Karlsruher Institut für Technologie

Virtualization and Migration with GPGPUs

Bachelorarbeit
von

cand. inform. Mathias Gottschlag

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuende Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand
	Dipl.-Inform. Jens Kehne

Bearbeitungszeit: 15. Februar 2013 – 14. Juni 2013

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 14. Juni 2013

Abstract

Recently, cloud computing providers have started to offer virtual machines specifically for high performance computing as a service (HPCaaS). The cloud computing providers usually employ virtualization as an abstraction layer between the application software and the underlying hardware. Virtualization allows flexible migration between physical systems, which is a requirement for many load balancing techniques. In the area of high performance computing, graphics processing units (GPUs) play a large role due to their high degree of parallelism and their high performance. If GPUs are used in a HPCaaS environment, they need to be integrated into the virtualization system. This GPU virtualization solution needs to provide near-native performance and needs to support migration. However, previous approaches to GPU virtualization either do not support migration or suffer from significant virtualization overhead.

In this work, we analyze existing GPU virtualization solutions. Based on this analysis, we present a GPU virtualization solution which exploits the features of the underlying GPU in order to significantly reduce virtualization overhead. In the proposed design, the virtual machine is given direct access to selected parts of the GPU so that it can directly execute programs on the GPU, without intervention of the hypervisor. We show that in the proposed design, the hypervisor is still able to transparently suspend and resume the GPU and can temporarily revoke direct access to it. This functionality can be used in order to implement transparent migration of the virtual machine as well as the GPU resources allocated to it. We propose a migration strategy which is designed to provide live migration for virtual machines with GPUs.

We evaluate the proposed GPU virtualization solution using a prototype built on top of Linux, the kernel-based virtual machine (KVM) and a NVidia GPU. Our experiments show that the proposed design results in very low virtualization overhead. Also, our experiments show that the proposed migration strategy provides low virtual machine downtime and keeps the overall impact of migration on performance low.

Deutsche Zusammenfassung

In der letzten Zeit haben Cloud Computing-Anbieter damit angefangen, virtuelle Machinen speziell für den Einsatz im Bereich des High Performance Computing anzubieten. Grundlage dafür ist üblicherweise Virtualisierung als Abstraktionsschicht zwischen den Anwendungen der Kunden und den darunterliegenden physischen Systemen. Diese Virtualisierung erlaubt das flexible Migrieren von virtuellen Maschinen zwischen physischen Systemen, das von vielen Load-Balancing-Techniken vorrausgesetzt wird. Gleichzeitig spielen im Bereich des High Performance Computing Grafikkarten (GPUs) eine immer größere Rolle, da diese durch den hohen Grad an Parallelität sehr hohe Rechenleistung bieten.Damit Grafikkarten im Bereich des High Performance Cloud Computing eingesetzt werden können, müssen diese in die Virtualisierungssysteme integriert werden. Lösungen für die Virtualisierung von GPUs dürfen dabei möglichst keine wesentlichen Leistungseinbußen verursachen. Zusätzlich muss die Virtualisierung transparente Migration unterstützen, da Migration für viele Techniken zum Beispiel im Bereich des Load Balancing erforderlich ist. Existierende Ansätze zur Virtualisierung von GPUs unterstützen jedoch entweder keine Migration oder verursachen signifikante Leistungseinbußen. Diese Arbeit untersucht die Eigenschaften von Grafikkarten sowie die Probleme existierender Virtualisierungslösungen für GPUs. Darauf aufbauend wird eine Virtualisierungslösung vorgestellt, die die durch Virtualisierung verursachten Leistungseinbußen wesentlich senkt, indem die Eigenschaften der GPU effizient ausgenutzt werden. In dem vorgestellten Entwurf wird dem Gastsystem direkten Zugriff auf Teile der GPU gegeben. Dadurch können aus der virtuellen Maschine heraus Speichertransfers initiiert und Programme auf der GPU gestartet werden, ohne dass die Kommunikation mit der GPU den Umweg über den Hypervisor gehen muss. Es wird gezeigt, dass in diesem Design der Hypervisor auch in der Lage ist, der virtuellen Maschine transparent den direkten Zugriff auf die Grafikkarte zu entziehen. Damit ist es möglich, die virtuelle Maschine transparent zu migrieren. Für die Migration des Grafikkartenzustandes zusammen mit der virtuellen Maschine wird eine Migrationsstrategie vorgestellt, die Live-Migration mit nur kurzer Unterbrechung der virtuellen Maschine ermöglicht.

Diese Arbeit demonstriert die Leistungsfähigkeit der vorgestellten Virtualisierungslösung anhand eines Prototypen basierend auf Linux, der Kernel-based Virtual Machine (KVM) sowie einer NVidia-GPU. Die Ergebnisse von Messungen des Virtualisierungsoverhead anhand mehrerer CUDA-Benchmarks zeigen, dass die Virtualisierungslösung nur geringe Leistungseinbußen verursacht. Zudem zeigen Experimente mit der vorgestellten Migrationslösung, dass diese nur geringe Unterbrechungen im Betrieb der virtuellen Maschine verursacht und die Leistungseinbußen bei während der Migration laufenden Programmen akzeptabel sind.

Contents

Chapter 1

Introduction

A recent development has been to provide more and more infrastructure as a service (IaaS). This form of cloud computing has been popular for example for databases, data storage and web servers. Cloud computing providers rely on virtualized environments to separate the software provided by the customer from the underlying hardware. This separation enables them to efficiently perform load balancing and lets them flexibly allocate resources based on the demand of the customer.

Recently, cloud computing providers have specifically started to provide processing power for high performance computing tasks. Due to the economics of scale, such offerings can cause significant cost reduction for customers who would otherwise have to purchase expensive hardware [17]. More and more, the systems for high performance computing are equipped with graphics processing units (GPUs), which, due to their large amount of simple processors, can be magnitudes faster than CPUs when used with highly parallel algorithms [16] [29]. However, if GPUs are to be used in an IaaS environment, they have to be integrated into virtual machines. The virtualization solution must have a performance which is close to native performance, because any overhead results in additional costs for the service provider. The virtualization solution also has to allow seamless migration of virtual machines as the service provider has to be able to move virtual machines between physical systems either for load balancing or for maintenance reasons.

In this thesis, we present a software solution which allows for both efficient and flexible use of GPUs in a virtualized environment. Existing virtualization solutions for GPUs are either very fast but lack flexibility, or result in a significant virtualization overhead [39].The use of these solutions in a cloud computing environment therefore is only partially viable.

There are two main sources of overhead in existing GPU virtualization solutions, which are a lower memory bandwidth for data transfers between the GPU and the virtual machine [39], and an increased latency when launching GPU programs [31]. Both stem from the fact that the virtual machine does not have access to large parts of the GPU. Instead, the virtual machine has to notify the hypervisor whenever it wants to send a command to the GPU. However, modern GPUs are designed to be used in a multitasking environment. They contain MMUs and command submission systems which can be used to provide direct access to most GPU functions to applications. This work uses these hardware features to pass this direct access into the virtual machine without breaking the isolation between the virtual machine and the host system. The same hardware features can also be used to transparently migrate the virtual machine to a different physical system. In this work, we analyze different migration algorithms and their viability for the use with this virtualization technique. Based on this analysis, we present a migration strategy which allows migration of virtual machines with GPGPU applications with low migration overhead and low virtual machine downtime.

The performance of the proposed design is evaluated using a set of GPGPU benchmarks on top of the CUDA GPGPU API. We measure the virtualization overhead caused by our GPU virtualization prototype. The results show that our proposed GPU virtualization design provides near-native performance. Also, we measure the downtime during migration which turns out to be below 200 milliseconds for our benchmarks, low enough that we consider the migration to be "live".

In the next section, we provide a background and discuss existing work on the area of virtualization of GPUs and migration, and present the advantages and drawbacks of other virtualization methods. In Chapter 3 we then present the features of modern GPUs which are relevant to the problem and discuss how they can be used to give userspace applications direct access to the GPU. In Chapter 4 we then discuss how they can be used to build a virtualization solution which fulfills the requirements described above. Chapter 5 contains some important details about the prototype implementation. Chapter 6 contains the methodology and result of the evaluation of the prototype. Finally, in Chapter 7 we summarize the results and present the conclusion of the thesis.

Chapter 2

Background

In this chapter, we will present some background information on high performance computing and GPUs. We will describe the problems which make virtualization of GPUs necessary and describe the basic requirements for virtualization systems with and without GPUs. Afterwards, we will present previous approaches to virtualization with GPUs and discuss their drawbacks. One aspect of the thesis is to demonstrate that migration with very low virtual machine downtimes is possible with GPUs. Therefore, we also will present some background about migration and live migration algorithms.

2.1 HPC as a Service

A recent development which motivates this thesis is that high performance computing applications are moved into the cloud. The high performance computing infrastructure is offered as a service (HPCaaS) by cloud computing providers. For the users of such services, reasons against their own infrastructure are lower initial costs and the flexible scaling of the infrastructure according to their needs [17]. Cloud computing providers operate very large centralized data centers which, due to economics of scale, can be more cost-efficient than many small or medium-size data centers. One important example for a HPCaaS platform is Amazon EC2 [1] which contains specialized offers for high performance computing.

HPCaaS is an example for Infrastructure as a Service (Iaas) which means that physical computing resources are dynamically allocated to users. This dynamic allocation requires some kind of abstraction from the underlying hardware, usually in the form of virtualization. For HPC applications, this virtualization must cause as little overhead as possible. The flexibility of the virtualization solution is also very important in a cloud computing environment. Especially, flexible migration of virtual machines has to be supported. Migration support allows cloud computing providers to react to demand through load balancing and server consolidation [45]. Additionally, GPGPU applications also benefit from load balancing, because the performance of a cluster is often limited by the node with the highest load [19].

2.2 Usage of GPUs for High Performance Computing

Graphics processing units (GPUs) are playing an increasingly important role in the area of high-performance computing due to their high performance compared to CPUs. In the following, we will first show how GPUs are used for general purpose computing. Then, we will describe existing GPU virtualization solutions and a set of metrics in order to evaluate the GPU virtualization solutions using the criteria from [23].

2.2.1 GPGPU

Initially, GPUs had been designed for acceleration of 2D and 3D graphics operations. The GPUs contained fixed function hardware blocks which supported basic drawing operations. Later, parts of the graphics pipeline like lighting and geometry transformation could be programmed through small GPU programs (shaders) [46]. Because such GPU programs are executed for every pixel on the screen or every point in the scene geometry, the GPU usually contains a large number of simple processors which execute the same program in parallel. With every generation of GPUs, the flexibility of the processors which execute these GPU programs was increased.

Modern GPUs are flexible enough that the processors can be used not only for graphics but also for other types of parallel data processing. These GPUs are also called general purpose graphics processing units (GPGPUs). Due to the large number of processors, GPUs perform significantly better for highly parallel workloads, with modern GPUs having a performance of one or more teraflops [16], far more than any single CPU [29].

The programming model for GPUs differs significantly from the common parallel programming models for CPUs. GPGPU APIs like CUDA or OpenCL follow the stream programming model. In this model, one GPU program or kernel is executed independently on all entries of the input data set [32]. The GPU has its own memory and access to system memory from the GPU is rather slow. Therefore, the typical GPGPU application first uploads some data into the GPU, then starts one or more small GPU programs through the GPGPU API and finally downloads the results to system memory [36].

2.2.2 GPU Virtualization

If the high performance of GPUs shall be usable for HPCaaS, the GPU has to be usable from within a virtual machine. In [23], Dowty and Sugerman have divided GPU virtualization techniques into either frontend or backend virtualization. Backend virtualization techniques grant the virtual machine direct access to the GPU. Frontend virtualization techniques place the virtualization boundary above the vendor-provided GPU API. Also, they have proposed a set of criteria for the evaluation of GPU virtualization solutions:

- **Performance:** The effect of the virtualization on the performance of GPU applications shall be as low as possible.
- **Fidelity:** All features of the GPU shall be available in the virtual machine.
- **Multiplexing:** Multiple virtual machines shall have concurrent access to one GPU.

- **Interposition:** Suspending and resuming as well as migration of the virtual machine shall be supported by the GPU virtualization solution.

We will use these criteria in this thesis for the evaluation of the proposed GPU virtualization architecture.

2.2.3 PCI Passthrough

Backend virtualization solutions can be further divided into fixed pass-through, where one virtual machine has exclusive access to the GPU, and mediated pass-through, where the host manages access to the GPU and multiple virtual machines can use the GPU [23]. The virtualization solution which is described in Chapter 4 is an example for mediated passthrough virtualization. We are not aware of any other implementation of mediated passthrough.

Fixed passthrough is more common, with PCI passthrough being the only widely-used example for fixed passthrough virtualization. PCI passthrough is a technique to pass complete PCI devices into a virtual machine by mapping its memory regions into the VM's address space. Additionally, memory access operations from the PCI device into system memory have to be restricted to memory owned by the virtual machine. This mapping of addresses in the physical address space of the virtual machine (as seen by the PCI device) onto physical addresses as seen by the host is done by an IOMMU.

Because the virtual machine has direct access to the GPU and the GPU has direct access to memory owned by the virtual machine, PCI passthrough provides near-native performance for GPUs [42]. Also, direct access to the GPU means that the GPU virtualization solution provides a high degree of fidelity. PCI passthrough is supported by most common hypervisors. However, because the VM assumes exclusive access to the GPU, PCI passthrough does not allow any form of multiplexing.

Another problem with PCI passthrough is that the migration of a virtual machine which uses PCI passthrough for GPUs is rather problematic. There are solutions which implement migration for PCI passthrough of network cards or InfiniBand adapters. These solutions replace the PCI device by a virtual shadow device in the guest during the migration [44]. This approach is not viable for GPUs because the initialization of GPUs is much more complex, which makes it rather difficult to transparently migrate virtual machines with GPGPU applications. Especially, public documentation about the internals of GPUs is too incomplete to allow the construction of such a migration solution.

Instead, in [44] one possible solution has been presented which uses virtual ACPI S3 events to make the guest operating system save its hardware state before the migration, and which uses the existing ACPI code to resume the virtual machine after the migration. However, this method has not been implemented for GPUs yet. Our experiments have shown that this technique does not work with current GPU drivers because the drivers expect the hardware to be in a different state after resuming from the virtual ACPI S3 state. Additionally, such a migration is neither live nor is it transparent to the guest. We also expect the migration overhead to be rather large because the hypervisor does not know which parts of the GPU state need to be transferred.

2.2.4 Frontend Virtualization

Examples for frontend virtualization of GPUs are vCUDA [38], gVirtuS [30], rCUDA [24] GViM [31] and VOCL [41]. All these virtualization solutions intercept calls to the GPGPU API (CUDA or OpenCL) in the virtual machine and pass them to the hypervisor. The hypervisor then uses the vendor-provided GPGPU runtime to execute the calls. Frontend virtualization generally provides much higher flexibility because there is a well defined API layer between the hypervisor and the guest. Because the host GPGPU API supports multiple GPGPU applications, multiplexing is trivial.

It is even possible to place the physical GPU on a different physical system and make the virtual machine provide a virtual GPU which communicates with the physical GPU interface through a network connection [24]. Therefore, it is also possible to migrate the GPU independently from the virtual machine. The resulting migration is transparent to the virtual machine, and live migration algorithms can be used to achieve low migration downtimes. A possible method for migrating a GPU has been described in [40]: First, the virtual GPU is stopped and the queue holding the commands from the virtual machine to the GPU is drained. Then the state of the physical GPU is transferred and the virtual GPU is updated to refer to the new physical GPU. Finally, the virtual GPU is resumed.

2.2.5 Drawbacks of Frontend Virtualization

While frontend virtualization is very flexible and supports multiplexing as well as interposition, it generally does not provide near-native performance. While no clear definition of "near-native" exists in literature, "near-native" means that the performance does not differ significantly from the performance without virtualization. We consider a GPU virtualization solution to have near-native performance if the virtualization overhead is below 5%. However, the overhead of the frontend virtualization solutions listed above is between 10% and 100%.

There are two main sources for virtualization overhead: First, most of the virtualization solutions have a lower memory throughput for transfers between system memory and GPU memory [39]. Some of this performance degradation is caused by additional copies in system ram while the data is passed to the hypervisor or the GPGPU API in the host. It has shown for example in GViM [31] that this overhead can be avoided by a zero-copy path from the virtual machine to the host. However, the data still needs to be transferred from system memory to GPU memory. This is done by the DMA engine of the GPU, which in the case of frontend virtualization is accessible only through the GPGPU API in the host. Therefore, transfers between system memory and GPU memory cause a call from the guest into the hypervisor which causes additional latency.

The other source for virtualization overhead is the additional latency when the virtual machine launches a GPU program or when it waits for a GPU program to be executed. Again, these operations have to be initiated in the host through the GPGPU API, which adds a significant overhead to many GPGPU API functions in the guest. [31]

2.3 Live Migration

The migration of virtual machines should be possible with as little virtual machine downtime as possible and with low performance impact on running applications. Therefore, most of the migration process has to take place while the virtual machine continues running. We will later analyze how live migration is possible with virtual machines with GPUs. In Section 4.5.2 we will examine whether existing algorithms for CPU-based systems can be applied to the virtualization of GPUs.

There are two important live migration algorithms. The first, pre-copy live migration [21], reduces the downtime of the virtual machine by copying as much memory as possible in advance before the virtual machine is stopped for the migration. After the destination system has been selected, the algorithm first transfers all contents of virtual machine system memory to the destination. It then repeatedly copies pages to the destination which have been dirtied by the guest since the last time they have been transmitted (iterative pre-copy). The number of iterations is derived from the size of the writable working set of the virtual machine. After these iterations, the virtual machine is stopped, the remaining state is transferred to the destination system and the virtual machine is resumed.

Pre-copy migration achieves very low migration downtimes with many workloads [21]. Also, while the tracking of dirty pages is usually implemented using page faults and shadow paging, dirty pages can be tracked by other means when the memory management unit does not support shadow paging [20]. As we will describe later, the MMUs in GPUs often have this limitation.

Also, the performance of pre-copy migration heavily depends on the rate at which the virtual machine dirties its memory. Especially with workloads which have a high memory dirty rate, post-copy live migration often performs better. This technique continues execution at the destination machine before any memory has been transferred. When the virtual machine tries to access a page which has not been transferred yet, the virtual machine is paused while the page is transferred [33]. Unlike pre-copy migration, a page is only transferred once. For increased performance, the technique can also be combined with pre-copy migration by executing a single pre-copy iteration before execution is transferred to the destination system. In any case, the technique depends on an MMU which supports demand paging because, unlike pre-copy migration, post-copy migration requires the hypervisor to handle memory read operations if the memory is not available on the destination system.

Chapter 3

Analysis

There have been a number of previous solutions to the problem of GPU virtualization. However, those which are flexible enough to be used in the described HPC cloud computing environment (like the ones in [38], [30], [31] or [24]) generally suffer from significant virtualization overhead. This thesis takes a different approach to the problem by looking at the underlying hardware and exploiting hardware properties in order to reduce the virtualization overhead to a minimum.

Therefore, this chapter will first list and analyze the main sources of virtualization overhead in Section 3.1. One important source of virtualization overhead is the communication between the guest and the host parts of the GPU virtualization system. As this overhead can be reduced by reducing the number of operations which are delegated to the host, Section 3.2 will analyze which hardware features are commonly used by GPU vendors to implement large parts of the GPGPU stack in userspace. Afterwards, Section 3.3 will contain a description of the hardware limitations especially in the area of memory management which are relevant to the problem.

3.1 Main Sources of Overhead

Among all previous GPU virtualization approaches, the ones using frontend migration seem to be best suited for HPCaaS, because they support migration. However, all existing frontend migration designs cause significant virtualization overhead. We have identified two main sources of overhead, which we will describe in the following.

One of the two main sources of overhead is the reduced bandwidth of transfers between GPU memory and system memory [39], which is caused by the GPU virtualization layer. While GPGPU applications on the host system generally can make use of DMA to efficiently transfer memory without making additional copies in system memory, most existing virtualization implementations introduce at least one CPU copy during memory transfers to or from the GPU. This copy is introduced because the guest is not able to write directly into memory which is visible to the GPU. While it is also possible to have the host provide DMA support to the virtual machine in a way which completely removes the need for a CPU copy [31], DMA transfers are always initiated by the hypervisor. This results in significantly higher latency of these operations as they need to be passed to the hypervisor through a hypercall interface.

The second main source of overhead is that the same additional latency also occurs for other GPGPU operations which are executed by the host on behalf of the guest. Besides DMA operations as described above, examples for these operations include executing GPGPU programs on the GPU, waiting for previous GPU commands to finish and allocating GPU memory. Existing approaches build upon vendor provided CUDA or OpenCL stacks which reside outside the virtual machine. As resource management is done on top of the respective userspace APIs, the virtual machine cannot be granted direct access to the underlying GPGPU API and all operations have to be routed through the hypervisor. Therefore, these operations carry significant overhead. However, as the following sections show, in a system without virtualization, most of these operations can be implemented completely in userspace without affecting security or the separation of processes. Therefore, these operations could also be executed completely in the virtual machine in order to reduce the virtualization overhead.

3.2 GPU Hardware Features

For GPU virtualization the boundary between a guest and its hypervisor can be compared very well to the boundary between userspace GPGPU applications and the kernel. Therefore, this section analyzes how some GPU drivers minimize the number of system calls into the GPU driver. Similar measures can then be taken in order to reduce virtualization overhead. In the following, we first describe the interaction between GPGPU applications and the GPU and then discuss the implications on GPU command submission and memory management.

3.2.1 Interaction with the GPU

The main task of the GPU driver is to mediate access to the GPU. Therefore, a call into the GPU driver might be necessary whenever the CPU needs access to the GPU. Interaction with the GPU can be categorized as follows:

- **Executing programs on the GPU:** Before any GPU program is executed on the GPU, the GPU processors need to be configured to run the program. The application specifies the program and the input data address and starts the processors, which then asynchronously execute the program. The result then is either written into GPU memory or directly into system memory.

- **Initiating DMA transfers:** The preferred method for uploading input data or downloading results from the GPU memory to system memory for large amounts of data is DMA. In PCI and PCI-Express systems there is no central DMA engine. Instead the GPU itself contains a DMA engine which directly accesses system memory. To start a DMA transfer, the GPGPU application initializes the DMA engine, which then asynchronously copies the data.

- **Accessing GPU memory from the CPU:** There are cases where DMA access is either impossible or impractical, especially for very small amounts of data. In these cases, GPU memory is directly accessed from the CPU instead.

- **Synchronization:** GPU programs and DMA transfers run asynchronously on the GPU, and before the result can be used, the GPGPU application has to explicitly wait for the operation to be completed. In this case, the GPU can be configured to either issue an interrupt or write a predefined value at a predefined memory location when all previous commands have been executed.

There are GPU drivers which implement all these operations above completely in userspace, with the exception of interrupt handling. An example for such a driver is the PSCNV NVidia GPU driver [13], which exposes the GPU command submission system to GPGPU applications.

3.2.2 GPU Command Submission

All modern GPUs contain a command submission system, which is used to launch GPU programs, start DMA operations or synchronize CPU and GPU. The command submission system of GPUs is designed to be completely asynchronous to programs running on the CPU, with the goal of maximizing utilization of GPU and CPU cores and improving performance, especially in a multithreaded environment [37]. The concurrency is achieved by the use of a command buffer between the GPU and CPU and a hardware unit which reads commands from the buffer and forwards them to other hardware units. After the application has inserted a command into the command buffer, it updates the end pointer of the command ring buffer and the GPU starts executing the command as soon as it is idle.

One example for such a GPU is the NVidia "Fermi" class of GPUs: These GPUs contain an additional layer of indirection as the ring buffer does not contain commands, but rather contains the addresses and sizes of secondary memory regions which instead contain the actual commands [5]. On these GPUs the command submission registers are partitioned into two register regions: One region contains configuration registers which must not be modified by userspace programs because they can be used to circumvent security restrictions and access memory outside the application's address space. Examples for such privileged registers include virtual memory space control registers as described in the following sections. The other region contains the basic ring buffer control registers like read and write pointers and can be mapped into userspace programs. This region is actually held in GPU memory. All write accesses to this region are intercepted and forwarded to the command submission engine.

Through this separation, the ring buffer control region can be mapped into the address space of GPGPU applications. Therefore GPGPU applications can directly launch GPU operations from userspace.

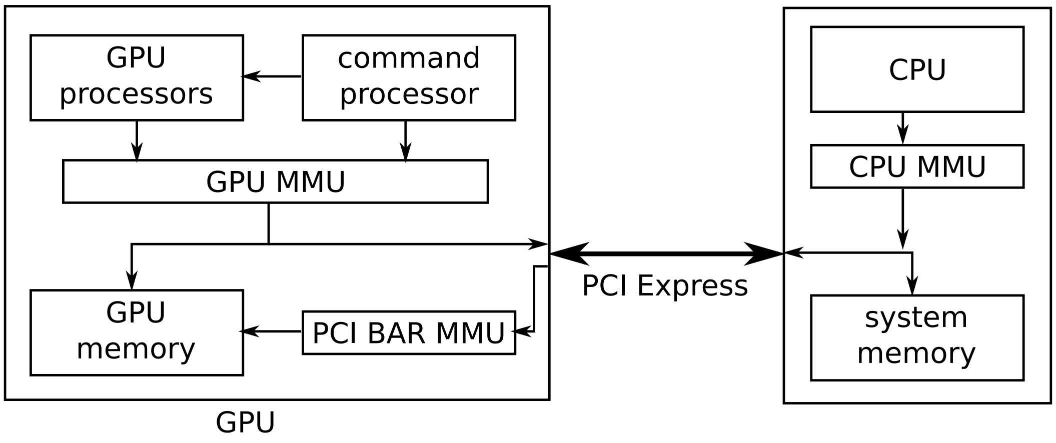

Figure 3.1: Memory access paths in system consisting of a CPU and a GPU

3.2.3 Memory Management

If the GPGPU application is supposed to be able to directly launch programs on the GPU, all different types of memory access have to be possible without intervention of the kernel part of the GPU driver. In the following, we describe the different types of memory access and show how the memory is protected by the hardware for each of them:

- **Access to GPU memory from the CPU:** Data which is regularly accessed by the GPU should be held in GPU memory, because the throughput of GPU memory when accessed from the GPU is significantly higher than access to system memory. There are cases where the CPU needs access to data held in GPU memory, for example if the CPU needs access to ring buffer control structures as described above. Direct access from the CPU is also used to fetch or upload small amounts of data where DMA would be slower than programmed input or output from the CPU. For these kinds of memory transfer, GPGPU applications need direct access to GPU memory.

  For access to GPU memory from the CPU, memory mapped I/O is used, which means that the device memory is mapped directly into the physical address space of the CPU. Each access to the device regions in the physical address space is then forwarded to the device. PCI devices can provide up to six linear memory regions which can be accessed from the CPU [22, p. 316].

  GPUs generally implement one PCI memory region which is used to access GPU memory over the PCI bus. Because the PCI memory region is often smaller than GPU memory, modern GPUs contain a mechanism to map parts of GPU memory into the PCI memory region [11]. All memory accesses from the CPU into GPU memory are routed through a special MMU as shown in Figure 3.1. This MMU can be used to flexibly map parts of GPU memory into different parts of the PCI memory region [7]. The combination of this PCI memory region MMU in the GPU and the MMU in the CPU can be used to enable a GPGPU application to access a part of GPU memory. This part of GPU memory is first mapped into the PCI memory region which is available in the physical address space of the CPU. The CPU MMU can then be used to map the region in the physical address space into the virtual address space of the GPGPU application. Different parts of GPU memory can be mapped into different GPGPU applications. Every GPGPU application only has access to the parts of GPU memory which are owned by the application. Therefore, this method ensures the separation of applications.

- **Access to GPU memory from within the GPU:** During execution, programs on the GPU mostly use GPU memory to store temporary data and results due to the high throughput of the memory interface. GPU memory is also directly accessed from DMA engines or the command submission system. On modern GPUs, most if not all of these memory operations are routed through one or more MMUs which translate virtual addresses in the address space of the GPU program into physical GPU memory addresses [7]. Each GPGPU application is allocated a different GPU page table, therefore GPGPU applications cannot use the GPU to read or write other applications' memory. On NVidia GPUs, the MMU uses two-level hierarchical page tables. Every command submission channel is bound to one such page table and all commands issued from the channel are executed in the virtual address space defined by that page table [7].

- **Access to system memory from the GPU:** Often, programs on the GPU or other hardware units (DMA, command submission system) do not access GPU memory but instead read from or write to system memory. Examples for this include DMA transfers between system memory and GPU memory or the command submission ring buffer, which, for performance reasons, is usually held in system memory. For these memory access operations, the GPU becomes the PCI express bus master and accesses system memory directly [28].

  As any PCI express device has access to all of system memory, access to system memory from GPU programs has to be limited to regions which belong to the GPGPU application. On modern GPUs, however, the same MMU which is used to restrict access from GPU programs to GPU memory can also be used to restrict access to system memory. Through this MMU, GPU memory and system memory are mapped into one unified virtual address space. As an example, on NVidia GPUs, there are two bits in every page directory entry which specify whether the entry points to GPU memory, to cached system memory or to uncached system memory [7].

As described above and as shown in Figure 3.1, both the GPU and the CPU contain MMUs. Every GPGPU application is allocated a GPU virtual address space. This address space is connected to the command submission channel of the GPGPU application which causes all GPU commands to be executed in that address space. The commands are restricted to memory owned by the GPGPU application. Therefore, the GPGPU application can be granted direct access to the command submission system because this access cannot be used to violate the access restrictions of the GPGPU application.

3.3 Hardware Limitations

While modern GPUs contain MMUs, they tend to be less flexible than CPUs regarding memory management. Live migration algorithms designed for CPUs are an example of algorithms which cannot be efficiently implemented with many GPUs. These algorithms usually rely on the processor to be able to resume operation after a pagefault. Pagefaults are used either to implement shadow paging in order to track the modified pages (pre-copy migration) or to implement demand paging on the destination system in order to determine the pages which need to be transferred (post-copy migration).

Modern AMD GPUs support flexible paging and virtual memory [3] and can generally recover after a pagefault has occured. The pagefault handling on NVidia GPUs however is much more limited. While these GPUs support virtual address spaces and raise a pagefault if an invalid virtual address is accessed, the GPU cannot be resumed once it has been stopped because of the pagefault. While the exact capabilities of the GPU are unknown outside of NVidia, it is assumed that the state of the processing units is not completely and consistently saved in the event of a pagefault [14].

Therefore, live migration algorithms which depend on pagefault handling in order to implement shadow paging or demand paging cannot be used with NVidia GPUs.

3.4 Summary

In this chapter we analyzed the hardware features of modern GPUs which can be used to reduce virtualization overhead. The three important types of interaction with the GPU which need to be virtualized efficiently are GPU command submission, GPU memory access and synchronization with the GPU. For systems without virtualization, there are drivers which implement very large parts of the GPU driver in userspace. GPGPU applications can directly submit commands to the GPU and access GPU memory without any call into the kernel.

The central parts of the GPU which make such secure access to the GPU from userspace possible are the memory management units which can be found in modern GPUs. These MMUs can be used to give every GPGPU application a different GPU virtual address space. Because the address space of a GPGPU application is bound to its command submission channel, the address space is used for all its GPU commands. The control structures for the command submission channel can therefore safely be mapped into the GPGPU application to let the application directly submit commands to the GPU. Access from GPU commands then is restricted to memory areas owned by the GPGPU application. Additionally, together with the MMU of the CPU, the GPU MMUs can be used to restrict access to GPU memory from CPU, so the GPGPU application can be given access to selected parts of GPU memory.

Because the GPGPU application has direct access to GPU memory and can directly submit commands to the GPU, most interaction with the GPU can be implemented completely in userspace. The same techniques could be applied to virtualization to allow the virtual machine to interact directly with the GPU.

Chapter 4

Design

This chapter describes the proposed GPU virtualization solution, which causes lower virtualization overhead than other GPU virtualization solutions. The low overhead is achieved by exploiting the hardware properties described in Section 3.2. The solution specializes on the use case of GPU virtualization where the GPU is on the same physical system as the virtual machine. While other GPU virtualization solutions support placing the GPU on a different physical system, the limited bandwidth of the network adds significant overhead, so we do not expect such a setup to be widely used for high performance computing. What has to be supported by the GPU virtualization design though is that multiple virtual machines on one physical system might share one GPU. The proposed solution is targeted at high performance computing on GPGPUs, but is not limited to HPC and could easily be adapted for graphics processing.

First, in Section 4.1 we present an overview over the components of the virtualization solution and the interaction between them. As the approach heavily relies on mapping of GPU memory into the virtual machine, we explain how the proposed virtualization solution constructs these mappings in Section 4.2. Section 4.3 contains a description of how the virtual machine can directly issue commands to the GPU without breaking its isolation. Together with the mapping of GPU memory into the virtual machine, the described method for command submission can reduce the overhead of many GPU commands. As GPUs often only have limited support for multitasking, Section 4.4 shows how the virtualization solution can implement multitasking on the host system and how fairness between multiple virtual machines can be enforced.

On top of this virtualization solution, in Section 4.5, we show how migration of GPGPU applications is possible. We discuss the use of existing live migration algorithms and present a migration strategy which is suitable for HPC GPGPU applications.

4.1 Architecture Overview

As shown in Chapter 3, it is possible to grant userspace programs direct access to the GPU. In the proposed design, we apply the same principles to the virtualization system in order to provide the same level of access to virtual machines. The virtualization software makes

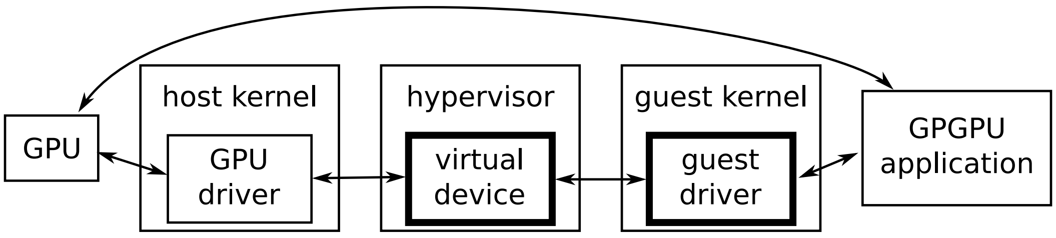

Figure 4.1: Architecture Overview (the two components of the design are marked)

the userspace API from the host available to userspace programs in the virtual machine. Therefore, the proposed design consists of two parts: A virtual device in the hypervisor, and a device driver for the virtual device in the guest kernel. Figure 4.1 shows the interaction between these two components of the proposed GPU virtualization design and other involved components of the system.

The hypervisor component implements a lightweight virtual GPU. It is responsible for resource management (memory management and management of GPU command submission structures) and maps parts of the GPU PCI memory regions into the guest. The guest kernel driver then uses the virtual GPU to pass resource allocation requests from guest userspace applications to the hypervisor, and maps the allocated resources into the application. The API which is presented to GPGPU applications in the guest system is compatible to the GPU API present in the host. Therefore the guest can use the original user space parts of the GPGPU software stack.

4.2 Memory Allocation and Mapping

In the proposed design, the host manages GPU memory. This is necessary for two reasons: First, the host has to enforce separation between host and guest and between multiple virtual machines. Therefore, both GPU and system memory have to be managed by the host. Second, the host has to retain full control over the guest and has to be able to unmap or remap memory for example for migration or to move data between system and GPU memory in the event of GPU memory shortages. For the same reasons, the host also has to initialize the GPU, initialize the command submission system and handle GPU interrupts. If these operations were done by the guest, they would allow the guest to either starve the host by generating lots of interrupts or to violate virtual machine boundaries by reconfiguring memory management.

GPU initialization and command submission management are normally implemented in the host kernel. The proposed virtual device in the hypervisor merely forwards calls from the guest to the host kernel and tracks and manages the allocated resources. Optionally, the hypervisor also performs additional checking in order to implement memory or processing time quotas.

If the GPGPU application in the guest requests GPU memory, the request is passed to the hypervisor which forwards it to the host kernel driver. If the guest requests to map the memory into its address space, the hypervisor lets the host driver map it into the PCI memory regions of the virtual GPU device. The virtual device driver in the virtual machine then maps parts of the PCI memory region into the GPGPU application.

By default, memory is not mapped after an allocation request because the PCI memory regions are of limited size and are usually smaller than the available GPU memory [6]. Therefore, only parts of GPU memory can be visible to the CPU (in the host or in the virtual machine) at any point in time. Limiting the amount of available memory based on the number of virtual machines running in parallel on the same system is impractical because this number can change at any time. Instead, virtual machines assume that a fixed amount of GPU memory can be mapped. If multiple virtual machines share one GPU, there can be situations where the total amount of memory which is supposed to be mapped by virtual machines is higher than the size of the underlying PCI memory regions. In this case, the size restriction can not be circumvented by moving allocated GPU memory regions into system ram, because GPU virtual address spaces cannot be modified while GPU programs are executed. However, while older GPUs often only map a linear region from GPU memory into the PCI memory regions, many modern GPUs contain a special MMU for accesses into the PCI memory regions [11]. The page tables for this MMU can be exchanged even while GPU programs are running. Therefore, at any time, this MMU together with the MMU in the CPU can be used to make different parts of GPU memory visible depending on which parts are needed at that time. Memory regions which are supposed to be mapped, but do not fit into the PCI memory region any more are marked as invalid memory in the CPU MMU. When a pagefault in these pages occurs, the GPU driver can then unmap other pages and use the available address ranges to map GPU memory to the faulting address. This technique makes it possible to multiplex the PCI memory region.

GPU memory itself also needs to be multiplexed if multiple virtual machines use the GPU. With the proposed memory allocation scheme, GPU memory can be shared among multiple virtual machines. The hypervisor needs to prevent one virtual machine from gaining exclusive access to GPU memory by completely allocating it. Exclusive access to GPU memory for one virtual machine would mean that other virtual machines have to use slower system memory instead. This problem can be solved by using fixed GPU memory quotas to limit the amount of GPU memory which can be allocated by a single virtual machine. However, because the maximum number of virtual machines using the GPU varies and is often not known in advance, the GPU memory limit for a single virtual machine has to be less than the total amount of available GPU memory. Such a fixed quota leads to low memory utilization and performance losses if only one virtual machine uses the GPU. We propose using dynamic memory quotas, which scale according to the number of virtual machines using the GPU. If an additional virtual machine starts using the GPU, the quota is lowered. If virtual machines then exceed the quota, their GPU command submission channels are temporarily suspended while some of their data in GPU memory is transferred into system memory. Since the GPU can access system memory directly, the GPU virtual address space can be updated transparently to point to the new location of the data. The GPU command submission channels can then be resumed even if some of the application's data is not in GPU memory. In this case, however, the performance of the application will be significantly limited due to the reduced memory bandwidth. However, in a HPCaaS environment, load balancing should usually be able to avoid such situations.

4.3 GPU Command Submission

In order to use the GPU, GPGPU applications in the virtual machine need to send commands to the GPU. Examples for such commands include commands to start GPU programs, launch DMA operations or initialize parts of the GPU. This command submission system needs to be efficient in order to keep the virtualization overhead low. Also, applications must not be able to break the isolation of the virtual machine by sending commands to the GPU which access memory outside the virtual machine boundaries. Therefore, GPU commands must only be able to access memory which belongs to the GPU application.

As shown in Chapter 3, it is possible to grant applications direct access to the GPU in a way that they can directly submit GPU commands. In the proposed design, the same technique is applied to virtual machines. In addition to GPU memory, the hypervisor also maps the GPU command submission structures into the virtual machine. These structures include a GPU command ring buffer as well as its ring buffer control registers (put and get pointers). One virtual machine can request multiple command submission channels (usually one per GPGPU application), so one set of command submission structures is mapped per such request. Whenever a GPGPU application submits a command, it writes the command into the application's ring buffer and increases the value in the corresponding ring buffer put register.

One example for such a situation is launching GPU programs. In this case, the application writes commands to configure and start the GPU processors into the ring buffer. No additional calls into the hypervisor are necessary [8]. Therefore, command submission normally does not incur any virtualization overhead.

A prerequisite for direct access to the GPU is that the GPU guarantees memory safety in order to ensure isolation of the virtual machine. As described above, modern GPUs contain MMUs to restrict memory access of GPU commands, and every command is executed within the GPU address space of its command submission channel. [7] This fixed connection between command submission channels and an address space can be used to ensures virtual machine isolation.

In the proposed design, the hypervisor allocates address spaces on behalf of the virtual machine. Whenever the virtual machine requests a channel, the hypervisor connects it to one of these address spaces. The mappings of the virtual address space are constructed by the hypervisor purely from memory allocated to the virtual machine. Therefore, no access to memory outside the virtual machine is possible.

Through this method of command submission channel passthrough, it is possible to let the guest perform most interaction with the GPU without intervention of the hypervisor.

4.3.1 GPU Interrupts

There is one exception where the hypervisor has to be involved: When the application has launched a GPU command, it has to wait for its completion, either by polling or by waiting for an interrupt. While it has been shown that polling leads to lower latency and higher performance in certain situations [43], especially the multitasking performance can be significantly increased by making use of GPU interrupts to signal the completion of a command. Because GPGPU applications in virtual machines are supposed to be able to use interrupts for synchronization, the proposed GPU virtualization solution needs to notify the virtual machine whenever one of its GPGPU applications has triggered an interrupt.

Other GPGPU virtualization solutions are built on top of either the OpenCL or CUDA GPGPU API. Therefore, synchronization is implemented by calling either `clFinish` [12] or `cuCtxSynchronize` [4] in the hypervisor. Whenever an application waits for GPU commands to be completed, it requests synchronization, causing the hypervisor to call the appropriate function. After the function has returned, the hypervisor raises an interrupt in the virtual machine to notify it of the completion of previous operations.

Because the proposed solution builds directly on top of the GPU hardware, it has direct access to GPU interrupts. Whereas previous solutions required a hypercall in order to call the synchronization function of the underlying API, having direct access to all interrupts means that applications do not have to notify the hypervisor when they expect an interrupt. Instead, they can directly configure the GPU to raise an interrupt after a command has been executed. For NVidia GPUs, the GPGPU application inserts an additional command into its command submission channel which raises an interrupt when it is executed by the GPU [5]. As this interrupt is handled by the host GPU driver, the GPU virtualization software needs to forward the interrupt to the GPGPU application in the virtual machine.

In order to forward the interrupt to the correct virtual machine, the host driver first determines the command submission channel which caused the interrupt. Then, the driver informs the hypervisor of the interrupt, which in turn raises an interrupt in the virtual GPU device. This interrupt is then handled by the guest kernel driver, which notifies the corresponding GPGPU application.

4.4 Multitasking on the GPU

Contrary to techniques like PCI passthrough, the proposed virtualization software allows multiple virtual machines to share one GPU. In this scenario, multiple command submission channels have been allocated and execute commands in parallel. On NVidia GPUs, up to 128 command submission channels are supported by the hardware, which also implements automatic context switching between these channels [5]. On other GPUs there might be significantly fewer command submission channels. However, it is also possible to save and restore the state of a hardware command submission channel by temporarily unmapping the channel control structures from the GPGPU application and saving or restoring their content. Because suspended command submission channels do not occupy a hardware command submission channel, it is possible to emulate more channels in software.

In most modern GPUs, context switching between multiple channels is not preemptive [26] and commands which are executed have to be completed before a different channel can be activated. There are GPU schedulers which either provide a fair distribution of processing time [25] or provide responsiveness for soft realtime applications [34]. These scheduler algorithms can be combined with any GPU virtualization solution. However, they do not solve the problem that one virtual machine can starve all others by executing a GPU program containing an endless loop.

On all modern GPUs, it is possible to kill commands which take too much time by resetting parts of the GPU while retaining all memory content. In the proposed GPU virtualization solution, the host kernel driver exposes a function to forcibly stop and destroy a command submission channel. The hypervisor then uses this function to implement a policy which restricts the time a single GPU command is allowed to take. This policy is activated only when there are multiple virtual machines on one physical system which share one GPU. If only one virtual machine is using the GPU, long-running GPU programs do not have any negative effect on the system. Such a policy is viable in general because almost all GPGPU applications execute rather short-running GPU programs [35]. Longer-running GPGPU applications usually do not execute one long-running monolithic GPU program but rather execute multiple shorter programs.

If an application does want to execute a long-running GPU program, it is also possible that the guest splits the GPU program into multiple smaller parts [18]. In the proposed design, the host cannot enforce splitting GPU programs into smaller parts, because the guest has direct access to the GPU and can directly execute arbitrary GPU programs. However, if the guest does not cooperate and instead starves other virtual machines by using long-running GPU programs, the host can always kill the GPU program as described above in order to enforce fairness.

4.5 Migration

Important for the viability of the virtualization solution in the cloud computing scenario is the ability to migrate the virtual machine between different physical systems. Other virtualization solutions have the GPGPU API as a well defined abstraction between the virtual machine and the GPU. At this API boundary, all information which is needed for migration is available, and the hypervisor intercepts all communication with the GPU and is therefore able to hide the migration from the virtual machine. With the proposed GPU virtualization solution, this abstraction layer does not exist. Instead, the virtual machine communicates directly with the GPU.

This section shows that, even though the virtual machine has direct access to the GPU, there is still enough information and enough control over the GPU to transparently migrate the virtual machine. The GPU migration consists of two parts, pausing the GPU before the migration and later resuming execution of GPU programs, and storing the GPU state and restoring the GPU to exactly the same state as before the migration. After a description of how these phases can be implemented on GPUs, in this section, we discuss the use of CPU-centric live migration algorithms together with GPUs. We also present an algorithm to migrate the virtualized GPU with very low virtual machine downtime.

4.5.1 Suspend and Resume

During migration, the state of the GPU needs to be transferred between the two involved physical systems. Because GPU commands can change the GPU state in unpredictable ways, it is impossible to save a consistent image of the GPU state while the GPU is running.

Therefore, all GPU commands which belong to the virtual machine need to be paused before the migration can start and later need to be resumed on the destination system.

As GPU commands are not interruptible, the only way to pause the GPU is to wait until the current GPU commands have been completed. As this step has to be completed within a finite amount of time, the hypervisor has to make sure that the virtual machine is not able to submit additional GPU commands while the virtual machine is trying to pause the GPU. This can be ensured for example by unmapping the command submission structures.

However, the GPGPU application still expects to have access to the GPU and will continue to read and modify the command submission structures. Therefore, the host has to create a copy of the command submission structures and place it in system memory. This shadow copy is then mapped into the virtual machine, at the addresses where the command submission structures used to be. With this copy in place, the GPGPU application does not notice that the GPU has been suspended, which is a key requirement for transparent migration.

After having made sure that no new commands can be sent to the GPU, the host needs to wait until all previously submitted commands have been executed. As described in Section 4.3.1, waiting for the GPU is best implemented by configuring the GPU to raise an interrupt after the last command has been completed. The host writes a command to raise an interrupt into the command submission channel of the GPGPU application. After the interrupt has been received by the host, the GPU has been suspended and cannot modify GPU memory or system memory. At this point, a migration of the virtual machine is possible.

After the migration has been completed, the GPU needs to be resumed. However, in the meantime, both the GPU and the application have continued execution during parts of the migration. After the command submission structures have been unmapped from the application, the GPU continues to execute commands which have been previously submitted by the guest. Every time the GPU has completed a command, it updates the original command submission structures to reflect the state of the GPU. At the same time, the guest has been given a shadow copy of the command submission structures. If the guest now submits another command to the GPU, it inserts the command into the shadow copy. Because in this situation there are two versions (original structure and shadow copy) of the command submission structure, which have been independently modified, both have to be combined in order to reach a consistent GPU state. The combined state needs to contain correct information about commands which have already been executed by the GPU as well as commands which have been submitted by the application while the GPU has been paused. The relevant information is stored in the ringbuffer control registers, where one register specifies the first GPU command which has not been executed yet, and another specifies the last GPU command which has been submitted by the application. The two states are combined by taking the content of first register from the original GPU command submission structure, and taking the content of the second register from the shadow copy. After the migration has been completed, the combined state can be used to initialize a new command submission channel on the destination system.

4.5.2 Pre- and Post-Copy Migration

During the migration, all content of GPU memory needs to be transfered to the destination system. During this transfer, the GPU is halted, which causes large interruptions for GPGPU applications. As described in Section 2.3, the same problem also exists for virtual machines without GPUs when system memory is transferred while the virtual machine is halted. However, there are two algorithms, pre-copy and post-copy migration, which allow system memory to be transferred while the virtual machine continues to run. In this section, we show that it is impossible to implement efficient pre-copy or post-copy migration for GPUs.

Pre-copy migration is a technique where the virtual machine continues to run on the source system while the hypervisor starts to transfer memory to the destination system. If the virtual machine modifies parts of memory afterwards, these parts are transferred again [21]. In theory, this technique is easily implemented for GPUs, because the CPU has direct access to GPU memory even while the GPU is busy executing GPU programs. In practice, however, it is impossible to achieve acceptable performance for the transfer of memory from the GPU to the CPU. Because the GPU does not support preemption, no DMA transfer can be initiated until the GPU has finished executing the GPU program. Therefore, the host has to fall back to programmed I/O to fetch the data from GPU memory. Because programmed I/O is often magnitudes slower than DMA, it is impossible to implement efficient pre-copy migration while the GPU is running.

Post-copy migration is another technique for live migration, where the virtual machine is started on the destination system before the virtual machine's memory has been transferred. Afterwards, whenever the virtual machine tries to access data which is not available on the destination system yet, the data is transferred and the virtual machine is resumed when the data is available. A key requirement of post-copy migration is that the hardware needs to be able to raise pagefaults in order to signal when it needs data which is not available yet. Also, the hardware must be able to continue execution once the data is available. However, as shown in Section 3.3, many GPUs do not support pagefault handling to this extent. On these GPUs, resuming execution of the previously running GPU program after a pagefault is assumed to be impossible [14]. Because pagefaults cannot be used to track accesses to unavailable memory, it is impossible to implement post-copy migration on these GPUs.

4.5.3 Migration Strategy

Although live migration of the GPU is not possible, GPGPU applications usually do not have any realtime requirements which make it necessary. Therefore it is often not important that the GPU is responsive throughout the migration. However, the time during which the GPU does not execute any command negatively affects overall performance of the system. Especially if the virtual machine participates in a larger cluster, any slowdown of one of the machines could cause the other machines to stall as they wait for results from the virtual machine which is migrated. Because of such cases, the time during which the GPU is not processing commands has to be minimized. This means that the hypervisor should already transmit the content of system memory while the GPU is still running.

Additionally, in a cloud computing environment it is highly desirable that the virtual machine stays accessible. Therefore, while the GPU might be paused during the migration, the rest of the system has to continue. This section proposes a migration strategy which combines a live migration algorithm for the CPU and system memory with asynchronous migration of the GPU state in order to achieve this goal.

The migration strategy consists of several steps:

1. **Restricting access to the GPU:** The command submission channel structures are unmapped from the virtual machine and replaced by a static copy in system memory. From this point on, the virtual machine cannot launch new GPU commands. As explained above, this restriction is necessary so that the hypervisor is able to pause the GPU. The copy is mapped into the virtual machine again because the migration is supposed to be transparent and the guest parts of the GPGPU stack assume the GPU to be accessible. If the virtual machine changes the command ring buffer pointers, these changes can be replayed later when the GPU is resumed.

2. **Transferring system memory:** The hypervisor transfers the content of system memory and the parts of GPU memory which are visible to the virtual machine. This includes the copies of the command submission structures which have been created in the first step and which are still mapped into the virtual machine. During this step, the hypervisor employs pre-copy live migration to minimize the migration downtime. Because the time during which the GPU is paused has to be minimized, this step is executed in parallel with steps 3 and 4.

3. **Pausing the GPU:** The hypervisor waits until the GPU has executed its current commands. Because GPU commands are not preemptible, commands cannot be canceled without destroying the state of the GPU. After the last command has been executed, the GPU is idle and the command submission structures can be destroyed.

4. **Transferring GPU memory:** The hypervisor transfers the parts of GPU memory which are not mapped into the virtual machine. The data in these memory regions cannot be modified at this time because the GPU is halted and the virtual machine does not have access to the data. Therefore, it only needs to be transferred once, unlike the memory regions which are accessible to the CPU and which might be modified during the migration.

5. **Transferring remaining virtual machine and GPU state:** The hypervisor halts the virtual machine and transfers remaining information about GPU memory allocation and mappings. The amount of data generally is very low and consists of few bytes per GPU memory allocation. Additionally, this data can be modified at any point during execution of the virtual machine as the virtual machine still has to be able to allocate or free GPU memory. Therefore, this data must be transferred while the virtual machine is halted.

6. **Restoring GPU state:** On the destination system, the GPU is initialized and the command submission channels are restored. This step includes the initialization of the GPU virtual address spaces and the configuration of the DMA engine and the GPU processors. If the virtual machine has written any commands while the GPU has been halted, the GPU immediately starts to execute these commands as soon as the ring buffer pointers are updated.

7. **Resuming the virtual machine:** At this point, initialization of the virtual machine is complete and the state of the GPU is the same as before the migration.

The migration scheme is transparent to the virtual machine because, although the virtual GPU is unresponsive for some time during the migration, no changes in the GPU state are visible to the virtual machine. The differences in GPU performance cannot be distinguished from performance changes due to the execution of GPU programs from other GPGPU applications on the same physical system. As GPGPU applications rarely have realtime requirements, this pause is not problematic for most of them. The migration scheme is also live because, although the GPU is halted for some time, the virtual machine itself is accessible during the migration except for a rather short amount of time while the pre-copy migration executes its last iteration and the last parts of the GPU state are transferred.

Chapter 5

Implementation

In Chapter 4, we described a solution for the virtualization of GPUs. In order to evaluate the performance, flexibility and general viability of the design, we implemented a prototype based on Linux and KVM. In the following sections we will give an overview over the prototype implementation and describe the limitations of the prototype. We will show how the prototype was integrated into the KVM hypervisor and Linux, and we will describe how the migration strategy was implemented.

5.1 Overview

The GPU virtualization prototype was built on top of the Linux kernel and the KVM hypervisor. For the sake of simplicity, the prototype is limited to one specific GPU model (NVidia GeForce GTX 480). This GPU was chosen because it has very good open source driver support and because it is modern enough to contain all relevant hardware features. The implementation for other recent NVidia GPUs is very similar as the command submission system and the commands do not differ much from the chosen GPU model.

There are several external components which are used by the prototype. The kernel driver used in the host system is pscnv, an open source driver for Fermi class NVidia GPUs [13]. This driver was chosen over the more stable and more widespread nouveau driver, because the latter does not provide GPGPU applications with direct access to the command submission system but instead implements command submission in the kernel. For the guest user space parts of the GPGPU stack, Gdev (an open source CUDA implementation for NVidia GPUs) was used [9].

Because the prototype is supposed to be used mainly for performance evaluation, it does not support mapping more memory than available through the PCI memory region of the physical device. This feature is not needed for any of the benchmarks described in Chapter 6. The reason for the limitation is mainly that pscnv does not support unmapping GPU memory once it has been mapped into GPGPU applications.

5.2 Integration into KVM and Linux

As we have described in Section 4.1, the design consists of two parts, one in the hypervisor and one in the guest kernel. In the prototype, the virtual device in the hypervisor has been implemented as a virtual PCI device. The device exposes two memory regions. One memory region is used to map memory into the virtual machine which has been allocated through the host GPU driver. Memory is mapped into this region using `mmap`. In the prototype, this region is not only used for GPU memory but also has to be used for all system memory which is visible to the GPU. This limitation results from the choice of the underlying GPU driver in the host. The pscnv driver expects all memory which is visible to the GPU to be allocated through its set of memory management functions. Therefore, it is not possible to make memory visible to the GPU which already has been allocated to the virtual machine before.

The other PCI memory region is used to implement a custom simple hypercall interface. The memory region is not backed by any physical memory. Instead, the KVM function `memory_region_init_io` is used to create the memory region. This function causes every access to the memory region to trigger a function call into the virtual device code. The virtual machine uses this hypercall interface to implement the remaining functionality which cannot completely be implemented in the guest. Examples for such functionality are allocation, deallocation and mapping of GPU memory as well as the creation and management of GPU virtual address spaces and command submission channels.

In the guest, a Linux PCI device driver has been implemented which uses the hypercall interface and simply relays this functionality to userspace applications. The userspace API of the driver is compatible to the API of the pscnv GPU driver, therefore the userspace applications do not have to be modified for use in the virtual machine.

5.3 Migration

The prototype implements the migration strategy which has been presented above. This migration strategy consists mainly of suspending the GPU, transferring GPU memory, transferring system memory and resuming the GPU. System memory is transferred by KVM through pre-copy migration. Because the KVM pre-copy migration implementation transfers all memory regions which are backed by memory (instead of virtual memory regions which trigger callbacks in KVM), the GPU memory which is mapped into the virtual machine is also transferred automatically. Memory which is not mapped into the virtual machine however is transferred differently. This memory cannot change once the GPU has been paused, so it only has to be transferred once by the implementation of the virtual device.

Between system memory and GPU memory, data can be transferred either via DMA or by mapping the GPU memory into the address space of the hypervisor by calling `mmap` and then letting the CPU copy the data into the mmapped region. For copying from system memory into GPU memory, we found CPU based copying to be fast enough. Therefore, the prototype implementation does not use DMA on the destination system. Downloading data from GPU memory, however, is significantly slower when done by the CPU because multiple memory accesses cannot be combined, so every memory access triggers a PCI transfer. Therefore, the hypervisor creates a GPU command submission channel before the migration is started and uses it to transfer data from GPU memory to system memory before it is sent to the destination system.

Once the complete state has been transferred, the GPU needs to be resumed on the destination system. Because the prototype uses an unmodified GPU driver in the host which does not support restoring the GPU state from memory, the hypervisor contains code to reinitialize the GPU. Resuming the GPU is done through four steps:

1. **Allocation of a new command submission channel:** On the destination system, a new command submission channel is allocated. The command submission channel is initialized to place its ring buffer at the same virtual address in the GPU virtual address space as before the migration. Also, the mapping of the command submission channel into the physical address space of the virtual machine is restored.

2. **Initialization of the GPU:** The virtual machine expects the GPU processors and the DMA engine to be ready to execute further commands. Therefore, the hypervisor submits the commands to initialize these parts of the GPU into the command submission channel channel. Because not all information about GPU state is available from the GPU driver, this code is specific to GPGPU applications as it only initializes the GPGPU parts of the GPU as well as the DMA engine.

3. **Initialization of the ring buffer get pointer:**

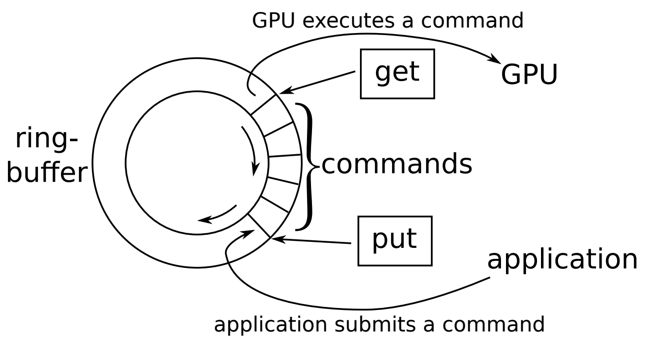

Figure 5.1: GPU command flow and ringbuffer control registers (get and put)

A central part of the command submission structures is the ring buffer where the application places lists of commands for the GPU. The GPU manages two pointers into the ring buffer as shown in Figure 5.1: The put pointer points to the address after the last command which has been written to the ring buffer by the application, while the get pointer points to the first command which has not been executed by the GPU yet.

After a migration, the command submission get pointer has to be placed after the last command which has been executed before the migration, because this is the place where the GPGPU application has written the next command which is supposed to be executed. Because the get pointer initially is at the beginning of the ring buffer, the hypervisor writes multiple NOP commands into the channel until the put pointer has reached the desired position, then waits until the GPU has executed the commands and the put pointer has the same value as the get pointer.

4. **Restoring the ring buffer content:** During the migration, the virtual machine has continued to run and might have placed additional commands in the shadow copy of the command submission channel. In this step, the corresponding ring buffer entries are restored.

5. **Initialization of the ring buffer put pointer:** The virtual machine expects all commands to be executed which have previously been written into the command submission channel. Therefore, the ring buffer put pointer is initialized with the value which is expected by the virtual machine. As soon as the pointer is set, the GPU starts to execute the commands which have been placed in the ring buffer in the previous step.

Chapter 6

Evaluation

In Chapter 4, a method for virtualization and migration with GPGPUs has been presented. This section will show that the method fulfills the requirements for a GPU virtualization solution.

In the next sections, we will first present the methodology of the evaluation. We will describe the evaluation criteria and describe our setup which has been used for the measurements which have been conducted for the evaluation. Section 6.3 then will contain an analysis of the proposed design to show that it meets the criteria which can be shown to be fulfilled without running any benchmarks. In Section 6.4, we describe the benchmarks which have been used to evaluate the other criteria. In Section 6.5, we describe and analyze the results of our evaluation of migration overhead. Section 6.6 then contains a description of the experiments conducted in order to measure migration overhead and virtual machine downtime as well as an evaluation of the results. Finally, in Section 6.7, we discuss remaining limitations of the proposed design.

6.1 Methodology

To evaluate the proposed GPU virtualization solution, we follow the criteria proposed by Dowty and Sugerman [23] as described in Section 2.2.2. These criteria are performance, fidelity, multiplexing and interposition.

Fidelity and multiplexing are attributes which cannot be measured, but instead are inherent to the GPU virtualization design. While fidelity depends on how much the virtualized GPU differs from the underlying hardware, multiplexing depend on whether multiple virtual machines can share one GPU. Therefore, Section 6.3 contains an analysis of the design in order to show that the proposed design fulfills these requirements.

Another criterion is the performance of the proposed design, which can only be judged by measuring the performance of a prototype implementation. The most important performance metric is the virtualization overhead. The virtualization is supposed to provide near-native performance. In order to show that the proposed design meets this requirement, we measure the runtime of several CUDA applications both with and without virtualization and analyze the runtime difference. Each time, the benchmarks are executed 11 times.

Because our goal is to benchmark only the GPU virtualization, the result from the first run is dropped in order to minimize the influence of virtualization overhead during filesystem accesses.

The last criterion, interposition, depends on how well features like suspend and restore as well as migration can be implemented within the proposed design. Additionally, the proposed migration strategy is supposed to be live, which means that the virtual machine downtime has to be kept as low as possible and the effect on running GPGPU applications has to be minimized. Migration overhead and virtual machine downtime are measured while migrating a virtual machine which runs a GPGPU application. During the migration, the duration for which the virtual machine is stopped is measured both in the hypervisor and in the virtual machine. Additionally, the runtime of the GPGPU application is measured and compared to the runtime without migration in order to calculate the migration overhead. All migration benchmarks are also repeated 10 times, the resulting numbers are averaged.

6.2 Evaluation Setup

The benchmarks were executed on a test system with two Intel Xeon E5-2620 CPUs, 32GB of RAM and a NVidia GeForce GTX 480. Both the host system and the virtual machine were running Ubuntu 12.04 with Linux 3.5.7. This kernel version was chosen because of compatibility with the pscnv driver.

Due to limited hardware availability, the migration tests were also conducted on only one physical machine. The migration was performed through a local TCP connection, trickle [27] was used to simulate a network connection with a speed of 100MB/s and an average latency of 1ms. For our cause, these numbers provide a sufficiently accurate approximation of gigabit ethernet. Gigabit ethernet was chosen because it is the slowest network type which is commonly found in high performance computing systems. Therefore, the benchmark results show a lower bound for the performance with different network speeds.

Before any benchmark was run, all but one CPU core were deactivated in the host. The virtual machine also uses only one core. The reason for this limitation is that, when all CPU cores were enabled, the benchmarks results showed significant variations which were not caused by GPU virtualization. In one case, these variations caused a benchmark to need significantly less time in the virtual machine compared to the host. Because we believe that these differences were caused by scheduling effects, we decided to restrict the number of processors.

6.3 Fidelity and Multiplexing

A high degree of fidelity means that all features of the underlying hardware are available to the virtual machine. [23] Because the virtual machine has direct access to the GPU command submission system, it can issue all commands which are supported by the GPU.

Also, the memory management interface provided to guest GPGPU applications is identical to the interface available to applications in the host because the hypervisor simply passes memory allocation requests to the host kernel and returns the results. Therefore, all GPU features which are available to applications in the host system are also available to applications in the guest system and the design provides a high degree of fidelity.

Multiplexing is also completely supported because the GPU driver in the host system supports multiple GPGPU applications at the same time. Therefore, multiple virtual machines can share one GPU because, to the host, each virtual machine behaves like one GPGPU application.

6.4 Benchmarks

In order to measure virtualization overhead, four benchmarks were selected and executed in the host as well as the guest system. The benchmarks were chosen in order to represent a wide range of GPGPU applications as well as to test specific parts of the prototype implementation:

- **mmul:** The mmul benchmark implements a simple matrix multiplication. It initializes the GPU, uploads a random matrix with 2048×2048 entries and squares it. Only one GPU program is executed, this program is rather long-running. Therefore, the benchmark spends little time on initialization and memory management compared to the amount of time spent in the GPU program. While this benchmark is generally not problematic for GPGPU virtualization, it shows that the performance of GPU programs does not differ from GPU programs launched by applications in the host.

- **lud:** The lud benchmark implements a LU decomposition which decomposes a matrix A into two triangular matrices L and U so that $A = LU$. The algorithm is a common method to solve systems of linear equations. The GPGPU application splits the problem into multiple shorter GPU program invocations, therefore this benchmark places more emphasis on the overhead during GPU command submission.

- **backprop/nn:** The backprop benchmark implements a machine learning algorithm which trains the weights of connecting nodes in a layered neural network. [15] The benchmark launches two GPU programs and, in our configuration, has a rather low runtime.

  The nn benchmark computes the nearest neighbours from a set of points. The benchmark launches only a single GPU program and, like the backprop benchmark, also has a very low total runtime.

  Because of the low runtime of the GPU programs, a large fraction of the runtime is spent on initialization. Therefore, the benchmarks stress overheads during initialization and memory allocation. Because we initially expected them to perform worst within the virtual machine because virtualization overheads are amplified, two different benchmarks with similar properties were tested.

Benchmark	Host	Guest	Difference	Overhead
mmul	2922ms	2920ms	-1.72ms	-0.06%
lud	868ms	864ms	-4.88ms	-0.56%
nn	26ms	26ms	-0.08ms	-0.33%
backprop	51ms	53ms	1.82ms	3.55%

Table 6.1: Virtualization overhead benchmark results

All benchmarks use the CUDA driver API which is provided by the gdev CUDA runtime. [9] The first benchmark was taken from the test programs for gdev [10], all others are taken from the rodinia benchmark suite [15].

6.5 Virtualization Overhead

In this section, we want to show that the virtualization solution provides near-native performance, by measuring the virtualization overhead. The overhead is measured by executing the benchmarks in the host system and in the guest system. Both runtimes are measured and compared. The difference between them is the overhead which is caused by virtualization. In the following, we first give some details on how the runtime of the benchmarks was measured, because these details are essential for the interpretation of the results. Then, we describe and interpret the benchmark results.

Because our goal was to benchmark only the GPU virtualization solution, we had to exclude effects from other parts of the virtualization system. The benchmarks spend a significant part of their time in code which is not using the GPGPU API but rather reads input data from the file system or displays results. This behaviour causes large runtime differences which are not caused by the GPU virtualization, but are rather caused by other parts of the virtualization system. Therefore, our measurements only include the parts of the benchmark which interact with the GPU or the GPGPU API. We call `gettimeofday` before the first call and after the last call into the GPGPU API. The runtime of the GPGPU parts of the benchmark then is the difference between these two times. The benchmarks were executed 11 times in the virtual machine as well as in the host system. The result for the first execution was removed because it showed significant overhead caused by file system operations unrelated to the GPU. The remaining values were averaged and compared.

Table 6.1 shows the results of the virtualization overhead benchmarks. The highest overhead was measured when running the backprop benchmark which needed 3.55% more time in the virtual machine. As the maximum overhead is well below 5%, this benchmark result shows that the GPU virtualization does not result in any significant slowdown. Interestingly, however, some benchmarks become faster in a virtual machine. This speedup is rather surprising given that the GPGPU software stack is identical except for the layers added by the virtualization software, and the reason for the speedup is unknown. In the next paragraphs we will present a more detailed analysis of the lud benchmark, which exhibited the largest speedup. Based on this analysis, we present a possible explanation for

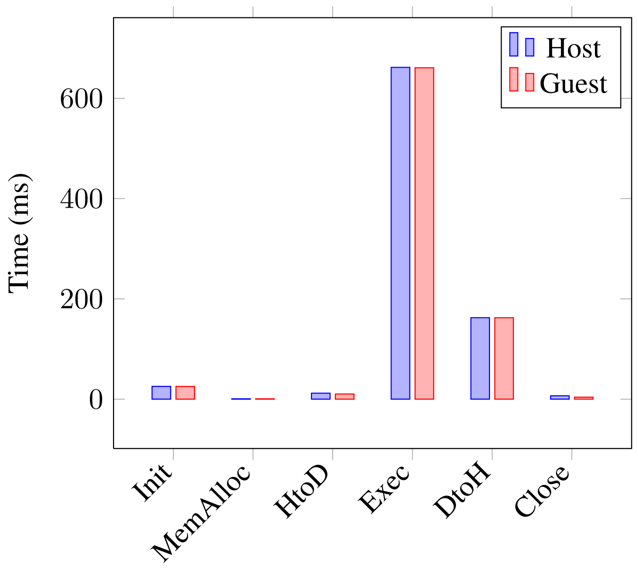

Figure 6.1: Lud Benchmark Results

the observed speedups. Afterwards, we will present a detailed analysis of the backprop benchmark which showed the largest slowdown. We will show how the result matches the expectation that most of the overhead was caused during initialization and memory allocation.

Inside a virtual machine, the lud benchmark showed a speedup of 5 milliseconds compared to execution in the host. Figure 6.1 shows the time spent in different parts of the benchmark. The parts are initialization (Init), GPU memory allocation (MemAlloc), transfer of source data from system memory to GPU memory (host to device transfer, HtoD), execution of the GPU program (Exec), transfer of the results back into system memory (device to host transfer, DtoH) as well as uninitializing the CUDA API (Close). Most of the time seems to be spent during execution of the GPU program and downloading the results into system memory.

The numbers in the figure (and also in Figure 6.1) are slightly misguiding though, due to the asynchronous nature of the GPU. After an application has sent a command to the GPU, the GPU starts executing the command asynchronously. The application can then put more commands into the GPU command queue, or execute code unrelated to the GPU, while the GPU executes previously submitted commands. Therefore, many CUDA functions are non-blocking and often return to the caller well before the GPU commands have completed. For our analysis, we simply measure the time between consecutive calls to CUDA functions. Therefore, our results do not necessarily match the time needed to execute the corresponding GPU commands. For example, in Figure 6.1, the benchmark already initialized the device-to-host memory transfer before the GPU program was completely executed. In this case, most of the time of the device-to-host transfers actually

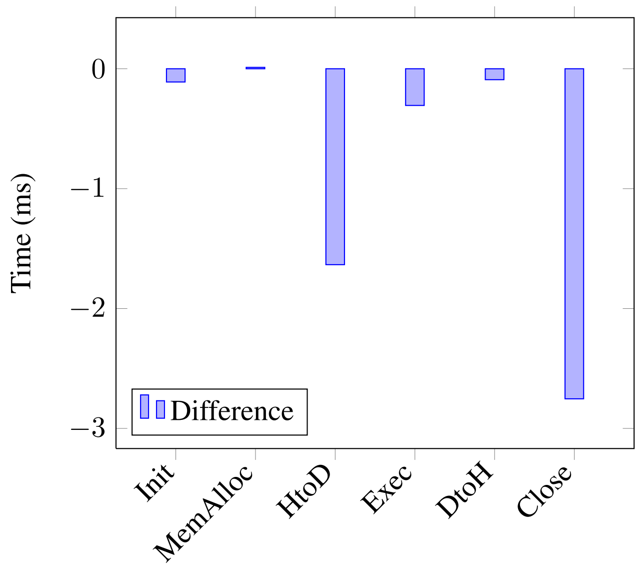

Figure 6.2: Lud Benchmark Results (absolute difference, positive values correspond to a slowdown)

has to be attributed to GPU program execution. Better benchmark results could have been achieved by calling `cuCtxSynchronize` before measuring the time. However, we believe that such forced synchronization can distort the results by changing the behavior of the benchmarks.

Figure 6.1 shows that the relative speedup of the lud benchmark is very small compared to the overall benchmark runtime. As it is hardly possible to see the differences in the figure, Figure 6.2 shows the absolute speedup of the different parts of the benchmark. All parts of the benchmark except memory allocation experienced a speedup in the virtual machine. The exact cause for these speedups is unknown, however the cause might lie outside the GPU parts of the benchmark. The lud benchmark spends more than a second loading input data from a file. These parts are slowed down by about 30 milliseconds in the virtual machine. This slowdown could cause the GPU to be in a slightly different state at the initialization of the GPU API, which could cause varying latencies during GPU command submission. However, due to lack of official documentation and third-party information about the internals of the GPU, we could not locate the exact point where the program needed less time in the virtual machine.

The largest slowdown was found in the backprop benchmark. This benchmark showed a relative slowdown of 3.55%. Figure 6.3 shows the detailed results for the backprop benchmark. Unlike the other benchmarks, the backprop benchmark is split into two phases where a GPU program is executed. Between them, the intermediate results are further processed by the CPU, therefore the benchmark has two additional memory transfer phases. As expected, the initialization, memory allocation and uninitialization phases of the pro-

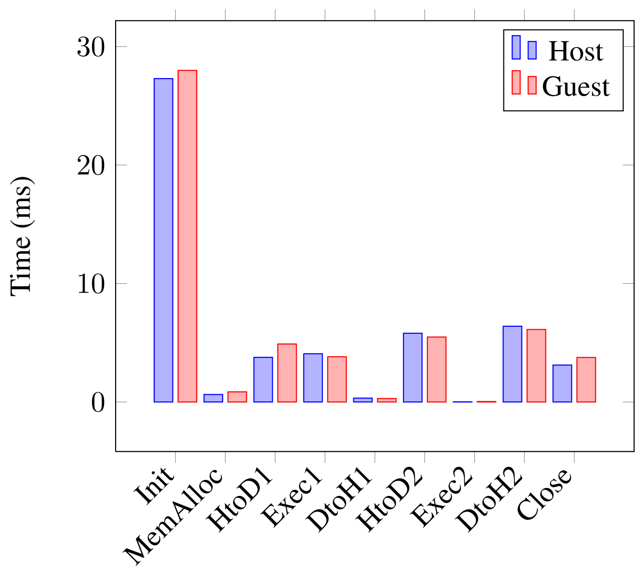

Figure 6.3: Backprop Benchmark Results

gram are slowed down. During these phases, the virtual machine issues multiple calls into the hypervisor. During the other phases, the GPGPU application has direct access to the GPU. Therefore, during these phases, the GPGPU application should not show any performance differences in the virtual machine.

In practice, the benchmark showed a slowdown during the first memory transfer phase. However, given that the slowdown is rather low, it might be possible that it is caused by different CPU cache utilization due to the earlier hypercalls. Also, several other phases showed speedups similar to the ones observed in the lud benchmark. As with the lud benchmark, the source of these speedups unknown. A possible explanation could be timing differences affecting the command scheduling on the GPU.

All in all, several benchmarks showed effects which we could not completely explain. However, the overall overhead has been shown to be very low. In Section 2.2.5 we stated that we assume the virtualization solution to have near-native performance if virtualization overhead stays below 5%. In all cases the measured overhead was lower than this number, even after some margin of safety was added to make up for the unexplainable speedups. Therefore, the experiments show that the proposed GPU virtualization solution provides near-native performance.

6.6 Migration Performance

There are two important metrics for the performance of migration, which are the length of the downtime of the virtual machine and the total migration overhead for running GPGPU applications. Here, as shown in Figure 6.4, migration overhead is the difference between

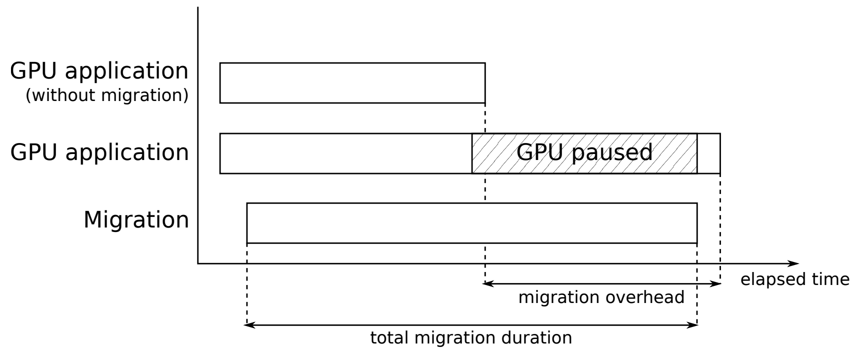

Figure 6.4: Relation between total migration duration and migration overhead

Benchmark	Downtime (Hypervisor)	Downtime (Guest)	Overhead
mmul	111.1ms	149.2ms	962.9ms
lud	121.8ms	196.0ms	1903.9ms

Table 6.2: Migration Overhead and Downtime

the runtime of the benchmark with migration and runtime without migration. Because the migration is supposed to be live, the GPU application continues to run during large parts of the migration. Therefore, the migration overhead is supposed to be smaller than the total migration duration. The migration downtime of the virtual machine is the maximum period during which the virtual machine is paused. Of the benchmarks used to measure virtualization overhead, only the two long-running benchmarks mmul and lud were used to evaluate the migration performance. The benchmarks nn and backprop were found to be impractical because migration had to be started during the few milliseconds while the GPU program was running.

Because the version of kvm used for the prototype did not include code to measure the virtual machine downtime, additional code was added to measure the time during which the virtual machine was paused. However, this value does not include effects from scheduling in the host system which could increase the downtime Therefore, measurements in the hypervisor only provide a lower bound to the actual downtime. To gain an upper bound for the resulting downtime, we additionally measure the discontinuity in the system time of the guest which is caused by the migration. We repeatedly sample the system time by calling `gettimeofday` once per millisecond. The differences between two consecutive time values then are used to compute an upper bound for the downtime. While repeated sampling of the system time is fairly expensive, our experiments have shown that these measurements introduce less than 1% overhead and therefore do not have any significant effect on the benchmark results.

As Table 6.2 shows, the resulting downtime is between 111.1 and 121.8 milliseconds when measured in the host and 149.2 and 196.0 milliseconds when measured in the guest. These numbers are low enough that the downtime will not be noticeable in most use cases,

Benchmark	Overhead	Data Size	GPU Program Runtime	Migration Duration
mmul	962.9ms	48MiB	2801.8ms	3051.3ms
lud	1903.9ms	39MiB	660.9ms	2293.0ms

Table 6.3: Migration overhead compared to allocated GPU memory size, nominal GPU program runtime (without migration) and total migration duration

therefore the migration strategy results in a "live" migration.

The overall migration overhead for the two benchmarks is significantly higher than the virtual machine downtime because it includes the time where the virtual machine was running but the GPU was paused. The mmul benchmark was slowed down by 962.9 milliseconds, while the lud benchmark was slowed down by 1903.9 milliseconds on average. As seen in Table 6.3, this large difference is not caused by a larger amount of GPU memory - the lud benchmark allocates less memory than the mmul benchmark.

However, the runtime of the GPU program has a large effect on the overhead. In our benchmarks, we start the migration while a GPU program is running, so the GPU is paused after the GPU program has finished and before the benchmark fetches the results from GPU memory. As shown in Figure 6.4, the overhead is approximately as large as the period during which the GPU is idle. Because, as described in Section 4.5.3, the prototype transfers system memory and GPU state in parallel, it unmaps the GPU command channel at the same time it starts to transfer system memory. The GPU program of the lud benchmark finishes considerably faster than the first iteration of the live migration algorithm which transfers system memory. While the GPU is then paused, the hypervisor does not only transfer GPU memory but also significant parts of system memory. As the benchmark uses DMA for transfers between GPU memory and system memory, it cannot fetch its results from the GPU until the GPU is resumed after the transfer. Because the GPU is paused longer than necessary, the migration overhead is significantly increased. This benchmark would benefit from a sequential migration strategy which performs the first iteration of the pre-copy live migration algorithm before the GPU is paused.

The mmul benchmark, however, pauses the GPU for about 800 milliseconds, which is reasonably close to the 480 milliseconds which would be needed to transfer the GPU memory over a 100MiB/s network connection. Because in such cases the GPU program is still running while system memory is transferred, the migration strategy provides low migration overhead for such programs which execute long-running GPU programs. However, as shown in Table 6.3, even for other programs, the migration overhead is less than the total migration duration. While our migration strategy has shown to be slightly less efficient for short-running GPU programs, for long-running programs the migration overhead should only depend on the amount of allocated memory and should not depend on the runtime of the GPU program.

6.7 Discussion

As shown above, the proposed GPU virtualization design does not cause significant virtualization overhead and provides near-native speeds for GPGPU applications. We have shown that the migration strategy provides live migration with low virtual machine downtimes and keeps the migration overhead low. However, there are cases where the proposed GPU virtualization solution is not as flexible as other solutions which are designed on top of CUDA or OpenCL.

When a virtual machine is migrated between two different physical systems, the virtualization method requires that the GPUs on both systems are compatible. Because the virtual machine has direct access to the GPU, its GPU driver expects a certain GPU model. Therefore, different GPU models require different drivers in the guest, and the driver needs to be changed when the underlying GPU changes. More precisely, the driver needs to be changed when the hypervisor unmaps the command submission structures before the migration. Starting from this point, all commands will be executed by the GPU on the destination system. Migration between different GPUs is possible, but, because the driver needs to be changed, the hypervisor has to inform the guest about the change and the migration is not transparent. In contrast, other GPU virtualization solutions can hide this complexity from the virtual machine because the GPGPU API serves as a communication standard between the virtual machine and the GPU.

However, in practice it is rather safe to assume that, during construction of a cluster system, not too many different GPUs are used. This includes homogeneous GPU clusters where only one GPU model is deployed, and heterogeneous clusters where different compatible GPUs from only one vendor and GPU generation are mixed. For example, Amazon only offers systems with one specific GPU model [2].

Chapter 7

Conclusion

In this thesis, we have proposed a GPU virtualization design which allows the use of GPGPU applications in virtual machines with very low overhead and high flexibility. Such GPU virtualization systems are required in high performance cloud computing where it is necessary that virtual machines can be migrated between physical systems in order to make maintenance easier and to allow for efficient load balancing. At the same time, virtualization overhead has to be kept as low as possible because the value of a high performance computing system depends on its resulting performance.

Other GPU virtualization solutions have shown to have either high overhead or low flexibility. The virtualization overhead is usually caused either by decreased memory throughput because the virtualization solution does not allow direct DMA access to memory owned by the virtual machine, or it is caused by increased GPU latency as the hypervisor has to process most GPU commands.

Our proposed GPU virtualization design achieves near-native performance by efficiently utilizing the hardware features found in modern GPUs. In our design, the command submission data structures of the GPU are mapped into the virtual machine, which can then access the GPU directly and submit commands. Also, allocated GPU memory is mapped into the virtual machine so that the virtual machine also has direct access to GPU memory. Through this design, most interaction with the GPU can be implemented completely in the virtual machine and, unlike in other approaches to GPU virtualization, does not involve the host. In all cases, memory management units in the GPU and the CPU guarantee isolation of the virtual machine. We have evaluated the performance of the virtualization solution using a set of CUDA benchmarks. Our benchmarks have shown that this GPU virtualization solution does not cause any significant virtualization overhead.

The proposed GPU virtualization is also very flexible, because even though the virtual machine has direct access to the GPU, this access can be transparently revoked for example before the virtual machine is migrated. On top of this, we have proposed a live migration strategy which is used to migrate the virtual machine as well as the GPU. This migration strategy pauses the GPU during the migration and uses pre-copy live migration of system memory in order to keep the virtual machine downtime as low as possible. The prototypical evaluation has shown that the migration strategy achieves very low virtual machine downtime and keeps the overall migration overhead to an acceptable level.

7.1 Future Work

We have shown that efficient virtualization is possible for HPC GPGPU workloads. Another important use of GPUs is, more traditionally, in the area of 3D graphics. A recent development is that 3D graphics applications are moved into the cloud. Further experiments have to show whether the proposed GPU virtualization solution is suitable for 3D graphics applications.

A significant problem with the proposed solution is that long-running GPU programs can prevent a migration because of their nonpreemptive nature. Further research has to be conducted to show whether it is possible to solve this problem even if no hardware support for preemptive multitasking is present. Possible solutions might be a checkpointing approach which allows GPU programs to be cancelled and restarted at the last checkpoint, or instrumenting of all GPU programs in order to implement cooperative multitasking. Also, this problem supposedly does not exist on recent AMD GPUs. [3] Further experiments have to show whether it is possible to implement seamless live migration of GPU programs on these GPUs.

Bibliography

[1] Amazon ec2. http://aws.amazon.com/de/ec2/.

[2] Amazon ec2 instance types. http://aws.amazon.com/de/ec2/instance-types/.

[3] Amd graphics core next (gcn) architecture. http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf.

[4] Cuda documentation: Context management. http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDA__CTX.html.

[5] Envytools nvidia fifo documentation. https://github.com/pathscale/envytools/tree/master/hwdocs/fifo.

[6] Envytools nvidia pci bar documentation. https://github.com/pathscale/envytools/blob/master/hwdocs/bars.txt.

[7] Envytools nvidia virtual memory documentation. https://github.com/pathscale/envytools/blob/master/hwdocs/memory/nv50-vm.txt.

[8] Gdev cuda implementation for nvidia gpus. https://github.com/shinpei0208/gdev/blob/master/common/gdev_nvidia_nvc0.c.

[9] Gdev cuda runtime. https://github.com/shinpei0208/gdev.

[10] Gdev matrix multiplication test. https://github.com/shinpei0208/gdev/tree/master/test/cuda/user/mmul.

[11] Nouveau nvidia hardware introduction. http://nouveau.freedesktop.org/wiki/HwIntroduction.

[12] Opencl documentation: clfinish. http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clFinish.html.

[13] Pscnv. https://github.com/pathscale/pscnv.

[14] Pscnv virtual memory implementation. `https://github.com/pathscale/pscnv/blob/master/pscnv/nv50_vm.c`.

[15] Rodinia benchmark suite. `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page`.

[16] Tesla gpu accelerators for servers. `http://www.nvidia.com/object/tesla-servers.html`.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. `http://doi.acm.org/10.1145/1721654.1721672`.

[18] 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.

[19] Milind Bhandarkar, Laxmikant V Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for mpi programs. In *Computational Science-ICCS 2001*, pages 108–117. Springer, 2001.

[20] Jui-Hao Chiang, Maohua Lu, and Tzi-cker Chiueh. Bootstrapped migration for linux os. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 209–212. ACM, 2011.

[21] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. `http://dl.acm.org/citation.cfm?id=1251203.1251223`.

[22] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition, 2005.

[23] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[24] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231, 28 2010-July 2.

[25] Ashok Dwarakinath. *A fair-share scheduler for the graphics processing unit*. PhD thesis, Stony Brook University, 2008.

[26] Glenn A Elliott and James H Anderson. Real-world constraints of gpus in real-time systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 2, pages 48–54. IEEE, 2011.

[27] Marius A Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[28] Omid Fatemi. Interface design - pci local bus. `http://ece.ut.ac.ir/classpages/F83/Interface/pci.ppt`.

[29] Pawel Gepner, David L Fraser, and Victor Gamayunov. Evaluation of the 3rd generation intel core processor focusing on hpc applications.

[30] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 379–391. Springer Berlin Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-15277-1_37`.

[31] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, New York, NY, USA, 2009. ACM. `http://doi.acm.org/10.1145/1519138.1519141`.

[32] Mark Harris. Gpgpu: General-purpose computation on gpus. *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[33] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[34] 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.

[35] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12. IEEE, 2009.

[36] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. `http://doi.acm.org/10.1145/1365490.1365500`.

[37] Steve Rennich. Cuda c/c++ streams and concurrency. `http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf`.

[38] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on*, 61(6):804–816, June.

[39] M.S. Vinaya, N. Vydyanathan, and M. Gajjar. An evaluation of cuda-enabled virtualization solutions. In *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference on*, pages 621–626, Dec.

[40] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized gpu environment. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2012.

[41] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[42] Chao-Tung Yang, Hsien-Yi Wang, and Yu-Tso Liu. Using pci pass-through for gpu virtualization with cuda. In James J. Park, Albert Zomaya, Sang-Soo Yeo, and Sartaj Sahni, editors, *Network and Parallel Computing*, volume 7513 of *Lecture Notes in Computer Science*, pages 445–452. Springer Berlin Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-35606-3_53`.

[43] Jisoo Yang Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. 2012. `http://static.usenix.org/events/fast12/tech/full_papers/Yang.pdf`.

[44] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. Live migration with pass-through device for linux vm. In *Proceedings of the Linux Symposium*, pages 261–267, 2008. `https://www.kernel.org/doc/ols/2008/ols2008v2-pages-261-267.pdf`.

[45] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010. `http://dx.doi.org/10.1007/s13174-010-0007-6`.

[46] Ye Zhao. Gpu computing - programmable graphics pipeline, 2008. `http://www.cs.kent.edu/~zhao/gpu/lectures/ProgrammableGraphicsPipeline.pdf`.