead. The simulator uses an ISA designed to mimic modern GPU ISAs with all the distinguishing features such as a large unified register file, explicit management of the divergence stack, fused-multiply-add instruction, approximate complex math instructions, etc.

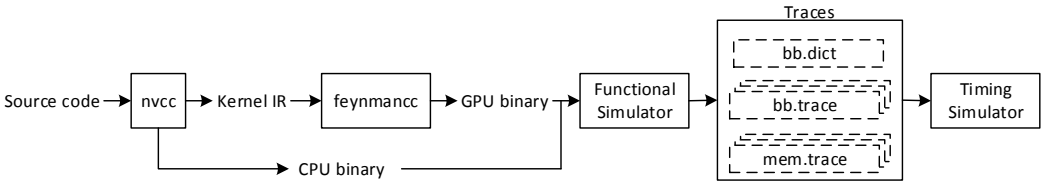


Figure 4.2: Microarchitectural simulation workflow.

Figure 4.2 shows the work workflow with our microarchitectural simulator. Benchmarks are compiled from their CUDA sources using NVIDIA nvcc compiler to generate the LLVM intermediate representation (IR) assembly for all the kernels. The kernel IR is then compiled to the target ISA by our compiler back-end (feynmancc), built on LLVM [93], and assembled before the functional simulation. The functional simulator emulates the kernel exe-

cution and generates the basic block dictionary (list of instructions for each basic block in the kernel code) and basic block[1] of and memory traces for each thread.

[1]Using the basic block dictionary with basic block traces is equivalent to using instruction traces, but also achieves the compression effect.

Chapter 5

Enabling Preemptive Multitasking

As we have previously discussed in Section 2, GPUs have been designed to maximize the performance of a single application, sacrificing the multitasking capabilities. Issues when sharing a GPU, such as priority inversion and no fairness, have already been noticed by operating systems [137, 77, 78, 125] and real-time [76, 14] research communities. Moreover, with the integration of programmable GPUs into mobile SoCs [141, 10] and consumer CPUs [5, 69], as well as increased deployment in the cloud and datacenter [63, 62], the demand for GPU sharing is likely to increase. In this chapter we argue that support for fine-grained sharing of GPUs must be implemented and provide our proposals on how to do so.

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 [3, 125] for the lack of a preemption mechanism in GPUs is the expected high overhead of saving and restoring the context of SMs (up to 256KB of register file and 48 KB of on-chip scratch-pad memory per SM), which can take up to 44μs in GK110, assuming the peak memory bandwidth. Compared to the context switch time of less than 1μs on modern CPUs, this might seem to be a prohibitively high overhead.

In this chapter we demonstrate that preemptive multitasking is a feasible approach to multiprogramming on GPUs. We design two preemption mechanisms with different effectiveness and implementation costs. One is similar to the mechanism in classic operating systems where the execution on SMs is

stopped, and their context is saved to the off-chip memory by a trap handling routine. 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 SMs, and draining them from currently running work. We show that both mechanisms can be used to provide improvements in system responsiveness and fairness at the expense of a small loss in throughput.

GPU sharing can be done at three different levels:

1. Time multiplexing the whole GPU, with all SMs belonging to one process.

2. Spatially sharing the GPU with an SM granularity.

3. Sharing the SM among multiple processes in the SMT fashion.

We argue that the spatial sharing is a natural choice for GPUs due to its simplicity and effectiveness. Our baseline GPU has private virtually tagged L1 and shared physically tagged L2 caches, as well as a large enough number of SMs, which makes the sharing with the SM granularity a good choice. Since the baseline GPU constraint of exclusive access to the execution engine prevents the spatial sharing out of the box, we propose further hardware extensions that allow the utilization of SMs, individually. These extensions enable different processes to concurrently execute GPU kernels on different sets of SMs.

Spatial sharing requires only slightly more complex logic than the time multiplexing of the whole GPU, yet provides benefits such is a decreased number of context switches, as well as a smaller context to save and restore on each switch. Even though sharing the SM in the SMT fashion [66, 144, 173, 161] could have a positive impact on the system throughput, it requires changes to the SM scheduling logic and first level of memory hierarchy. Furthermore, due to the dynamic partitioning of register file and shared memory, resource fragmentation could become a problem, leading to lower SM utilization. We thus leave the SM sharing for the future work.

With the preemption mechanisms in place and the sharing level chosen, a scheduler is required to complete the multitasking picture. As with the sharing, scheduling decisions can be done at multiple levels. Scheduling can be done in software, running on the CPU or GPU (e.g., adding general-purpose core designated to running operating system tasks). Alternatively, schedulers can be implemented in hardware as part of the existing SM scheduling logic (i.e., SM driver). In this chapter we present Dynamic Spatial Sharing (DSS), a hardware scheduling policy that dynamically partitions the resources (SMs)

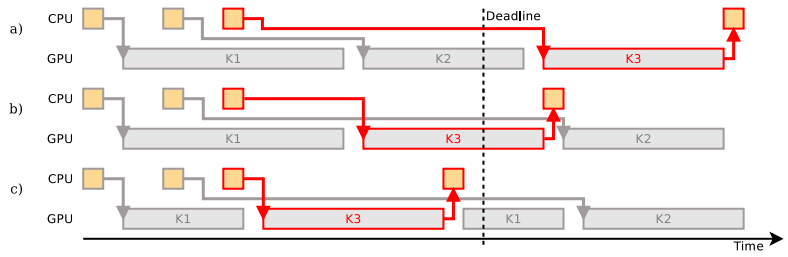


Figure 5.1: Execution of soft real-time application with (a) FCFS (current GPUs), (b) non-preemptive priority and (c) preemptive priority schedulers. K1 and K2 are low-priority kernels, while K3 is high-priority.

and assigns them to different processes. We opted for the hardware approach because of the significantly lower latency of the scheduler, which should simplify the scheduling decisions. Still, to make it flexible, we expose it to the SW, allowing the OS to influence the GPU scheduling by assigning priorities to processes.

5.1 Motivation

Let us consider an example that illustrates how scheduling support in current GPUs is not sufficient. Figure 5.1 shows a soft real-time (i.e., interactive) application competing for resources with some other applications. The execution on a modern GPU is shown in Figure 5.1a, 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 [76] or a modification to GPU command scheduler could allow priorities to be assigned to processes, resulting in the timeline shown in Figure 5.1b.

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 a kernel running on the GPU, we need a preemption mechanism. Figure 5.1c 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 work.

Preemptive execution on GPUs is not only useful to speed up high-priority tasks, it is also 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 [4, 59]. Preventing starvation when this kind of applications run in the multiprogrammed system is the second motivation of this work.

There is a widespread assumption that preemption in GPUs is not cost-effective due to the large cost of context switching [3, 125]. Even though it is clear that in some cases it is necessary [92], 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 work.

5.2 Architecture

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

5.2.1 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 [128]. 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 latter 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.

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 an 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.

5.2.2 Preemptive Kernel Execution

The scheduling policy is always in charge of figuring out which kernel should be scheduled to run, as well as when and where (i.e., on which of the SMs). If there are no idle SMs in the system, it can use a 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 5.2 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 5.2.3 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

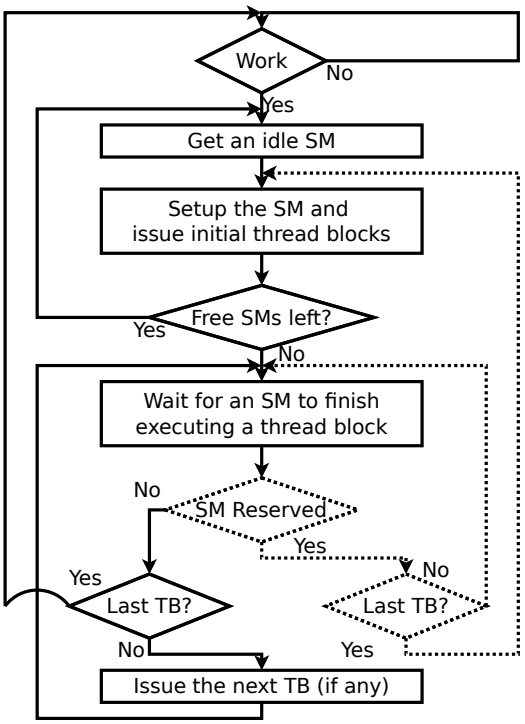


Figure 5.2: Operation of the SM driver. Dashed objects are proposed extensions.

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 software trap handler. 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 [119]. Since preemption is an asynchronous event, the problem of imprecise exceptions needs to be dealt with [147]. However, unlike page faults or similar instruction generated exceptions, handling of the preemption signal can be deferred to a more convenient time. Thus we opt for the simplest solution of draining the pipeline from all the in-flight instructions, before jumping to the trap routine[1]. The main drawback of the context switch mechanism is that during the context save and restore, thread

[1]This work was done before the GPU on-demand paging was implemented by the GPU vendors, and thus does not take it into account. The pipeline draining approach, even though correct, is inefficient when there are in-flight page faults caused by the preempted SM. We discuss our solutions to the problem of context switching an SM under faults in Section 6.

blocks do not progress with useful work, leading to a complete underutilization of the SM. This underutilization could be improved by using context minimization techniques, such as iGPU [108].

The second mechanism that we implement, **SM draining**, tries to avoid this underutilization by preempting the execution of the kernel 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 concurrent execution of 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 naturally, 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.

5.2.3 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 logic plugs into the framework and implements the logic of the concrete scheduling algorithm. Both the scheduling framework and scheduling policies are implemented in hardware to avoid the long latency of issuing commands to the GPU [77]. 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 on the critical path of the scheduling process. Thus, we do not introduce any additional OS noise.

Figure 5.3 shows the components of the scheduling framework. An exam-

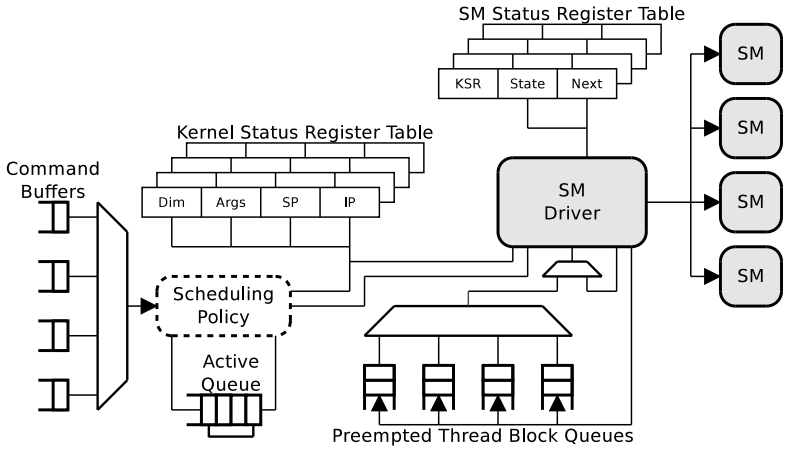


Figure 5.3: Scheduling framework. The rest of the execution engine (SM Driver and SMs) is shaded.

ple of interaction between the scheduling policy and the framework is given in Section 5.2.4. **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 entries 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 5.2.4 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 with 13 SMs. Command buffers, KSRT, SMST, and active queue together take less than 0.5KB of on-chip SRAM. PTBQs take 21KB of on-chip SRAM (for the whole GPU) and are present only if the context switch mechanism is implemented.

5.2.4 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 released by 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.

The scheduling algorithm can be invoked by a periodic timer or by some

Algorithm 1 Partitioning Algorithm

```
function PARTITIONING_PROCEDURE
    repeat
        idle_sm ← find_idle(SMSR.state)
        ksr_max ← max(KSR.count)
        ksr_min ← min(KSR.count)
        if KSR[ksr_max] = KSR[ksr_min] then
            return
        end if
        if idle_sm then
            SMSR[idle_sm].state ← running
            KSR[ksr_max].count ← KSR[ksr_max].count − 1
        else
            SMSR[ksr_min].state ← reserved
            SMSR[ksr_min].next ← ksr_max
            KSR[ksr_min].count ← KSR[ksr_min].count + 1
            KSR[ksr_max].count ← KSR[ksr_max].count − 1
        end if
    until KSR[ksr_max].count ≤ KSR[ksr_min].count + 1
end function
```

events occurring in the system. We choose to execute the algorithm only on the following two 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.

Compared to the CPUs, preempting the execution on the GPU can take a relatively large amount of time (with both mechanisms), yet the duration of a computation can be fairly short. In order to cope with the dynamic nature of the system and long latency operations, we allow the scheduler to change the decision of the next kernel (the kernel for which an SM is reserved) during the preemption of that SM.

CPU	**GPU**
Clock: 2.8 GHz	Clock: 706 MHz
Cores: 4	SMs: 13
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 5.1: Simulation parameters used in the experimental evaluation in Section 5.3. *Default configuration of the shared memory.

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 SMST and KRST. This operation takes as many cycles as SMs (13 in our configuration). Because of the simplicity of our DSS scheduler, we expect the area and energy overhead of the implementation to be negligible.

5.3 Evaluation

5.3.1 Methodology

We evaluate our proposals presented in this chapter using the full system simulator described in Section 4.2.1. The simulator parameters, based on an early version of Kepler GPU with 13 SMs (K20c) and Intel Core i7 930 CPU that were used to obtain traces, are provided in Table 5.1[2]. We assume the FCFS scheduling inside the data transfer engine, unless otherwise noted.

We use ten, out of eleven, benchmarks from the Parboil benchmark suite [152] 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. 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

[2]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.

at least 3 times. We are using multiprogrammed workloads with 2, 4, 6, and 8 processes. Replaying shorter benchmarks provides even workload for the longer benchmarks. Replaying even the longest benchmarks provides different workload interleavings. Running the whole applications provides more realistic execution workloads, as opposed to running kernels only. Statistics are gathered only for the completed executions and then averaged. This methodology is based on [160] and [163]. We choose the input sets of the benchmarks (shown in square brackets in Table 5.2) 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.

All the metrics used for evaluation in this chapter are calculated as suggested by Eyerman et al. [45]. Metrics are calculated based on the performance (execution time) of applications run in isolation and run in the multiprogrammed workload.

- *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).

Table 5.2 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).

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	binning	1	2021.41	5188	1.56	0	4096	4	21.05	4.10	LONG	LONG
[small]	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 5.2: Statistics of all the kernels from benchmark applications used in the experimental evaluation.

5.3.2 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 them by implementing the simple *priority queues* scheduler. This scheduler (used in our example in Section 5.1) 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). For this experiment, the scheduling policy in the data transfer engine is always NPQ. 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 of the workload.

We measure the turnaround time of the prioritized application and in Figure 5.4 show the improvement of the application's NTT when prioritized over its non-prioritized execution. When using the non-preemptive 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 non-preemptive 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.

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.

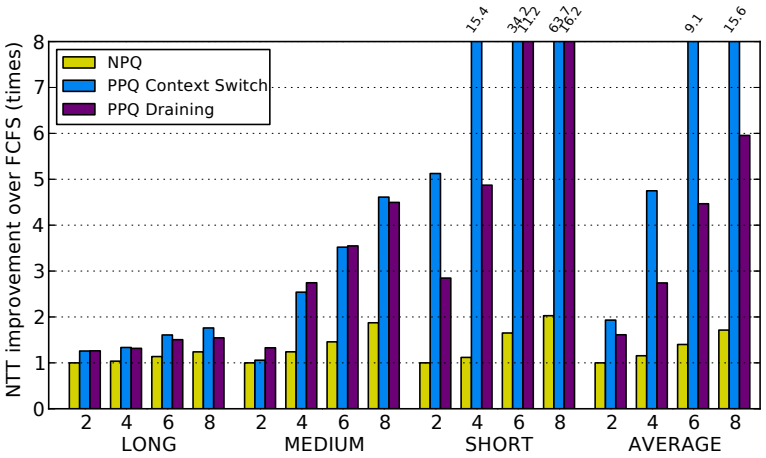
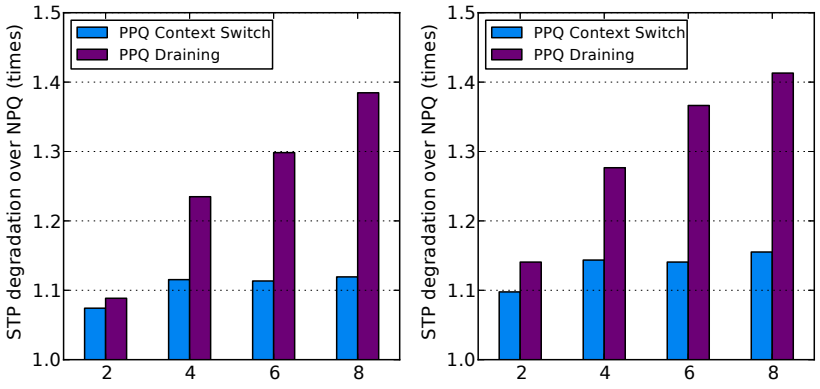


Figure 5.4: Turnaround time improvement of the high-priority process over its non-prioritized execution (higher is better). Showing workloads with 2 to 8 processes. Benchmarks in each group are listed in Table 5.2 as *Class 1*.

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 5.4, benchmarks are grouped by the average execution time of their kernels (Class 1 in Table 5.2). Both groups and execution times are listed in Table 5.2. 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 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



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

Figure 5.5: System throughput (STP) degradation when the prioritized kernel has exclusive and shared access to the execution engine (lower is better). Showing workloads with 2 to 8 processes.

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), increasing the chance of preempting a kernel with very long thread blocks.

5.3.3 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 5.5. We implement two slightly different PPQ schemes. The first scheme, shown in Figure 5.5a 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 thread

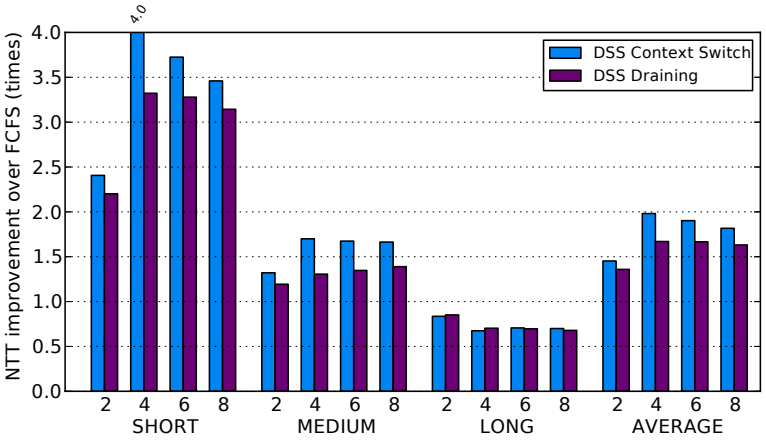


Figure 5.6: Turnaround time improvement with equal sharing (higher is better). Showing workloads with 2 to 8 processes. The list of benchmarks in each group is given in Table 5.2 as *Class 2*.

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 5.5b, results in higher overheads than the exclusive-access one, from Figure 5.5a.

5.3.4 Example Policy: Equal Spatial Sharing

We use the DSS scheduling policy described in Section 5.2.4 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

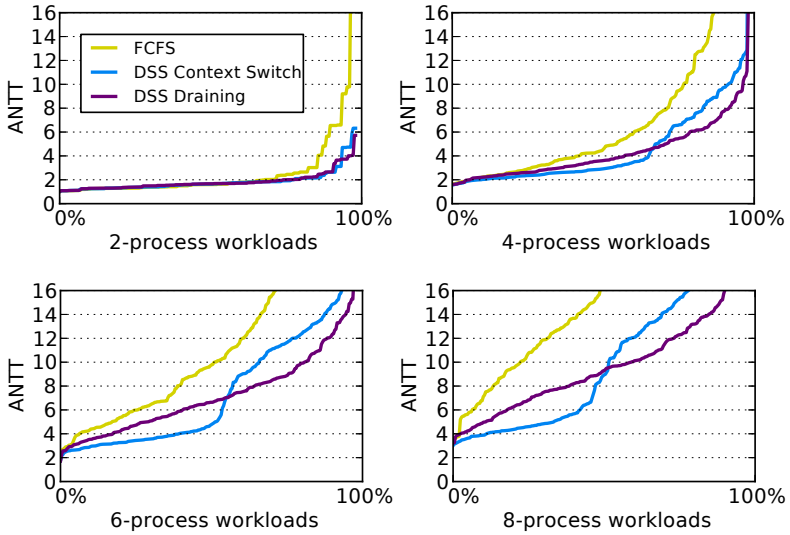


Figure 5.7: Average Normalized Turnaround Time (ANTT) for all the simulated workloads (lower is better), sorted by the increasing ANTT.

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.

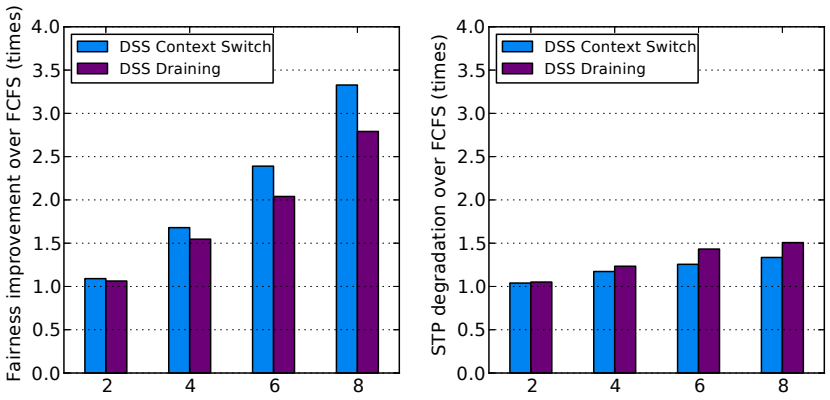
We first analyze the effects on the NTT of each benchmark application in all workloads and show their average improvements in Figure 5.6. Benchmarks applications are grouped by their execution time (Class 2 in Table 5.2). Short applications (<5ms), averaged in Figure 5.6 as *SHORT*, achieve the biggest 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 5.7 shows the ANTT achieved with the FCFS and DSS policies with both context switch and draining mechanisms for each workload. Workloads in Figure 5.7 are sorted by the increasing ANTT (to form the smoother, readable lines). For workloads with 2 processes, equal sharing improves the ANTT significantly for about 20% of the simulated workloads. In the other 80% of 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 which 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 Section 5.3.2), 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*etc.). 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 5.8a. 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 5.3.2. Like with the ANTT in Figure 5.7, 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 5.8b. The average STP degradation compared to the



(a) System fairness improvement (b) System throughput degradation

Figure 5.8: System fairness improvement (higher is better) and system throughput degradation (lower is better) with equal sharing. Showing workloads with 2 to 8 processes.

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 lower STP than the draining mechanism, this is not the case. Analyzing the throughput of individual workloads, a crosspoint similar to the one in Figure 5.7 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.

5.4 Summary and Concluding Remarks

Current GPUs do not provide the necessary mechanisms for the operating system 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 deployment of

GPUs in the cloud and data centers and integration of CPUs and GPUs in the same SoC, this problem will only escalate.

In this chapter we introduced 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 proposed two execution preemption mechanisms and the DSS scheduling policy that uses these mechanisms to implement dynamic spatial sharing of the SMs 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 showed that a dynamic scheduling policy that assigns different SMs to concurrently running kernels can greatly improve system-wide metrics such as fairness. Finally, we experimentally demonstrated that the wide-spread believe that context switching in GPUs is prohibitively expensive does not hold.

Chapter 6

Enabling Preemptible Exceptions

Recently, AMD and NVIDIA have commercialized discrete GPUs with support for automatic data transfers between CPU and GPU memories [92, 121]. On-demand page migration finally removes the need for explicit data transfers, improving programmability [54, 140, 55, 71], and enabling oversubscription of the GPU memory [78, 74, 94, 169]. However, demand paging requires using page faults on the GPU, a processor that does not support precise exceptions [148, 108]. To overcome this limitation, GPUs rely on offloading all the fault handling work to the CPU, while the faulted instruction on the GPU is stalled [175]. In this model the SM is not even aware of the exception (i.e., treated as a very long TLB miss). This is in contrast to CPUs, where exceptions trigger the execution of handling code on the core that raised the exception. The exception handler is then capable of saving the context of the faulting thread and restore its execution after the exception condition is resolved.

Preemptible exceptions are widely used in modern systems. For instance, both lazy memory allocation and on-demand paging are implemented in most modern operating systems using page faults. Whenever a thread causes a page fault, the operating system exception handler checks if the faulting virtual address corresponds to a memory page that has never been allocated (lazy allocation) or has been swapped out to disk (on-demand paging). In both cases, the operating system first saves the context of the running thread and switches into the fault handling routine. In case of lazy memory allocation, the OS needs to find an available physical page, updates the page table to reflect the new mapping, and restores the execution of the thread. On-demand paging requires the OS to bring the swapped out page from

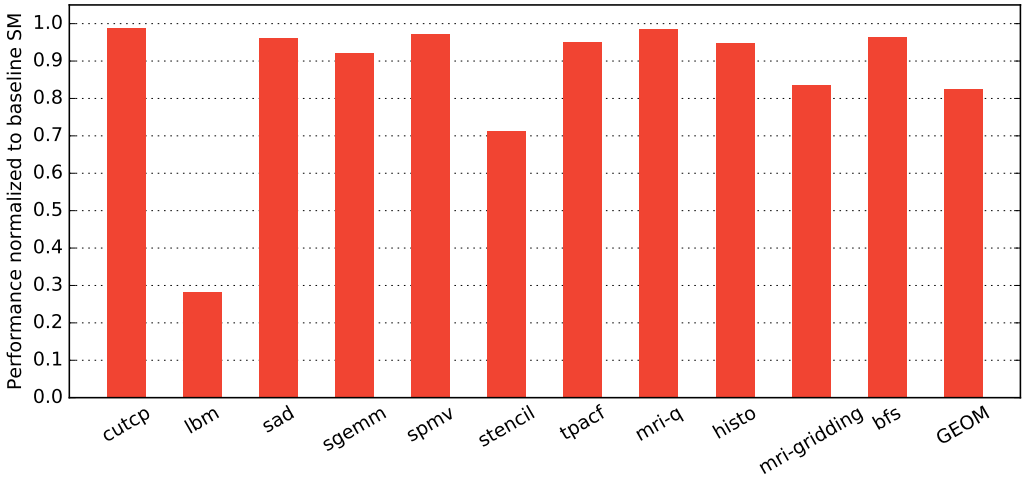


Figure 6.1: Performance of in-order issue and in-order commit cores, normalized to the baseline SM.

disk, which is a very long latency operation. Instead of waiting for the fault to be resolved, the OS typically schedules some different thread to run, to maximize the system throughput.

Besides ensuring the preemptability of exceptions, most modern CPUs also guarantee the architectural state to be *precise* when an exception happens. Precise exceptions support allows a very clean context switch and later restart of the faulting process. However, supporting precise exceptions on a modern GPU seems unfeasible. One option is to use simple in-order execution cores (in-order issue and in-order commit) that support precise exceptions by design. However, as Figure 6.1 shows, that option does not provide enough performance to meet the performance expected from modern GPUs. Another option is to implement in-order commit, similar to that of modern CPUs. However, this would increase the area and power consumed by each SM, and in turn decreasing the overall SM count and the overall system performance. Instead, we argue for the third option of supporting preemptible exceptions, whose architectural state is not precise, as a mean to provide the functionality needed by the system software (i.e., context switching) without harming the GPU performance.

In this chapter we present three alternative preemptible exceptions implementations on a modern GPU with varying performance-complexity trade-offs. We tailor our designs to support preemptible page faults only, because that is the main use of preemptible GPU exceptions today. Still, our designs can be extended to support other types of exceptions with an analogous approach, treating other potentially faulting instructions as we treat

the memory instructions in our proposals. The simplest approach achieves preemptible faults by introducing limitations to the execution model. As such, it does not require any additional hardware structures, but it does result in decreased performance. In our second approach, we relax some of the execution constraint with a mechanism that collects non-committed instructions for later replay. The most comprehensive solution introduces additional hardware structures and increases the area of the GPU core due to extra storage. With sufficient amount of resources, it can completely preserve the performance of the baseline GPU pipeline.

We further explore two use cases that aim to improve the system throughput thanks to the ability to context switch under a fault. In the first use case, we context switch out faulting threads during page faults, and context switch in new threads to execute while the fault is being resolved. We aim to hide the latency of the page migration that caused the fault by finding other work that the GPU can execute in the meantime. In the second use case, we handle page faults to non-committed physical memory (i.e., lazy memory allocation) on the GPU itself, instead of off-loading it to the CPU. The GPU code runs its own physical memory allocator, which reserves the required memory and fixes the GPU page table without interrupting the CPU and occupying the system interconnect.

6.1 Motivation

Exceptions are, by their very nature, events that happen rarely during the life of a thread and, thus, have a minor impact on the system performance. However, the massive amount of concurrent threads (e.g., 32768 in our baseline GPU) running on a GPU make such a rare event at the thread level to become quite common at the GPU level. For instance, on-demand page migration [121] can cause individual threads to trigger multiple page faults in a single instruction. Having in mind the concurrency level of the GPU, it is fairly common for a kernel to trigger dozens of concurrent page faults. We use this page fault handing in the GPU as the driving example to motivate our work.

Both signaling and data transfers between CPU and GPU are performed over the system interconnect which has a serializing effect on the handling. During this time, instructions that faulted cannot make any progress. Moreover, it is very likely that other warps (in the same, or different SM) are also trying to access one of the pending pages, and therefore are stalled as well, resulting in a severely underutilized system.

We demonstrate these issues on the execution of BFS benchmark with

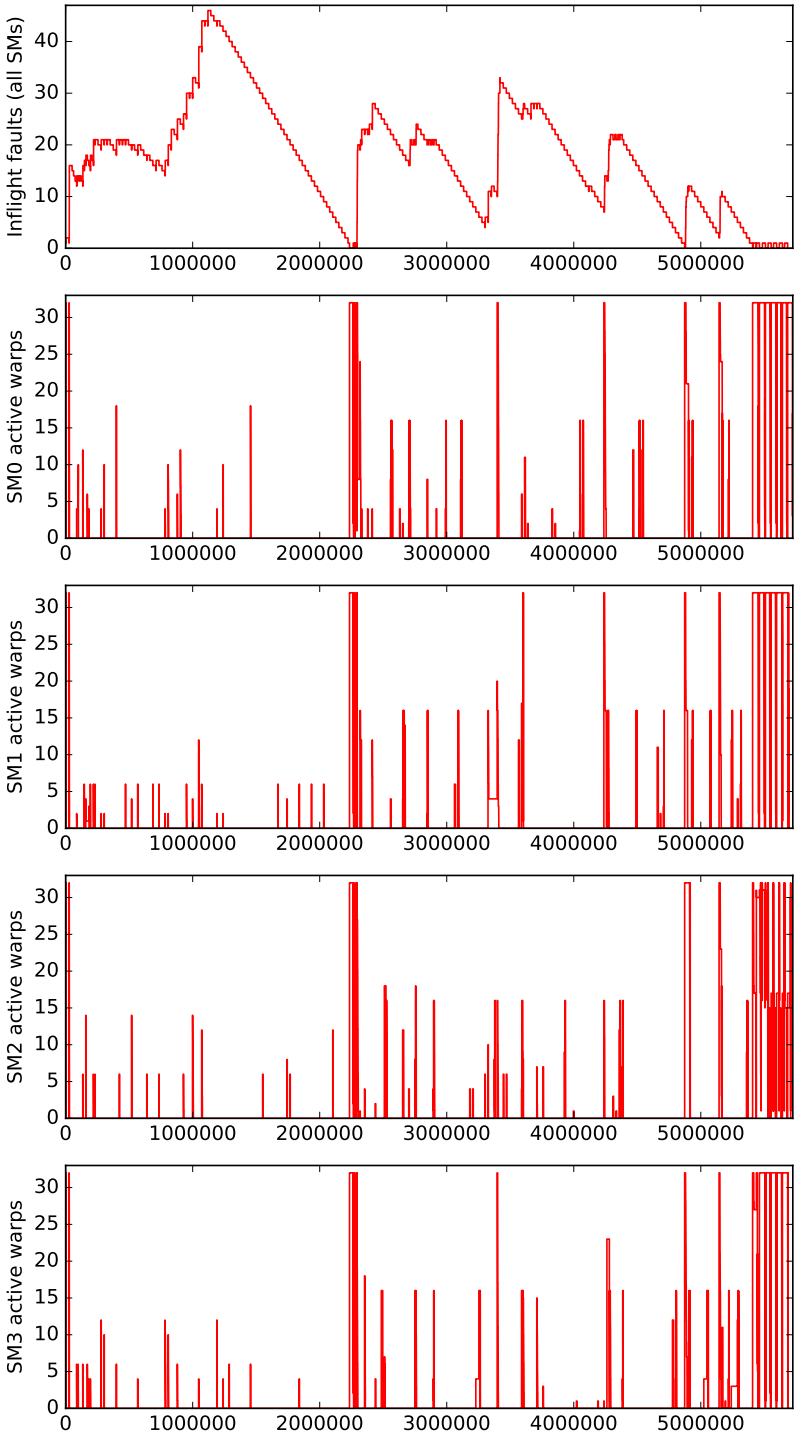


Figure 6.2: Number of in-flight faults (top timeline) and number of active warps (showing only four SMs) for an execution of the BFS benchmark (showing cycles on the x axis).

on-demand paging in Figure 6.2. The top timeline is showing the number of unique in-flight faults for the whole GPU and their bursty nature. The GPU tends to fault in spikes, after which migrations are resolved (one by one) until the next spike. The four timelines at the bottom are showing the number of active (non faulted) warps on four SMs (the behavior is similar for the remaining 8 SMs). Here, we can observe that most of the time (except for the very end of the execution) there is very few active warps. We can also observe that multiple warps can fault on the same page and do not make much progress until that fault is resolved (e.g., big spike in active warps around cycle 2.2 million, when most in-flight migrations have completed).

In an analogous CPU situation, after faulting on a page that needs to be brought from disk, operating systems typically context switch the process that faulted, and schedule some other process ready to run. Such context switching is not possible on GPUs, due to the stall on fault execution. This is an unfortunate situation because the GPU programming model encourages programmers to request the execution of a number of independent thread blocks much higher than the number of thread blocks the GPU can concurrently execute. Therefore, when a page fault happens it is very likely that there are pending thread blocks ready to be executed. Allowing context switching in the GPU during the page faults motivates us to enable preemptible faults in the GPU. We detail the design of such scheme in Section 6.4.1.

Low context switching latency is the key to achieving good multitasking performance on the GPU, as demonstrated by us in Section 5 and Park et al. in [126]. These studies were done for the legacy systems without support for on-demand page migration, which further complicates the matter. Due to the stall on fault execution, all the in-flight faults need to be serviced before the scheduler can perform the context switch to another process. The underutilization of the system while being context switches lowers the system throughput, while long latencies make it harder to improve responsiveness and system fairness. Therefore, stall on fault execution would possibly void any effort to improve system performance through scheduling. Facilitating the implementation of simpler and more effective schedulers for improved multiprogrammed system performance is another motivation of this work.

Finally, the inability to do the context switch under a fault stands in the way of handling the page faults on the GPU itself, because the forward progress of the fault handling routine cannot be guaranteed. In Section 6.4.2 we detail the design of a system that, instead of offloading the fault handling to the CPU, handles faults on the SMs that have faulted, in the cases where it is feasible to do so.

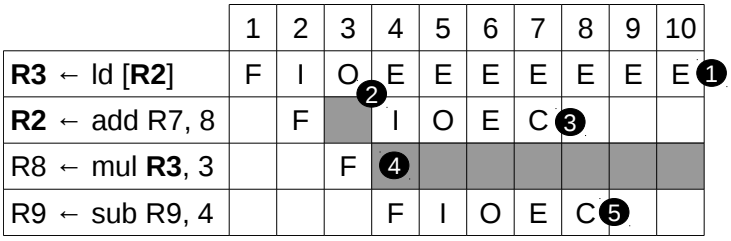


Figure 6.3: Timeline showing the culprits of non-preemptble faults: sparse replay and RAW on replay. All instructions are from the same warp. Stages are Fetch, Issue, OpRead, Execute and Commit.

6.2 Problem Statement

To understand why exactly does the baseline pipeline prevent preemtible faults, let us consider the simplified pipeline operation in Figure 6.3. The oldest instruction in program order, *ld*, goes through the fetch, issue, operand read stages and arrives to the global memory pipeline for execution. The global memory pipeline is deep, variable latency, and at some point (cycle 10 in the example) a page fault is detected. The second instruction, *add*, is fetched but stalled for one cycle, due to the *WAR* hazard with the *ld* instruction. When *ld* reads R2 at the cycle 3, it releases the score-board, clearing *add* for issue. At cycle 7, *add* will commit, having previously written the new value to the register R2. The third instruction, *mul* will be stalled after the fetch, because of the *RAW* hazard with *ld*, the producer of R3. Finally, the youngest instruction, *sub*, goes through all the stages, and commits at cycle 8.

When *ld* faults at cycle 10, the instruction cannot be just squashed and later replayed from a saved architectural state (context). The first problem is that after the fault we have to replay *ld* (the faulting instruction) and *mul*, but we must not replay *add* and *sub*, which have been already committed. However, no information is available in the pipeline to prevent the replay of *add* and *sub*. Note that this problem exists even with in-order issue pipelines, if for example two memory instructions fault but instructions in-between execute normally. We refer to this problem as **sparse replay**. The second problem is that the early source score-board release (implemented in our baseline pipeline) during the operand read stage allows *add* to write a new value to the register R2 at commit. Therefore, when we replay *ld*, the instruction reads the value in R2 produced by *add*, leading to incorrect execution of the program. We refer to this problem as **RAW on replay**.

6.3 Support for GPU Page Faults

In this section we present three different approaches to support preemtible faults on our baseline GPU architecture. The first approach (*warp disable*) treats memory instructions as code barriers. In case a memory instruction causes a fault, all younger instructions are committed and the faulting instruction is replayed after the fault is resolved. The second approach (*replay queue*) keeps in-flight memory instructions in a replay queue: they enter the queue when issued and exit the queue at commit. When a fault is detected, all in-flight instructions are squashed and the content of the replay queue is saved. After the fault is resolved, all the saved instructions are replayed. Finally, the third approach (*operand log*) logs the source operands of memory instructions during their execution. When replaying faulted instructions, the source operands are read from the log instead of the register file.

Each of these approaches presents a different trade-off between the amount of ILP the architecture can exploit and the additional hardware needed. *Warp disable* introduces no hardware modifications, but limits the amount of ILP exploited by the SM. *Operand log* exploits as much ILP as our baseline architecture, but requires the most additional hardware. *Replay queue* is an intermediate solution.

6.3.1 Approach 1: Warp Disable

This scheme addresses both sparse replay and RAW on replay problems defined in Section 6.2 by treating global memory instructions (i.e., the only instructions that can potentially page fault) as an instruction barrier. We enforce this by disabling the warp fetch once a global memory instruction is fetched and re-enabling it once the instruction commits. The execution of other warps is not affected and they can continue with the normal execution. By the time the instruction is ready to commit it had finished all the work, including the TLB accesses for all the active threads. Thus, at commit time we can guarantee that the memory instruction will not fault and will not need to be replayed. If the fault does occur, the limitation of the model has provided us with two benefits. First, we guarantee that only one of the warp in-flight instruction can fault. Second, we know that it is the last fetched and issued instruction for the warp that faulted. Hence, we only need squash and later replay that one faulted instruction.

The pipeline timing diagram in Figure 6.4 illustrates how *warp disable* works. If *ld* instruction finishes successfully, we can enable the fetch again (1) and let the younger *add*, *mul* and *sub* execute. However, if *ld* faults, the only other instructions potentially in the pipeline are older instructions that never

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
R3 ← ld [**R2**]	F	I	O	E	E	E	E	E	E	E	C							
R2 ← add R7, 8												F	I	O	E	C		
R8 ← mul **R3**, 3													F	I	O	E	C	
R9 ← sub R9, 4														F	I	O	E	C

Figure 6.4: Pipeline timing diagram with the warp disable approach. Global memory instruction disables the warp until it can be guaranteed that it will not fault.

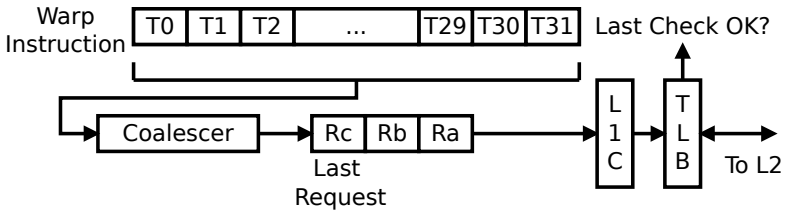


Figure 6.5: Last TLB check for a warp memory instruction: the earliest point in the pipeline where memory instruction is guaranteed not to cause a page fault.

cause a page fault. To recover from the fault, we squash the faulting instruction and drain all other in-flight instructions of the warp, before invoking the exception handler. To restart the execution, the exception handler restores the program counter to instruction that caused the exception. This way the faulting instruction is replayed once the handler has resolved the exception.

We can further optimize the performance of this scheme by realising that in all cases, we could enable the warp before the commit stage. Because a warp consists of 32 threads, one memory instruction of a warp can be accessing multiple pages at the same time. As shown in Figure 6.5, the instruction first goes through the coalescing unit that generates one memory request for each unique cache line accessed by the warp (part of the baseline SM). The earliest cycle where we can re-enable the warp so that it continues fetching instructions is right after the TLB check for the last generated request has completed successfully. The result of moving fetch-enable to the earliest cycle possible is letting other instructions enter the pipeline as soon as possible and therefore recovering some of the lost ILP.

The negative side of the warp disable scheme is that hinders the ILP achieved by the baseline SM by temporarily disabling the warp on a memory instruction. Since the SM is a throughput oriented processor that heavily relies on multi-threading to achieve high performance we can expect smaller performance impact than a similar technique would have on a CPU. Furthermore, since other instruction types cannot page fault, their execution is

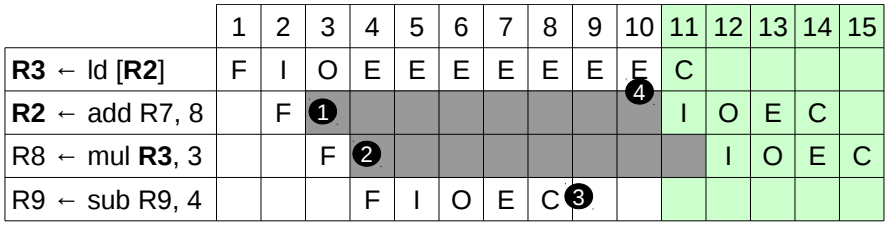


Figure 6.6: Pipeline timing diagram with the replay queue approach.

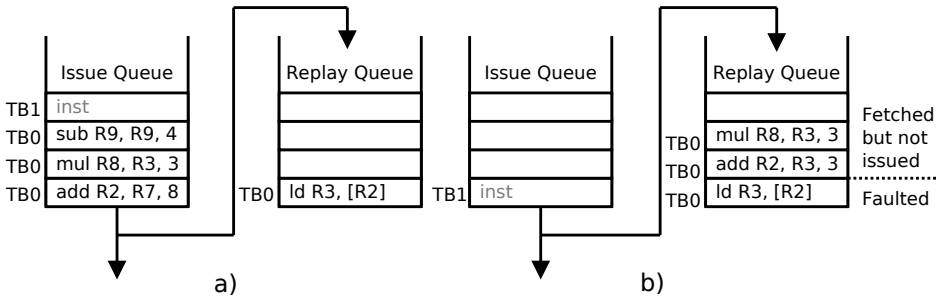


Figure 6.7: The snapshot of issue queue and replay queue after a) ld is issued and b) ld has faulted, and the rest drained.

unchanged from the original SM. The positive side is that we have enabled preemptible faults without requiring any additional hardware.

6.3.2 Approach 2: Replay Queue

The goal of this scheme is to remove the instruction barrier semantics imposed by the warp disable scheme, so the processor can exploit larger amounts of ILP. We first deal with the RAW on replay problem by releasing the source operands after the last TLB check has completed successfully, which only leaves the problem of the sparse replay open. Let us consider the pipeline timing diagram for our example program in Figure 6.6. *Add* is stalled (1) over a WAR hazard on register R2, while *mul* is stalled (2) over RAW hazard on register R3. *Sub* has no dependencies, so it gets issued and commits a few cycles later (3). If *ld* does not fault (4), execution continues normally (shaded green in Figure 6.6). Otherwise, if *ld* faults, we must replay *ld*, *add*, and *mul* when the fault is resolved, but the committed *sub* instruction must not be replayed.

To deal with the sparse replay problem, we introduce the replay queue next to the issue queue, as shown in Figure 6.7. Memory instructions (the only ones that can cause a page fault) are inserted in the replay queue on issue, and removed once they commit. In case of a fault, we first drain all

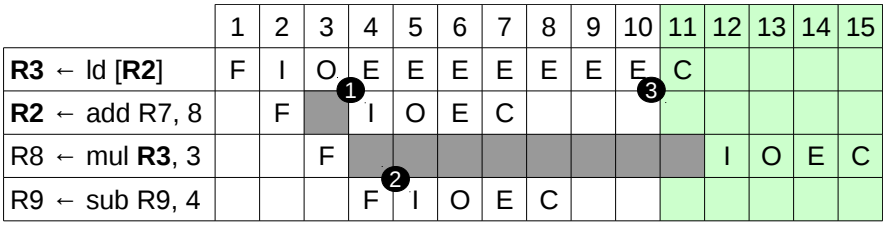


Figure 6.8: Pipeline timing diagram with the operand logging approach.

the non-faulted in-flight instructions, and then squash all the faulted ones. Once this is done, we need to drain the issue queue of the warp's fetched, but not yet issued instructions, and insert them to the replay queue also.

The instructions in the replay queue now become part of the context and need to be saved during a context switch. On context restore, the saved instructions are replayed first, before we continue with normal execution. Because the replay instructions are captured in the program order (relative to each other), the baseline issue logic can guarantee correct execution.

The negative side of the replay queue scheme is that it introduces additional hardware to the baseline SM and increases the complexity of the software that also needs to save the replay queue as part of the context. The positive side is that the replay queue is an unobtrusive addition to the pipeline that improves the ILP over the warp disable scheme by eliminating the barrier instruction semantics. Furthermore, the replay queue does not hold any data (i.e., registers) produced or consumed by the instructions, an important property bearing in mind that a warp instruction is basically a 32 wide SIMD instruction. As such, we assume that the logic needed to implement the replay queue is negligible compared to the rest of the SM.

6.3.3 Approach 3: Operand Log

Our final scheme is designed to complement the replay queue scheme. It improves the performance by removing the constraint of releasing the source operands after the last TLB check has completed successfully. Instead, we wish to release the score-board during the operand read stage, just like in the baseline SM. Let us consider the pipeline timing diagram for our example program in Figure 6.8. The WAR hazard on register R2 is removed in the cycle when *ld* reads the R2 value (1), and *add* gets issued in the next cycle. *mul* is stalled because of the RAW hazard on register R3, while *sub* gets issued (3). By the time the potential fault is raised (3), *add* and *sub* have committed and updated the values of registers R2 and R9. During the replay, *ld* would read the wrong (updated) value of R2 and load data from a wrong

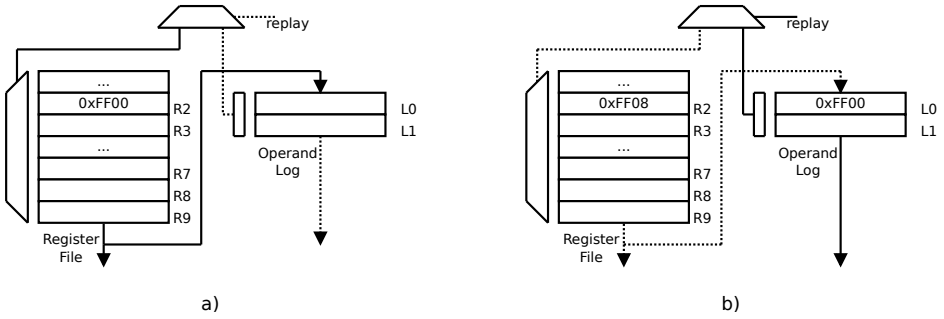


Figure 6.9: Design of the operand log with active path during a) first issue and b) replay of a faulted instruction.

address.

To handle the RAW on replay problem, we augment the SM with an operand log that holds the source data of in-flight global memory instructions (the only ones that can cause a page fault). Note that the operand log only eliminates the RAW on replay problem, so we still need the replay queue in order to handle the sparse replay problem. The allocation of the log entries is performed during issue of a memory instruction. During the operand read stage, data read from the register file is written to the log. To optimize the use of log space, load instructions take up only one log entry (source address), while store instructions take two (source data and destination address). Entries are released once we know the instruction is not faulting (after the last TLB check for the instruction has completed). On replay, the instruction accesses the log instead of the register file in order to read the input data. The log makes it safe to release the score-boards in the operand read stage, since there is now a copy of the source operands that is used on replay. Just like the contents of the replay queue, the log is now also part of the context and needs to be saved and restored during context switch. Since we need to provide the context switching at a thread block granularity, the log is partitioned so that each running block gets its own partition. Thus, kernels with lower number of active thread blocks (SM occupancy) will have higher number of log entires per thread block, and vice versa.

The negative side of the operand log scheme is that it introduces further hardware overheads (i.e., the log itself), and because of the increased context size it causes a higher context saving and restoring latency. The area overheads of this scheme are further explored in Section 6.5.2 where we analyze the performance as a function of the log size. The positive side is that now both culprits of preemptible faults (sparse replay and RAW on replay) are eliminated, and we can achieve both the performance of the baseline SM and

enable the preemptible faults (with a sufficiently large log).

6.4 Use Cases

In this section, we present two use cases that require preemptible page fault support in the GPU. In the first use case we context switch the SM when a page fault triggers an on-demand page migration from the CPU, which is a long latency operation. In the second use case, we rely on page faults to perform on-demand allocation of physical memory on the GPU.

6.4.1 Block Switching on Fault

In Section 2.1 we discussed how the baseline GPU observes page faults as very long latency memory accesses because of the stall-on-fault execution [175]. Although the SM can still issue other instructions that do not depend on the faulting memory access, oftentimes the pool of available independent instructions gets exhausted before the faulting memory access completes. Hence, the SM sits idle waiting for the fault to be resolved, underutilizing its hardware resources.

The preemptible exception support presented in this chapter opens the door to context switch the SM when a page fault happens, so that some other threads can use the SM resources while the fault is being resolved. The programming model of the GPU maximizes the likelihood of pending threads to be available for execution when a page fault happens. The number of thread blocks executing concurrently on the GPU depends on the available hardware resources of each specific GPU chip (i.e., depends on the GPU microarchitecture and number of SMs). For that reason, the GPU programming model encourages programmers to launch a much larger number of thread blocks than the GPU can concurrently execute. This oversubscription is key to preserve the scalability of the application as newer GPU chips are capable of concurrently executing larger number of thread blocks [120]. Hence, it is likely that when an instruction from a thread block causes a page fault, we can find pending thread blocks to execute.

The context switch in an SM has to be performed with a thread block granularity. The GPU programming model provides a notion of shared memory region that is accessible by all threads within a thread block. Unless we context switch the full thread block, we can not release its shared memory and schedule another thread block to run in its place.

We augment the SM with a thread block scheduler. This scheduler tracks the thread blocks running on the SM, as well as those thread blocks that

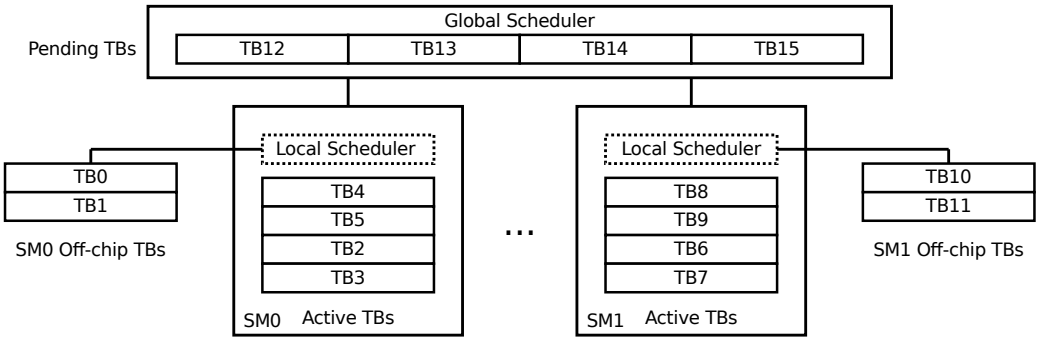


Figure 6.10: Block switching.

have been preempted. The fault handler can trigger a thread block context switch, depending on the warps waiting for a page fault to be resolved and thread blocks ready to execute. On a context switch, the handler notifies the scheduler, which first inspects if there are thread blocks switched out whose page faults have been resolved. If no thread block is found, the SM requests a new pending thread block to the global scheduler (i.e., the SM driver), in a similar way the SM currently does whenever a thread block finishes its execution. To prevent explosion of the off-chip memory space used for the thread block contexts, the local scheduler is allowed to bring only a limited number of extra blocks to the SM. Once this limit is reached, the SM cycles through the active and off-chip blocks only.

6.4.2 Local Handling of Faults

In the description of the GPU on-demand paging so far, we assumed that the page being accessed by the GPU was previously written by the CPU and, thus, resides in system memory. When the CPU has not previously initialized that memory, the faulting memory page is not present in system memory and, therefore, no migration is required. However, the CPU is still in charge of managing both CPU and GPU page tables and physical address spaces. Hence, the CPU still needs to allocate the physical memory on the GPU and update the GPU page table before the faulted instruction can continue. There are several cases that lead to such faults on the GPU, including the pages that hold the output data of the kernel or backing a memory allocation (i.e., malloc) performed by the kernel itself.

The preemptible exception support we introduce in this chapter allows us to run a page fault handler in the GPU that performs physical memory allocation and page table management. To prevent the user code from directly modifying system data structures, such as the GPU page table and the

physical memory allocator, we also assume two different privilege execution levels in the GPU: user and system. When an exception is detected, the SM switches to system mode and executes the corresponding exception handler. The page fault exception handler checks the faulting address and determines whether it corresponds to memory owned by the CPU, has no physical memory assigned, or is an invalid memory access. If the page is owned by the CPU, the handler sends the data migration request to the CPU. If the address is an invalid memory access, it requests the device driver running on the CPU to abort the kernel execution. Finally, if the address has not been assigned any physical memory yet, it marks the page as owned by the GPU (to prevent the CPU from allocating memory for this page), calls the GPU physical memory allocator to allocate a new physical memory page, updates the GPU page table, and restarts the execution.

This lazy physical memory allocation scheme allows many GPU threads to allocate physical memory in parallel, instead of serializing allocations through the CPU. We expect this scheme to improve the performance of those codes that rely on dynamic memory and have single-use output data structures.

6.5 Evaluation

6.5.1 Evaluation Methodology

The results in this chapter are obtained using the microarchitectural simulator described in Section 4.2.2. The simulator parameters, based on a hypothetic modern GPU, are given in Table 6.1.

The evaluation in this chapter is performed with Parboil benchmarks [152]. Additionally, some of the evaluation in Section 6.5.4 are also performed using benchmarks distributed with the Halloc CUDA dynamic memory allocator [2]. We simulate one kernel from each benchmark to its completion. If a benchmark has multiple kernels, we choose the main kernel for simulation. If the chosen kernel is lunched multiple times, we simulate the launch with the biggest amount of thread blocks. We choose the input sets of the benchmarks to minimize the simulation time, yet guarantee that there is enough thread blocks to occupy all SMs for several rounds of execution[1].

In experiments that do perform the on-demand page migration, all data is initially residing in the CPU memory. We assume 4KB [129, 128, 175] pages. Related work [175] and our own experiments indicate that some form of prefetching is necessary to make the on-demand migration competitive

[1] By a round of execution we refer to the amount of TBs to occupy the SM. For example, two rounds of TBs means two times more TBs than the SM can run concurrently.

SM	
Frequency	1GHz
Max TBs	16
Max Warps	64
Register File	256KB
Shared memory	32KB
Issue type	Nonspeculative out of order
Issue ways	2 instrcutions total from 1 or 2 warps
Issue queue	96 entry unified for all warps
Backend units	2 math, 1 special function, 1 ld/st, 1 branch
L1 cache	32KB / 4-way LRU / 128B line
	32 MSHRs / 40 clk latency / virtual
L1 TLB	32 entires / 8-way LRU
System	
Number of SMs	16
L2 cahce	2MB / 8-way LRU / 128B line
	70 clk latency / 512 MSHRs
L2 TLB	1024 entries / 8-way LRU
	128 MSHRs / 70 clk latency
Numer of PT walkers	64
Walking latency	500 clk
DRAM bandwidth	250 GB/s
DRAM latency	200 clk

Table 6.1: Simulation parameters used in the experimental evaluation in Section 6.5.

in performance. Thus, in experiments that include page migration (Section 6.5.3 and Section 6.5.4) we do handling with a 64KB granularity. This helps to amortize the high cost per fault caused by communication, system software and inefficient small data transfers.

We use the execution time in cycles as our performance metric in this chapter. The execution time is measured until all instructions of all thread blocks for a given kernel finish. Because the GPU has multiple cores and kernels can launch arbitrary number of TBs of arbitrary lengths, kernel execution leads to the inevitable tail effect which is in negligible in most cases (but not all).

6.5.2 The Performance Cost of Preemptible Faults

As we have discussed in Section 6.3, the different pipeline organizations that support preepmtible faults are by design expected to have different performance. We use as baseline a GPU architecture without preemptible exception support, and thus it represents the maximum performance our proposals can achieve. In Figure 6.11 we show the performance normalized to the baseline of two *warp disable* scheme variants described in Section 6.3.1 (warp disable until commit - *WD-commit* and warp disable until last TLB check - *WD-lastcheck*), alongside the performance of the *replay queue* scheme described in Section 6.3.2. We are here foremost interested in the performance of kernel execution that does not cause any faults (e.g., expert written program that uses explicit data management). Such execution will show us exactly how much performance loss is caused by our pipeline changes.

Comparing the geometric mean performance achieved across all benchmarks, we can see that WD-commit achieves only 85% of the baseline performance while WD-lastcheck achieves 91% of the baseline performance. The difference between these two schemes is related to how early in the pipeline we re-enable warp fetch. This results show that with a simple modification to the warp disable scheme (WD-lastcheck), we are able to recover significant amount of performance. The replay queue scheme is able to close this gap further, achieving 94% of the baseline performance. There are few cases where even the replay queue scheme is not sufficient, most notable with *lbm* that achieves only 59% of the baseline performance.

We have already explained in Section 6.3.3 that the *operand log* scheme can, by design, achieve the performance of the baseline SM if a sufficiently large log is used. In order to find out what is sufficiently large log size, we show the performance normalized to the baseline of operand log scheme with various log sizes in Figure 6.12. We start exploring log sizes from 8KB because it is the smallest log that guarantees that all thread blocks of a ker-

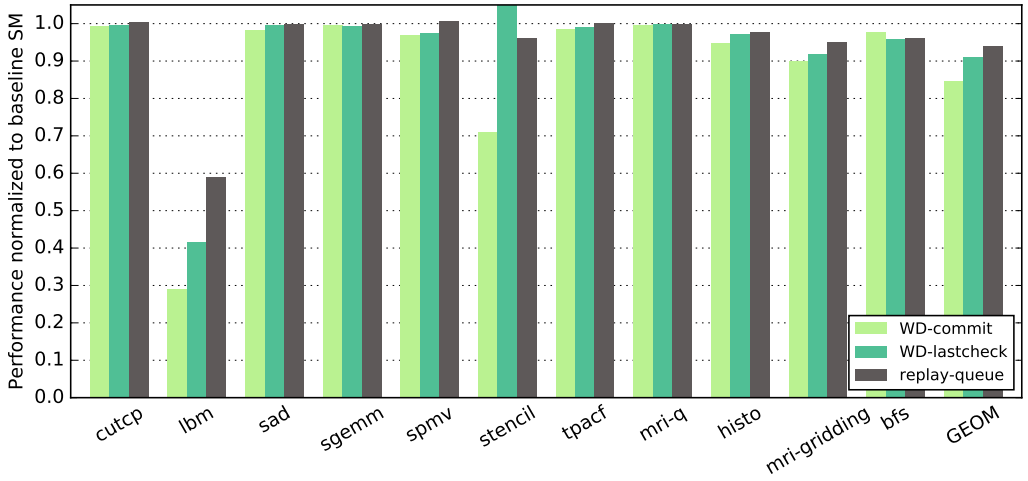


Figure 6.11: Performance of *warp disable* and *replay queue* pipeline organization that support preemptible faults, normalized to the baseline SM with stall on fault approach (higher is better).

nel with maximum occupancy (i.e., 16 in our baseline system) can execute concurrently. Indeed, the biggest amount of source data that an instruction needs to log is 512B, assuming 8B address and 8B data (times the 32 threads). Thus such log guarantees that each thread block can have at least one memory instruction in case that the SM occupancy is 16 thread blocks.

Comparing the geometric mean performance achieved across all benchmarks, we can see that even the 8KB log is capable of achieving 96.9% of the baseline performance, while 16KB log is capable of achieving 99.8%. To put these log sizes into perspective, the register file in the baseline SM is 256KB while the unified L1 data cache and scratch-pad memory is 64KB, making the 16KB log a 5% increase in the overall context size. This is not to be confused with 5% area increase, since the log overhead is amortized across the rest of the SM (frontend with all the scheduling logic and 32 wide SIMT backend) and the GPU itself (host interface, interconnects, L2 of memory hierarchy. etc.). The operand log scheme is the most effective with the *lbm* benchmark, where a 16KB log improves the performance from 67% to 96% of the baseline, compared to the replay queue scheme. Due to the low thread block occupancy *lbm* runs only 8 warps (one eighth of the total warps supported by the SM), resulting in the lowest IPC of all evaluated kernels. This demonstrates that the operand log scheme is effective across kernels with various levels of TLP, but it is the most compelling with difficult codes that exhibit insufficient parallelism to saturate a modern GPU. Such codes are easy to come across because GPUs are constantly increasing in size, exposing

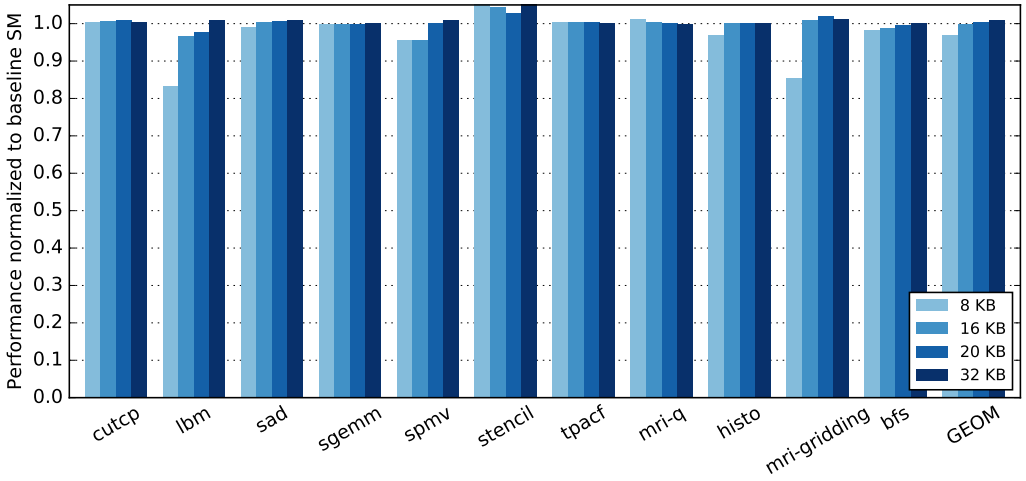


Figure 6.12: Performance of the operand log scheme with various log sizes normalized to the baseline SM (higher is better).

performance portability issues of legacy code.

6.5.3 Use Case 1: Thread Block Switching on Fault

In Section 6.4.1 we have described a thread block scheduling scheme that context switches faulted thread blocks and schedules ready thread blocks in their place. We show the performance of this scheme in Figure 6.13 for NVLink and PCI express 3.0 interconnects. We have measured several principal components that add up to the round trip latency of a page fault (page pinning, physical frame allocation and data transfer itself) and combined them with interconnect latencies to compute the cost of a page fault. We estimate the separate costs of faults for the case when there is a data transfer and for the case when only the allocation is necessary (pages not dirty in the CPU page table). These estimates are 12μs /10μs for NVLink and 25μs /12μs for PCIe, respectively. We have setup the local scheduler to allow a maximum of 4 extra thread blocks per SM, consider switching out the thread block once all the warps in a thread block are blocked (faulted or at a barrier) and consider switching in once all the faults for the thread block have been serviced. For each interconnect, the execution time is normalized to the on-demand paging implementation with stall on fault approach.

Starting from the NVLink, we can observe that several benchmarks show notable performance improvement. These are *sgemm* with 13.6%, *stencil* with 7.6% and *histo* with 9.9%. With the PCIe interconnect, the same benchmarks exhibit performance improvement, albeit a lower one (*histo* is

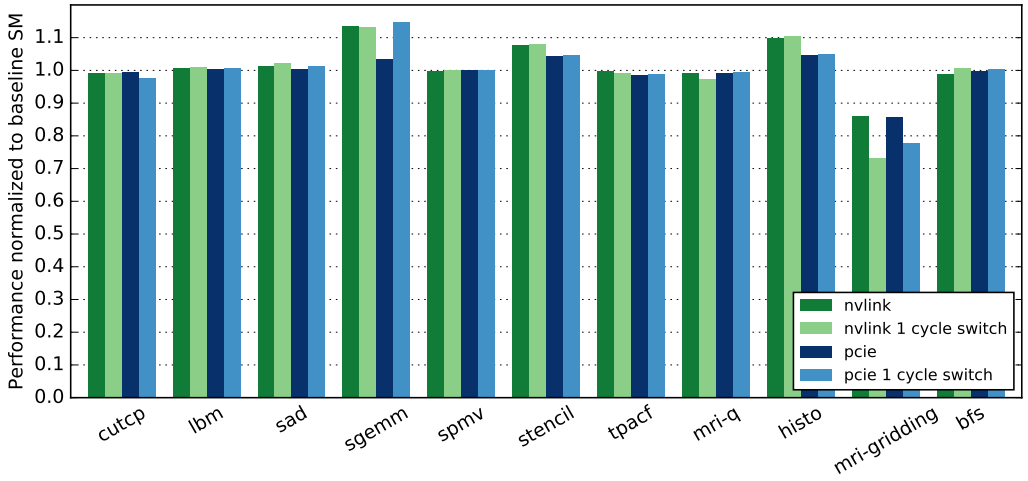


Figure 6.13: Performance improvement with thread block switching on a fault over baseline stall on fault approach. Showing NVLink and PCIe configurations with normal context switching and ideal 1 cycle context switching.

the highest with 4.6%). We also show the performance of this scheme with ideal context switching (1 cycle save and 1 cycle restore). Notice that there are only small improvements, mainly with *sgemm* in the PCIe configuration. Performance with ideal context switching demonstrates how our local scheduler is doing a good job on avoiding unnecessary context switching. It also shows that we captured most of the performance improvement that can be achieved by increasing the set of active thread blocks. We have studied this performance further, and found out that out of 11 benchmarks, 5 have either a very low or very high interconnect utilization. Thus, any scheme that tries to overlap computation with transfers is not going to improve performance on these benchmarks. From the rest of the benchmarks, three have either unfavorable access patterns such is faulting at the end of the block, or suffer from severe tail effect. Performance degradation of *mri-gridding* counters the improvement of other benchmarks, resulting in unchanged average performance.

It is important to note that no benchmark has a notable performance degradation except *mri-gridding* which achieves 86% of the original performance due to the massive load imbalance that the kernel exhibits. In this benchmark there is a two orders of magnitude difference in thread block execution time, owing to the different amount of work performed by different thread blocks. We have traced the execution and noticed that the original thread block distribution happens to spread the longest blocks more or less evenly across the SMs. Once context switching starts changing this order,

most SMs finish faster due improved latency hiding, but a minority of SMs get penalized with extra long blocks. Since we measure the execution time of the kernel as the cycle when the last thread block finishes, this ultimately leads to longer execution. This is further evident from *mri-gridding* performance with one cycle context switching being lower than with normal context switching.

6.5.4 Use Case 2: Local Handling of GPU Faults

In Section 6.4.2 we have described a fault handling scheme that allows handling page faults on the GPU itself, if the data transfer from the CPU is not required. The prime example of this are pages that are backing up memory allocations performed by the kernel itself (e.g., through the CUDA device version of malloc). Since Parboil kernels do not use device side malloc, we evaluate the performance using benchmarks that ship with Halloc CUDA dynamic memory allocator [2]. There is no page migrations in this experiment (i.e., explicit transfers), and all the page faults are caused by accesses to unmapped pages (first use). We prototyped the fault handler code and measured performance and scalability on a real GPU. We estimate the latency of the GPU handler to be 20μs, an order of magnitude more than the estimated latency of CPU handler (2μs) used in the rest of this section.

Figure 6.14 shows performance improvement with the geometric mean speedup of 80% and 98% for NVLink and PCIe respectively. The reason for such performance improvement, even with the 10x higher latency of the handler, lies in the number of concurrent page faults. The GPU is running many threads concurrently, and even though the frequency of fault in each thread is low (as the name *exceptions* imply, these are not very common events), the large working set of a GPU produces enough faults to overwhelm the system interconnect and the CPU who have to handle them one by one. In contrast, handling them on the GPU results in a clear throughput win, despite the longer latency of each fault.

In Figure 6.15 we show performance of handling the faults to output pages caused by Parboil kernels. These pages hold the output data of the kernel that is not used by the CPU until the execution of the kernel is finished. Benchmarks like *lbm* and *histo* show significant performance increase in both configurations. Contrary to the results in Section 6.5.3, this time the PCIe configuration shows overall bigger performance improvement than the NVLink configuration. Geometric mean across all benchmarks for NVlink is 4%, and for PCIe is 8%. Bigger performance improvement is seen with PCIe because the higher fault cost compared to the NVLink leads to higher contention of the system interconnect.

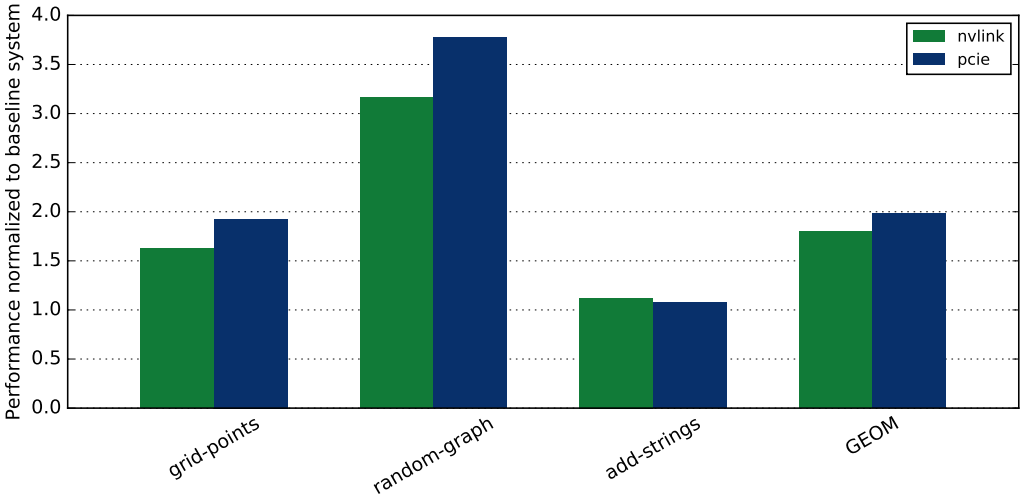


Figure 6.14: Performance improvement when handling faults to pages that are backing up dynamically allocated memory on GPU over baseline handling by the CPU.

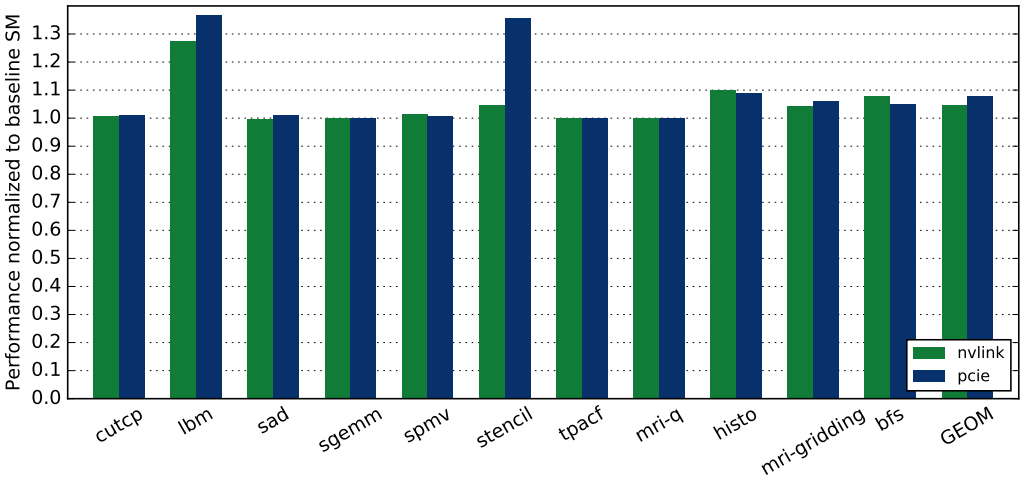


Figure 6.15: Performance improvement when handling faults to output pages on GPU over baseline handling by the CPU.

6.5.5 Summary and Concluding Remarks

In this chapter we have presented three different approaches to support exception on modern GPU architectures. There is a trade-off between the additional hardware required to add this support and the overhead introduced in the baseline architecture. We show how with a relatively small increase in the area, exceptions can be supported on a modern GPU while achieving 99.8% of the baseline performance.

We have also explored two potential use cases for exceptions on modern GPUs, context switching during page migrations and lazy physical memory allocations. Although context switching produces modest average performance improvements on our baseline system, it boosts the performance of two of the most common applications for GPUs: *sgemm* and *stencil*. This performance improvement is likely to benefit a large number of applications, ranging from physical simulations, to linear algebra solvers. The performance of lazy physical memory allocation for output data pages is also encouraging. However, being able to apply this technique to device side malloc greatly improves its usefulness. Without the ability of allocating physical memory on demand, current implementations of device side malloc are required to statically allocate large portions of GPU physical memory at the application load time. This effectively reduces the available memory on the GPU for applications and, thus, most programmers avoid using device side malloc. By allowing device side malloc to only consume those physical memory actually required, we expect this functionality to be more widely used.

Besides the use cases we have discussed, the exception support in the GPU we have presented in this chapter opens the door for further facilities provided by the operating system in the CPU to become available to GPU codes. This would increase the number of applications suitable to be accelerated by GPUs and, overall, improve the programmability of such systems.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

A constant shift towards visually immersive computing has put graphics processing units into most personal computing devices, from phones to workstations. The boom of deep learning techniques has further increased the GPU reach, making them indispensable in the data center, too. With evolving programmability and ever increasing performance, GPUs are often being used to accelerate the computationally intensive and data parallel code sections in many domains.

The increased use of GPUs with non-graphics applications is becoming a problem because sharing of the GPU is starting to take place, yet GPUs have practically no multiprogramming support. As the result, unresponsiveness is common in the interactive systems, if the GPU sharing occurs, and QoS hard to enforce in the data center.

In this dissertation we made initial necessary steps towards enabling multiprogramming on GPUs. We have first focused on improving the multitasking support by eliminating the constraint of non-preemptive GPU kernel execution and implemented a scheduler that controls the sharing of the GPU among multiple kernels. In an effort to further improve the multiprogramming support, we improve the GPU virtual memory functionality and performance through improved exceptions support.

We have introduced two GPU specific preemption mechanisms in this dissertation: the *draining* mechanism and the GPU tailored context switch mechanism. If only one them is to be implemented, we argue for the context switch mechanism because of its greater usability and on average higher

performance. However, we demonstrated the performance benefits of both and concluded that some form of a hybrid preemption mechanism could bring the best of both worlds. We have also introduced the Dynamic Spatial Sharing scheduling policy that uses these mechanisms to improve system responsiveness and fairness through spatial sharing of the GPU execution engine.

We have further identified the problem of non-preemptible exceptions as the main culprit behind inadequate virtual memory support in GPUs and proposed three solutions that would enable them. All three solutions provide the same functionality but at a different performance-complexity trade-off. We showcase the utility of preemptible faults by introducing two use cases that require context switches to be performed on a fault. The thread block context switching scheme improves performance of a kernel by finding other useful work to perform in place of faulted thread blocks and this way hides the long latency of handling the fault. We also demonstrate how local handling of GPU faults can improve system performance and introduce new programmability features.

The work presented in this dissertation outlines the initial architectural support that enables the true multiprogramming on GPUs. Even though further research could be conducted in order to improve the performance, efficiency, and software functionality, we believe that the proposals made in this dissertation would be a great first implementation.

7.2 Future Work

The work presented in this dissertation opens up new research lines in the field of GPU architecture and accelerator-aware operating systems. These new research lines are translated in the potential future work outlined in this section.

7.2.1 Fast and Efficient Preemption

We have observed that many workloads have a clear bias towards one of the two preemption mechanisms, performance-wise. Since the draining and context switch mechanisms can co-exist, we could introduce some logic to decide which preemption mechanism should be used and when. This can be a simple heuristic such as to start draining and then context switch, if preemption is not finished in some time frame, or something more complex like prediction or profile based policy.

Another direction that should be explored is minimizing the amount of

context that is saved, and subsequently restored on the context switch. Several optimizations can be made in this direction. One is that the compiler can emit the information about the live register range, therefore only the live registers need to be saved and restored. Doing a partial context save and restore by lazily saving pieces of the context could be another way to improve the latency of the context switch. For example, if the next scheduled kernel does not use shared memory, we can leave all the shared memory of the preempted kernel in place and only save it if some other kernel needs this space in the future.

7.2.2 Kernel Scheduler Design

We have opted to implement our DSS kernel scheduling policy in hardware because of its simplicity. We believe that hardware schedulers are great initial candidates, but they tend to be rigid and not very configurable by design. As such, we see future GPU schedulers moving some of the scheduling logic into software. Further research is necessary in order to decide how to split the scheduling responsibilities between hardware and software. Likewise, the question of where does the software portion of it run (on the CPU, general-purpose core on the GPU, or the SMs themselves) remains open.

The DSS policy that we introduced in this dissertation does not try to directly optimize any of the multiprogramming metrics. Instead, it takes the more general approach of spatial sharing to improve the fairness and turnaround time. We see the need to develop different scheduling policies that can be deployed in different environments, similar to the CPU schedulers. However, simply adopting the CPU scheduling policies is not appropriate because of different design constraints such as increased scheduling latency, longer preemption time and overall different nature of the workloads. More research is necessary on designing the new scheduling algorithms that could target a specific metric.

7.2.3 Fault Aware Scheduling

Previous research on GPU multitasking was not taking into the account the demand page migrations, since it predates the GPU paging implementations. In our experiments, we have observed the tendency of many warps to fault on the same page, making the GPU excessively underutilized. We have also noticed that some benchmarks are data-transfer bound and thus no amount of execution resources provided could improve their performance. As such, using the migration information (i.e., number of in-flight faults) while making

the scheduling decision could improve performance by giving the execution resources to kernels that can most benefit from them.

Furthermore, the classic optimization of context switching on a fault, in order to hide the migration latency could be applied also. Similar to our proposal of thread block switching on a fault, the scheduler could decide to vacate an underutilized SM and schedule some other process to run on it until the migrations are performed.

7.2.4 Heterogeneous Memory Management

Our proposals on local handling of GPU faults require some level of synchronization and communication with the CPU, in order to keep the CPU and GPU memory management structures coherent. This communication can be less involved when dealing with the lazy memory allocation than in the general fault handling case. Design of the heterogeneous memory allocator and coherence protocol for the CPU and GPU hosted allocator structures is still an open problem. Further research on the synchronization primitives (e.g., system wide atomic operations and memory fences) over system interconnects (e.g., PCI-Express and NVLink) is necessary to support deeper system integration of accelerators with good performance.

Appendix A

Publications

A.1 Thesis Related Publications

- *"Efficient Exception Handling Support for GPUs"*. Ivan Tanasic, Isaac Gelado, Marc Jorda, Eduard Ayguade, Nacho Navarro. Submitted for review to the 44th International Symposium on Computer Architecture (ISCA 2017). Toronto, Canada. June 2017.

- *"Enabling Preemptive Multiprogramming on GPUs"*. Ivan Tanasic, Isaac Gelado, Javier Cabezas, Álex Ramirez, Nacho Navarro, Mateo Valero. Published in Proceedings of the The 41st International Symposium on Computer Architecture (ISCA 2014). Minnesota, USA. June 2014.

- *"Hardware Support for GPU Multiprogramming"*. Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Poster in the NVIDIA GPU Technology Conference (GTC 2014). San Jose, USA. March 2014.

- *"CUsched: Multiprogrammed Workload Scheduling on GPU Architectures"*. Ivan Tanasic, Isaac Gelado, Javier Cabezas, Nacho Navarro, Alex Ramirez, and Mateo Valero. UPC-DAC-RR-CAP-2013-7 Technical Report. Barcelona, Spain. March 2013.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[2] Andrew V. Adinetz and Dirk Pleiter. Halloc: a high-throughput dynamic memory allocator for gpgpu architectures, 2014.

[3] Jacob T Adriaens, Katherine Compton, Nam Sung Kim, and Michael J Schulte. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[4] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149. ACM, 2009.

[5] AMD. AMD A-Series Processor-in-a-Box, 2012.

[6] DW Anderson, FJ Sparacio, and Robert M Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[7] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 11–20. ACM, 1988.

[8] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM, 1991.

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the Linux symposium*, pages 19–28, 2009.

[10] ARM. ARM Mali, 2012.

[11] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau, 2015.

[12] Jongmin Baek, Dawid Pająk, Kihwan Kim, Kari Pulli, and Marc Levoy. Wysiwyg computational photography via viewfinder editing. *ACM Transactions on Graphics (TOG)*, 32(6):198, 2013.

[13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[14] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 287–296. IEEE, 2012.

[15] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphic engine resource management. In *SPIE 2008*, volume 6818, page 68180O, 2008.

[16] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[17] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, pages 528–537, Feb 1993.

[18] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 557–558. ACM, 2010.

[19] David Blythe. Rise of the graphics processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.

[20] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[21] Thomas Bradley. Hyper-q example. *NVidia Corporation*, 2012.

[22] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *Micro, IEEE*, 32(2):28–37, 2012.

[23] Ian Bratt. HSA queueing. In *2013 IEEE Hot Chips 25 Symposium (HCS)*. IEEE, 2013.

[24] Roger A Bringmann, Scott A Mahlke, Richard E Hank, John C Gyllenhaal, and Wen-mei W Hwu. Speculative execution exception recovery using write-back suppression. In *Microarchitecture, 1993., Proceedings of the 26th Annual International Symposium on*, pages 214–223. IEEE, 1993.

[25] Werner Buchholz. The ibm system/370 vector architecture. *IBM systems journal*, 25(1):51–62, 1986.

[26] James R Bulpin and Ian Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*, pages 399–402, 2005.

[27] Javier Cabezas Rodríguez et al. On the programmability of multi-gpu computing systems. *Materia (s)*, 29:06–2015, 2015.

[28] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164. ACM, 1991.

[29] Kwang-Ting Cheng and Yi-Chu Wang. Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones. In *VLSI Design, Automation and Test, 2011 International Symposium on*, pages 1–4. IEEE, 2011.

[30] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[31] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358. ACM, 2006.

[32] Alan Cobham. Priority assignment in waiting line problems. *Journal of the Operations Research Society of America*, 2(1):70–76, 1954.

[33] Edward G Coffman Jr and Leonard Kleinrock. Computer scheduling methods and their countermeasures. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 11–21. ACM, 1968.

[34] Henry Cook, Miquel Moreto Planas, Sarah L Bird, Khanh Dao, David Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ISCA 2013: the 40th Annual International Symposium on Computer Architecture: conference proceedings: June 23-27, 2013: Tel-Aviv, Israel*, pages 308–319. ACM, 2013.

[35] Fernando J Corbató, Marjorie Merwin-Daggett, and Robert C Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 335–344. ACM, 1962.

[36] Standard Performance Evaluation Corporation. SPEC ACCELL benchmark, 2016.

[37] Foley Denis Danskin John. Pascal GPU with NVLink. In *Proceedings of the 28th annual symposium on High Performance Chips (HotChips)*. IEEE, 2016.

[38] Frederica Darema. The SPMD model: Past, present and future. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 1–1. Springer, 2001.

[39] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The clouds distributed operating system. *IEEE Computer*, 24(11):34–44, 1991.

[40] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 140–151. ACM, 2011.

[41] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 261–276. ACM, 1999.

[42] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, et al. Tarantula: a vector extension to the alpha architecture. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 281–292. IEEE, 2002.

[43] Roger Espasa, Mateo Valero, and James E Smith. Out-of-order vector architectures. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 160–170. IEEE Computer Society, 1997.

[44] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100. IEEE, 2010.

[45] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *Micro, IEEE*, 28(3):42–53, 2008.

[46] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[47] Alexandra Fedorova, Margo Seltzer, Christopher Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. pages 395–398, 2005.

[48] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. A non-work-conserving operating system scheduler for smt processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, volume 33, pages 10–17, 2006.

[49] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE Computer Society, 2007.

[50] Robert Fitzgerald and Richard F Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems (TOCS)*, 4(2):147–177, 1986.

[51] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 211–223. ACM, 1989.

[52] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[53] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2007.

[54] Isaac Gelado, John H Kelm, Shane Ryoo, Steven S Lumetta, Nacho Navarro, and Wen-mei W Hwu. Cuba: an architecture for efficient cpu/co-processor data communication. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 299–308. ACM, 2008.

[55] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM.

[56] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, pages 10–10. USENIX Association, 2012.

[57] Khronos OpenCL Working Group et al. The opencl specification. *Version*, 1(29):8, 2008.

[58] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009.

[59] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[60] Mark Hampton and Krste Asanovic. Implementing virtual memory in a vector processor with software restart markers. In *Proceedings*

of the 20th annual international conference on Supercomputing, pages 135–144. ACM, 2006.

[61] Mark Harris. Fast fluid dynamics simulation on the gpu. *GPU gems*, 1:637–665, 2004.

[62] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Trevor Mudge, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2015.

[63] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238. ACM, 2015.

[64] Felix Heide, Markus Steinberger, Yun-Ta Tsai, Mushfiqur Rouf, Dawid Pająk, Dikpal Reddy, Orazio Gallo, Jing Liu, Wolfgang Heidrich, Karen Egiazarian, et al. FlexISP: a flexible camera image processing framework. *ACM Transactions on Graphics (TOG)*, 33(6):231, 2014.

[65] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer, 2001.

[66] H Hirata, K Kimura, S Nagamine, Y Mochizuki, A Nishimura, Y Nakase, and T Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 136–145. IEEE, 1992.

[67] M Houston. Anatomy of amd's terascale graphics engine. *URL http://s08.idav.ucdavis.edu/houston-amd-terascale.pdf*, 2008.

[68] Wen-mei W. Hwu and Yale N Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th annual international symposium on Computer architecture*, pages 18–26. ACM, 1987.

[69] Intel. 4th generation Intel Core processors are here, 2012.

[70] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266. ACM, 2004.

[71] Thomas B Jablin, James A Jablin, Prakash Prabhu, Feng Liu, and David I August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2012.

[72] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[73] Davies Jem. Bifrost, the new GPU architecture and its initial implementation, Mali-G71. In *Proceedings of the 28th annual symposium on High Performance Chips (HotChips)*. IEEE, 2016.

[74] Feng Ji, Heshan Lin, and Xiaosong Ma. Rsvm: a region-based software virtual memory for gpu. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 269–278. IEEE, 2013.

[75] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[76] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011.

[77] Shinpei Kato, Karthik Lakshmanan, Ragunathan Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC'11)*, page 17, 2011.

[78] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX ATC*, volume 12, pages 37–37, 2012.

[79] Judy Kay and Piers Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[80] Stephen W Keckler, Andrea Chang, WSLS Chatterjee, and William J Dally. Concurrent event handling through multithreading. *Computers, IEEE Transactions on*, 48(9):903–916, 1999.

[81] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 10–10. USENIX Association, 1994.

[82] Richard E Kessler. The alpha 21264 microprocessor. *IEEE micro*, 19(2):24–36, 1999.

[83] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE, 2004.

[84] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8. IEEE, 2009.

[85] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122. ACM, 2004.

[86] David Kirk and Wen-mei Hwu. Programming massively parallel processors. 201software 0.

[87] Kenji Kitagawa, Satoru Tagaya, Yasuhiko Hagihara, and Yasushi Kanoh. A hardware overview of sx-6 and sx-7 supercomputer. *NEC research & development*, 44(1):2–7, 2003.

[88] Leonard Kleinrock. Analysis of a time-shared processor. *Naval research logistics quarterly*, 11(1):59–73, 1964.

[89] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE micro*, 28(3):54–66, 2008.

[90] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th Annual*

International Symposium on Computer Architecture, ISCA '03, pages 399–409. ACM, 2003.

[91] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[92] George Kyriazis. Heterogenious System Architecture: a technical review, 2012.

[93] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[94] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Vast: the illusion of a large memory space for gpus. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 443–454. ACM, 2014.

[95] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W Keckler, and Krste Asanovic. Exploring the design space of spmd divergence management on data-parallel architectures. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 101–113. IEEE, 2014.

[96] Samuel J Leffler. *The design and implementation of the 4.3 BSD UNIX operating system*. Addison Wesley, 1989.

[97] Adam Levinthal and Thomas Porter. Chap-a simd graphics processor. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 77–82. ACM, 1984.

[98] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[99] Kai Li, J Naughton, and James Plank. Concurrent real-time checkpoint for parallel programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming*, 1990.

[100] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742. IEEE, 2011.

[101] Erik Lindholm, Mark J Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM, 2001.

[102] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[103] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 1. ACM, 2016.

[104] Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling for vliw and superscalar processors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247. ACM, 1992.

[105] Micheal Mantor and Mike Houston. Amd graphic core next: Low power high performance graphics & parallel compute. *AMD Fusion Developer Summit*, 2011.

[106] Richard McDougall and Jim Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education, 2006.

[107] Soham Uday Mehta, Kihwan Kim, Dawid Pajak, Kari Pulli, Jan Kautz, and Ravi Ramamoorthi. Filtering environment illumination for interactive physically-based rendering in mixed reality. In *Eurographics Symposium on Rendering*, 2015.

[108] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. igpu: Exception support and speculative execution on gpus. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 72–83. IEEE, 2012.

[109] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–316. ACM, 2014.

[110] Ingo Molnar. Modular scheduler core and completely fair scheduler [cfs]. *Linux-Kernel mailing list*, 2007.

[111] John Montrym and Henry Moreton. The geforce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[112] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *26th annual international symposium on Microarchitecture*, pages 202–213. IEEE, 1993.

[113] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160. IEEE Computer Society, 2007.

[114] Pınar Muyan-Özçelik, Vladimir Glavtchev, Jeffrey M Ota, and John D Owens. A template-based approach for real-time speed-limit-sign recognition on an embedded system using gpu computing. In *Joint Pattern Recognition Symposium*, pages 162–171. Springer, 2010.

[115] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222. IEEE, 2006.

[116] Kyle J Nesbit, James E Smith, Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. Multicore resource management. *IEEE micro*, 28(3):6–16, 2008.

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

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

[119] NVIDIA. Programming guide - CUDA toolkit documentation, 2014.

[120] NVIDIA. CUDA C programming guide, 2016.

[121] NVIDIA. NVIDIA Tesla P100 white paper, 2016.

[122] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[123] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[124] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[125] Sreepathi Pai, Matthew 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*, pages 407–418. ACM, 2013.

[126] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–606. ACM, 2015.

[127] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. In *ACM Transactions on Graphics (TOG)*, volume 29, page 66. ACM, 2010.

[128] Bharath Pichai, Lisa Hsu, and Abhishek 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*, pages 743–758. ACM, 2014.

[129] Jason Power, Mark D Hill, and David A Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578. IEEE, 2014.

[130] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time computer vision with opencv. *Communications of the ACM*, 55(6):61–69, 2012.

[131] Kari Pulli, Wei-Chao Chen, Natasha Gelfand, Radek Grzeszczuk, Marius Tico, Ramakrishna Vedantham, Xianglin Wang, and Yingen Xiong.

Mobile visual computing. In *Ubiquitous Virtual Reality, 2009. International Symposium on*, pages 3–6. IEEE, 2009.

[132] Timothy J Purcell, Ian Buck, William R Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 703–712. ACM, 2002.

[133] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006.

[134] George Radin. The 801 minicomputer. *IBM Journal of Research and Development*, 27(3):237–246, 1983.

[135] Vignesh T Ravi, Michela Becchi, Gagan Agrawal, and Srimat 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*, pages 217–228. ACM, 2011.

[136] Javier Rodríguez-Navarro and Antonio Susín Sánchez. Non structured meshes for cloth gpu simulation using fem. In *3rd Workshop in Virtual Reality Interactions and Physical Simulation*, pages 1–7. EUROGRAPHICS, 2006.

[137] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[138] Kevin W Rudd. Efficient exception handling techniques for high-performance processor architectures. Technical report, Technical Report CSL-TR-97-732. Coordinated Science Laboratory, Stanford University, 1997.

[139] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[140] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and

Avi Mendelson. Programming model for a heterogeneous x86 platform. In *ACM Sigplan Notices*, volume 44, pages 431–440. ACM, 2009.

[141] Samsung. Samsung Exynos, 2012.

[142] Mark Segal and Kurt Akeley. The design of the opengl graphics interface. In *Silicon Graphics Computer Systems*, 1994.

[143] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, et al. Larrabee: a many-core x86 architecture for visual computing. In *ACM Transactions on Graphics (TOG)*, volume 27, page 18. ACM, 2008.

[144] Mauricio J Serrano, Roger Wood, and Mario Nemirovsky. A study on multistreamed superscalar processors. *University of California, Santa Barbara, Tech. Rep. Technical Report*, pages 93–05, 1993.

[145] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: a case for software address translation on gpus. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, 2016.

[146] Burton J Smith. Architecture and applications of the HEP multiprocessor computer system. In *25th Annual Technical Symposium*, pages 241–248. International Society for Optics and Photonics, 1982.

[147] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th annual International Symposium on Computer Architecture*, ISCA '85, pages 36–44, 1985.

[148] James E Smith and Andrew R Pleszkun. Implementing precise interrupts in pipelined processors. *Computers, IEEE Transactions on*, 37(5):562–573, 1988.

[149] Gurindar S Sohi et al. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE transactions on computers*, 39(3):349–359, 1990.

[150] David A Solomon and Helen Custer. *Inside Windows NT*. Microsoft Press, 1998.

[151] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: dynamic scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):161, 2012.

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

[153] John Stratton, Sam Stone, and Wen-mei Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. *LCPC 2008*, pages 16–30, 2008.

[154] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and Wen-mei W. Hwu. Parboil: a revised benchmark suite for scientific and commercial throughput computing. Technical report, 2012.

[155] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[156] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[157] James E Thornton. Design of a computer—the control data 6600. 1970.

[158] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[159] Hwa C. Torng and Martin Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, 1993.

[160] Nathan Tuck and Dean M Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2003, pages 26–34. IEEE, 2003.

[161] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, pages 392–403. ACM, 1995.

[162] Teruo Utsumi, Masayuki Ikeda, and Moriyuki Takamura. Architecture of the vpp500 parallel supercomputer. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 478–487. IEEE Computer Society Press, 1994.

[163] Javier Vera, Francisco J Cazorla, Alex Pajuelo, Oliverio J Santana, Enrique Fernandez, and Mateo Valero. FAME: Fairly measuring multithreaded architectures. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 305–316. IEEE, 2007.

[164] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, 2016.

[165] Jeffrey S Vetter, Richard Glassbrook, Jack Dongarra, Karsten Schwan, Bruce Loftis, Stephen McNally, Jeremy Meredith, James Rogers, Philip Roth, Kyle Spafford, et al. Keeneland: Bringing heterogeneous gpu computing to the computational science community. *Computing in Science and Engineering*, 13(5):90–95, 2011.

[166] Carl A Waldspurger and William E Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1. USENIX Association, 1994.

[167] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE, 2010.

[168] Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. Accelerating computer vision algorithms using opencl framework on the mobile GPU-a case study. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 2629–2633. IEEE, 2013.

[169] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. Gdm: Device memory management for gpgpu computing. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 533–545. ACM, 2014.

[170] W-D Weber and A Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Computer Architecture, 1989. The 16th Annual International Symposium on*, pages 273–280. IEEE.

[171] Florian Wende, Thomas Steinke, and Frank Cordes. Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture. *ZIB-Report 14-19 June 2014*, 2014.

[172] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU architecture. *Micro, IEEE*, 31(2):50–59, 2011.

[173] Wayne Yamamoto and Mario Nemirovsky. Increasing superscalar performance through multistreaming. In *Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, 1995.

[174] Kenneth C Yeager. The mips r10000 superscalar microprocessor. *IEEE micro*, 16(2):28–41, 1996.

[175] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.

[176] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *arXiv preprint arXiv:1303.5164*, 2013.

[177] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 129–142. ACM, 2010.

[178] Craig B Zilles, Joel S Emer, and Gurindar S Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 219–229. IEEE Computer Society, 1999.