Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Υπολογιστικων Συστηματων

Σχεδιασμός και Υλοποίηση Μηχανισμού Εικονικοποίησης Μονάδων Επεξεργασίας Γραφικών και Διαμοιρασμού τους σε Εικονικές Μηχανές

Διπλωματική Εργασία

του

Δημήτριου Βάσιλα

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2016

National Technical University of Athens
School of Electrical and Computer Engineering
Computer Science Division
Computing Systems Laboratory

Design and Implementation of a GPU Virtualization Framework

Diploma Thesis

of

Dimitrios Vasilas

Supervisor: Nectarios Koziris
Professor N.T.U.A.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 30 Μαρτίου 2016.

.....	.....	.....
Νεκτάριος Κοζύρης	Δημήτριος Σούντρης	Γεώργιος Γκούμας
Καθηγητής Ε.Μ.Π.	Αναπληρωτής Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Athens, March 2016

.....

Δημήτριος Βάσιλας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Graphics Processing Units (GPUs) have become a powerful platform, that can provide significant performance benefits to data parallel applications. Graphic processors are being increasingly introduced as accelerators in high performance computing (HPC) systems due to the development of GPGPU (General-Purpose Computation on GPUs). Furthermore, virtualization technologies are gaining interest in these domains, due to their benefits on server consolidation as well as the isolation and ease of management they offer. There is thus a growing need to combine the benefits of both fields by providing heterogeneous resources, particularly GPUs, in virtual environments.

In this thesis we address the challenge of integrating GPGPU into virtualized environments. We propose a mechanism that enables the execution of GPU accelerated applications within Virtual Machines (VMs). Our framework consists of two components: a user level library and a paravirtualized driver, which enables communication with the host's GPU driver. To validate our approach, we conduct experiments on a variety of GPU applications, focusing on the virtualization overhead and the scalability of our framework.

Keywords— Graphic Processing Unit (GPU), General-Purpose Computation on GPUs (GPGPU), Virtualization, Cloud, High Performance Computing (HPC), CUDA, Virtio

Περίληψη

Οι μονάδες επεξεργασίας γραφικών (Graphics Processing Units - GPUs) έχουν εξελιχθεί σε ισχυρούς επεξεργαστές, οι οποίοι μπορούν να παρέχουν σημαντικά οφέλη σε εφαρμογές κατάλληλες για παράλληλη επεξεργασία. Οι επεξεργαστές γραφικών χρησιμοποιούνται όλο και περισσότερο σε συστήματα υπολογισμών υψηλών επιδόσεων (High Performance Computing - HPC) εξαιτίας της ανάπτυξης των υπολογισμών γενικού σκοπού σε GPUs (General-Purpose Computation on GPUs - GPGPU). Επιπλέον, οι τεχνολογίες εικονικοποίησης κερδίζουν έδαφος σε αυτούς τους τομείς, λόγω των οφελών τους στην ομαδοποίηση εξυπηρετητών καθώς και την απομόνωση και την ευκολία διαχείρισης που προσφέρουν. Προκύπτει επομένως η ανάγκη να συνδυαστούν τα οφέλη και των δύο πεδίων με την παροχή ετερογενών πόρων, ιδιαίτερα μονάδων επεξεργασίας γραφικών, σε εικονικά περιβάλλοντα.

Η παρούσα εργασία εξετάζει το ζήτημα της ενσωμάτωσης πραγματοποίησης υπολογισμών γενικού σκοπού σε GPUs (GPGPU) σε εικονικά περιβάλλοντα. Παρουσιάζει έναν μηχανισμό ο οποίος επιτρέπει την εκτέλεση εφαρμογών που χρησιμοποιούν επιτάχυνση από GPUs, σε Εικονικές Μηχανές (Virtual Machines - VMs). Ο μηχανισμός αποτελείται από δύο μέρη: μία βιβλιοθήκη επιπέδου χρήστη και έναν οδηγό συσκευής (driver) ο οποίος υλοποιεί παρα-εικονικοποίηση, επιτρέποντας την επικοινωνία με τον driver της GPU του host υπολογιστή. Για την αξιολόγηση της επίδοσης του μηχανισμού διεξάγονται πειράματα σε πληθώρα εφαρμογών GPU, και αξιολογείται η επιβάρυνση στην επίδοση τους λόγω εικονικοποίησης, καθώς και η κλιμακωσιμότητα του συστήματος.

Λέξεις Κλειδιά— Μονάδες Επεξεργασίας Γραφικών, Υπολογισμοί Γενικού Σκοπού σε GPUs (GPGPU), Εικονικοποίηση, Cloud, High Performance Computing (HPC), CUDA, Virtio

Ευχαριστίες

Με την εκπόνηση της παρούσας διπλωματικής εργασίας ολοκληρώνεται ένα σημαντικό κεφάλαιο της ακαδημαϊκής μου πορείας. Θα ήθελα να ευχαριστήσω τους ανθρώπους που με βοήθησαν στη διαδρομή αυτή.

Αρχικά θα ήθελα να ευχαριστήσω τον Καθηγητή Ε.Μ.Π. κ. Νεκτάριο Κοζύρη που μου έδωσε την ευκαιρία να γνωρίσω τον τομέα των υπολογιστικών συστημάτων και τη δυνατότητα να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα σε αυτή την εργασία. Θα ήθελα ακόμη να ευχαριστήσω τον Λέκτορα κ. Γεώργιο Γκούμα και όλα τα μέλη του Εργαστηρίου Υπολογιστικών Συστημάτων για τις εποικοδομητικές συζητήσεις που οδήγησαν στη διαμόρφωση της εργασίας. Ένα ιδιαίτερο ευχαριστώ οφείλω στον Υποψήφιο Διδάκτορα Στέφανο Γεράγγελο για την άρτια συνεργασία μας και τη διαρκή καθοδήγηση του για την εκπόνηση της παρούσας διπλωματικής εργασίας.

Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου για την αγάπη και τη στήριξη τους και τους φίλους μου, οι οποίοι στάθηκαν δίπλα μου σε κάθε βήμα αυτής της προσπάθειας.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays, Graphics Processing Units (GPUs), driven by the insatiable market demand for real-time, high-definition 3D graphics, have evolved into general-purpose, high performance, multicore processors capable of high computation throughput and memory bandwidth. In addition to being efficient at manipulating computer graphics and image processing, their highly parallel structure makes them well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. As a result, GPUs are being introduced as accelerators in order to achieve speed-ups in applications traditionally handled by the central processing unit (CPU). This approach, known as general purpose computation on GPUs (GPGPU), is being increasingly adopted in HPC (High Performance Computing) applications. Research has demonstrated that computationally intensive applications from a wide range of scientific fields, such as finance [1], chemical physics [2], weather broadcast [3], fluid dynamics [4] etc. can leverage GPUs to obtain major gains in performance. In addition to the scientific domain, GPUs are used in software routers [5], encrypted networks [6] and database management systems [7] as well. One of the reasons general purpose computing on GPUs has been well established is the introduction of dedicated programming environments, compilers, and libraries such as CUDA from Nvidia.

On the other hand, virtualization technology has an increasing influence on how computational resources are used and managed. The growth in hardware performance and the increased demand for service consolidation from business markets, leads virtualized cloud environments to host an ever growing amount of computations. Virtual Machines (VMs) can improve resource utilization, as several different customers may share a single computing node with the illusion that

they own the entire machine in an exclusive way, while providing process isolation and ease of management. Consequently, virtualization techniques are a promising effort to run high performance software on a grid, as obtaining virtualized cloud computational resources is an elastic, time and cost efficient alternative to the traditional way of obtaining resources. With the recent advances in both virtualization and GPU technology, there is an ever-growing need to provide heterogeneous resources, particularly GPUs, within the cloud environment in the same scalable and on-demand way as traditional virtualized hardware. Cloud providers are thus facing the challenge of integrating GPGPUs into their platforms. For example, Amazon Elastic Compute Cloud (EC2) [8] provides GPU instances as computing resources, but each client is assigned with an individual physical instance of GPUs. Unfortunately, in the cloud context I/O virtualization suffers from poor performance, due to the overhead incurred by indirect access to physical resources and the need to multiplex the application's access to I/O resources. Virtualization and sharing of GPU hardware face additional challenges due to the characteristics of graphic processing units that do not enable preemptive scheduling and time-sharing capabilities.

1.2 Thesis Contribution

In this thesis we propose an efficient approach to expose GPGPU capabilities in virtual machines. We present the design and implementation of a framework that enables applications executing in virtual environments, to accelerate their performance exploiting GPU resources. By using our framework, multiple VMs co-located in the same host computer can share physical GPU resources. As a proof of concept we target the virtualization of CUDA-enabled GPUs by enabling applications developed using the CUDA platform to execute within VMs. Our approach employs paravirtualization techniques and uses a split driver model. It consists of a user-level library, a frontend driver located at the guest OS and a backend driver implemented at the hypervisor. The framework's architecture is shown in Figure 1.1.

In summary, the main contributions of our work are:

- We propose an efficient GPU virtualization framework that enables GPGPU applications to execute within VMs, and implements GPU resource sharing among co-located VMs.
- We maintain CUDA Runtime binary compatibility, so that existing applications can use our framework without any source code modification.

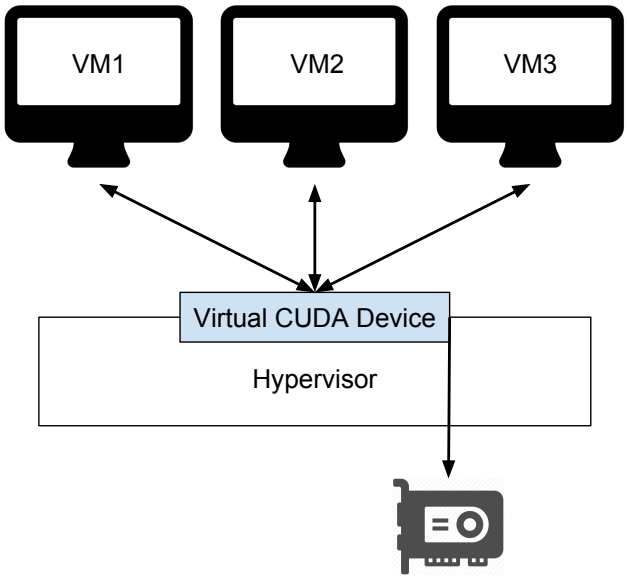

Figure 1.1: The framework's architecture

- We categorize GPU accelerated applications based on their computation and memory access profile and discuss which types applications could benefit by using our framework.

Our performance evaluation shows that our framework achieves low virtualization overhead, making GPU accelerated applications executing within VMs competitive to those executing in native environments with direct access to GPU resources.

Chapter 2

Background

2.1 From Single Core to Heterogeneous Systems

2.1.1 Moore's Law

The evolution of computer systems has been tightly associated with the pursuit of processing power. Today's personal computers have many times more processing power than the first supercomputers, introduced in the 1970s. Moreover, modern supercomputers perform millions of times more floating-point operations per second than their first predecessors. Considering that this milestone has been reached in a little over three decades, it is evident that the evolution of computers in terms of processing power has been exponential. Intel co-founder Gordon Moore first made this famous observation in 1965. He stated that the number of components in integrated circuits was doubling every 12 months and projected this rate of growth would continue for at least another decade. The doubling of the number of components is generally interpreted as the doubling of processing power.

This prediction has been proven accurate for several decades as Moore's law has been used in the semiconductor industry to guide long-term planning and to set targets for research and development. The exponential improvement that the law describes has transformed the first personal computers of the 1970s, with processing speed of a few kHz, to todays supercomputer, many-core smartphones and Internet of Things devices.

2.1.2 Single Core Systems

The first computing systems implemented the von Neumann architecture and processed information serially using a single processing core. For about three

decades, advancements in hardware technology followed Moore's law allowing computing systems to increase their performance. The ever growing number of transistors was used to achieve instruction level parallelism (ILP) through technologies such as branch prediction, out of order execution and superscalar architectures as well as building larger caches and more cache levels. Programmers could benefit from architecture improvements and application's performance increased at no programming cost.

2.1.3 Multicore Systems

Another important effect that contributed to the evolution of single core processors was Dennard Scaling. This scaling law stated that as transistors become smaller their power density remains constant, so that smaller transistors requires less voltage and lower current. As a result, as the size of the transistors shrunk and the voltage was reduced, manufacturers were able to raise clock frequencies without significantly increasing overall circuit power consumption. However, since around 2005 Dennard Scaling appears to have broken down. While transistor count in integrated circuits is still growing, improvements in performance through frequency increases have become challenging due to increased heat and energy costs.

The failure of Dennard scaling led CPU manufacturers to focus on increasing the number of cores on chips as an alternative way to improve performance. Multicore systems enabled performance improvements through parallel processing, solving a problem by dividing it in multiple tasks which can execute simultaneously on multiple processors. The increase in processing power through multicore systems is more challenging to benefit from, since programmers need to develop parallel software that exploits the multicore architecture.

Based on the number of instructions and data streams that can be processed simultaneously, parallel systems can be classified into four categories:

- **Single Instruction Single Data (SISD)**: A SISD computing system is a traditional uniprocessor machine. Such a sequential computer exploits no parallelism in either the instruction or data streams.
- **Single Instruction Multiple Data (SIMD)**: A SIMD computing system is a multiprocessor machine capable of executing the same instruction on multiple cores, operating on different data streams. Example of SIMD systems are array processors and graphics processing units.

- **Multiple Instruction Single Data (MISD)**: A MISD computing system is a multiprocessor machine capable of executing different instructions on multiple cores, all operating on the same data set. This architecture is uncommon and is generally used for fault tolerance.

- **Multiple Instruction Multiple Data (MIMD)**: A MIMD computing system consists of multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multicore processors and distributed systems.

Multicore architectures are categorized into shared memory and distributed memory systems based on how processing cores are coupled to the main memory.

Shared Memory

In the shared memory model all processing cores are connected to a single global memory which they can access. Communication between cores takes place through the shared memory, since modification of the data by one core is visible to all other cores. This model enables fast and uniform data sharing between tasks due to the proximity of memory to CPUs. Global address space also offers ease of programming. However this approach suffers form lack of scalability. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path as well as traffic associated with cache coherence.

Based on memory access times, shared memory architectures can be classified into two categories:

- **Uniform Memory Access (UMA)**: All processors share the physical memory uniformly and access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data.

- **Non-Uniform Memory Access (NUMA)**: Each processor has its own local memory and can also access memory owned by other processors. Memory access time in this model depends on the memory location relative to the processor, since processors can access their local memory faster than non-local memory. The NUMA architecture was designed to surpass the scalability limits of the UMA architectures. It alleviates the bottleneck of multiple processors competing for access to the shared memory bus.

Distributed Memory

In the distributed memory model all processing cores have their own local memory and computational tasks operate only on local data. If remote data are required, the computational task needs to communicate with one or more remote processors. Communication between cores takes place through the interconnection network. The distributed memory model achieves high scalability, since memory can increase proportionally to the the number of processor, as well as cost effectiveness, since commodity processors and networking infrastructure can be used to build such systems. Moreover, each processor can rapidly access its local memory without the overhead incurred by global cache coherence operations. On the other hand, programming is more challenging as developers are responsible for details associated with communication between processors.

2.1.4 Heterogeneous Systems

By the end of 2010, multicore processors had entered the mainstream of affordable computing. Nearly all new desktop computers used dual-core and even quad-core processors. Meanwhile, advances in semiconductor technology led GPUs to grow in sophistication and complexity. Those developments gave rise to heterogeneous computing systems. Heterogeneous architectures use more than one kind of processor (CPUs, GPUs, FPGAs etc.) and gain performance not just by adding cores, but also by incorporating specialized processing capabilities of each kind of processor to handle particular tasks.

GPUs are widely used in heterogeneous systems. They have vector processing capabilities that enable them to perform parallel operations on very large sets of data at lower power consumption, relative to the serial processing of similar data sets on CPUs. While their value was initially derived from their ability to improve 3D graphics performance by offloading graphics from the CPU, they are becoming increasingly attractive for general purpose computations, such as addressing data parallel programming tasks. CPU-GPU systems can cooperate so that mathematically intensive computations are offloaded to GPUs while CPUs can run the operating system and perform traditional serial tasks.

Another kind of processors used in heterogeneous computing systems are coprocessors. An example of such systems is the Intel Many Integrated Core (MIC) architecture which contains the Intel Xeon Phi coprocessor. The Intel Xeon Phi coprocessor is primarily composed of processing cores, caches, memory controllers and a high bandwidth ring interconnect. The coprocessor is connected to an Intel Xeon 'host' processor, through a bus and runs a Linux operating system. Thus,

users can connect to the access as a network node and directly run individual jobs as well as execute heterogeneous applications where a part of the application executes on the host while a part executes on the coprocessor.

Modern smartphone and tablet market has created the need for high performance combined with low power consumption. ARM big.LITTLE heterogeneous architecture has been designed to address theses requirements. This architecture uses two types of processors. 'LITTLE' processors are designed for maximum power efficiency while 'big' processors are designed to provide maximum compute performance. Using this technology, each task can be dynamically allocated to a 'big' or 'LITTLE' core depending on the instantaneous performance requirement of that task. Typically, only one side is active at once, but since all cores have access to the same memory areas, workload can be swapped from 'big' to 'LITTLE' and back on the fly.

2.2 General Purpose Computing on GPUs

2.2.1 Graphics Processing Units

A Graphics Processing Unit (GPU) is a single-chip processor that performs rapid mathematical calculations, primarily for the purpose of rendering images. In the early days of computing, the central processing unit (CPU) used to perform these calculations. As more graphics-intensive applications were developed, their demands degraded performance of CPU. GPU was introduced as a way to offload those tasks from the CPU, freeing up its processing power. Nvidia introduced the first GPU, GeForce 256, in 1999. This GPU model could process 10 million polygons per second and had more than 22 million transistors. ATI Technologies released the Radeon 9700 in 2002 using the term visual processing unit (VPU). Over the past few years there has been a significant increase in performance and capabilities of GPUs due to market demand for more sophisticated graphics.

Nowadays, GPUs are widely used in embedded systems, mobile phones, personal computers, and game consoles. Modern GPUs are very efficient at manipulating graphics as well as in image processing. Furthermore, their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. As a result, a large discrepancy in floating-point capability between the CPU and the GPU was emerged. The main reason is that GPUs are specialized for compute-intensive, highly parallel computation and therefore designed such as more transistors are devoted to data processing rather than data caching and flow control, as is the case for the

CPU. More specifically, GPU is especially well-suited to address problems that can be expressed as data-parallel computations with high arithmetic intensity. Because the same program is executed in each data element, there is a lower requirement for sophisticated flow control. Memory access latency can be hidden with calculations instead of big data caches. GPUs can therefore be considered as general-purpose, high-performance, many-core processors capable of very high computation and memory throughput.

2.2.2 GPGPU Programming Interfaces

General purpose computing on GPUs is the term referring to the use of graphics processing units, which typically handle computation only for computer graphics, to perform computation in applications traditionally handled by central processing units (CPUs). GPGPU is used in a variety of applications such as physics calculations, encryption/decryption, scientific computations and the generation of crypto currencies such as Bitcoin. CUDA (Compute Unified Device Architecture) [9] and OpenCL (Open Computing Language) [10] are two widely used interfaces offering general-purpose computing capabilities on GPUs. Both present similar features but through different programming interfaces.

OpenCL, developed by the Khronos Group, is an open standard for cross-platform, parallel programming on heterogeneous platforms consisting of central processing units (CPUs), GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies a C-like language that enables development of applications that execute on these platforms, and defines an API that allows programs running on the host to launch functions (kernels) on the compute devices and manage device memory.

CUDA is a parallel computing platform and application programming interface (API) introduced by Nvidia. It allows software developers to use CUDA-enabled GPUs for general purpose processing. The CUDA platform is accessible to developers through libraries, compiler directives such as OpenACC and extensions to programming languages including C, C++ and Fortran. It also supports other computational interfaces, such as OpenCL.

CUDA offers two programming interfaces: (1) the Runtime API and (2) the Driver API

- The Runtime API is a high-level interface that provides a set of routines and language extensions and offers implicit initialization, context and module

management.

- The Driver API is a low-level interface that offers an additional level of control by exposing lower level concepts such as CUDA contexts, the analogue of host processes for the device, and CUDA modules, the analogue of dynamically loaded libraries for the device. However, using the Driver API requires more code and effort to program and debug.

Figure 2.1 shows CUDA software stack. Most applications do not use the Driver API, as they do not need the additional level of control, and use the Runtime API instead in order to produce more concise code.

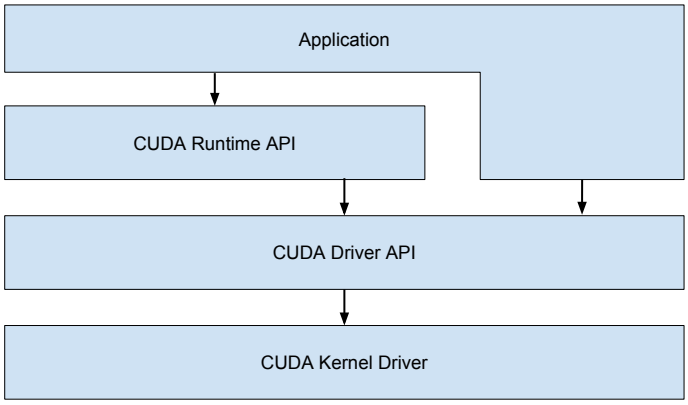

Figure 2.1: CUDA Software Stack

At the core of the CUDA programming model there are three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization. From the view of software developers the execution model is a collection of threads running in parallel. A block is a group of threads executing on a single multiprocessor and dividing its resources equally amongst themselves. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated in Figure 2.2.

A problem is partitioned into coarse sub-problems that can be solved independently in parallel by the blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. CUDA extends the C/C++ programming languages by allowing programmers to define functions (kernels) that, when called, are executed on each thread in parallel. Each thread and block is given a unique ID that can be accessed within the

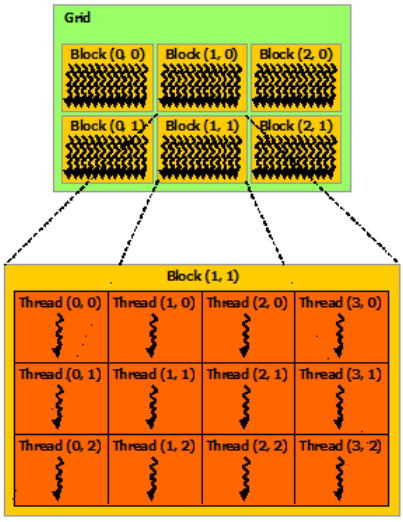

Figure 2.2: Grid of Thread Blocks

thread during its execution, allowing it to perform the kernel task on different set of data.

CUDA exposes its features through a runtime library as well as a set of language extensions. These language extensions allow programmers to define device functions (kernels), configure and execute them on CUDA-enables GPUs. Extensions include function type qualifiers that specify whether a function executes on the host or the device and whether it can be called from the host or the device, variable type qualifiers, that specify the memory location of a variable on the device and build-in variables, that specify dimensions and indices for the GPU's multiple cores. Kernels are configured and launched using the execution configuration extension, denoted with the `<<<...>>>` syntax. A function declared as:

```
__global__ void Func(float* parameter);
```

is called using:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

where Dg specifies the dimension and size of the grid, Db the dimension and size of each thread block and Ns the number of bytes in shared memory that is allocated for this call.

Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable. `ThreadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional,

two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

When executing CUDA programs the GPU operates as co-processor to the main CPU. GPU handles the core processing on large quantities of parallel information while CPU organizes, interprets, and communicates information. CPU manages data transfers between host and device memory and initiates work on GPU. A typical CUDA application's flow is: First allocate required buffers in device memory and copy input data from host to device. Then setup execution configuration and trigger execution on the GPU and, finally, copy back the results to the host memory.

The following sample code adds two vectors `A` and `B` of size `N` and stores the result into vector `C`:

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main()
{
        int i;
        float a[N], b[N], c[N], *dev_a, *dev_b, *dev_c;
        size_t size = N*sizeof(int);

        cudaMalloc((void **)&dev_a, size);
        cudaMalloc((void **)&dev_b, size);
        cudaMalloc((void **)&dev_c, size);

        for (i=0; i<N; i++) {
                a[i] = i;
                b[i] = 2*i;
        }

        cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
```

```
        VecAdd<<<1, N>>>(A, B, C);

        cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost)

        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);

        return 0;
}
```

Source files of CUDA applications contain both host and device code. More specifically, they contain language extensions and device functions that need to be compiled with the `NVCC` compiler. `NVCC` separates device code from host code and compiles the device code into an assembly form of CUDA instruction set architecture (`PTX` code) or binary form (`cubin` objects). Host code is modified by replacing the execution configuration extension (`<<<...>>>`) with the necessary runtime function calls to load and launch each compiled kernel from the `PTX` code or `cubin` object. This procedure is illustrated in Figure 2.3. Device code is loaded from `cubin` or `PTX` files either during initialization from the runtime or explicitly using the Driver API.

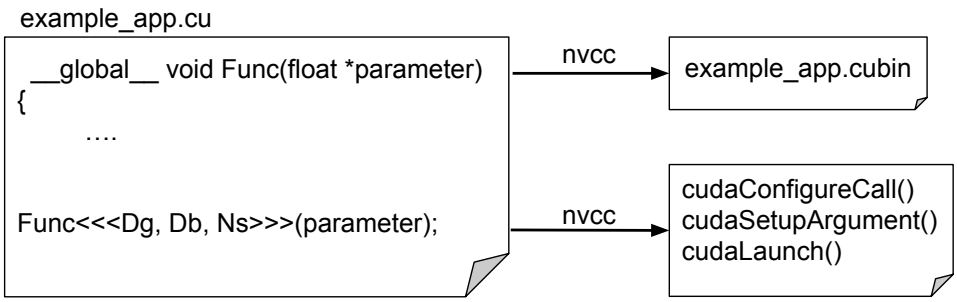

Figure 2.3: Compilation Output

2.3 Virtualization

Software and hardware are traditionally intertwined and inseparable from one another. Traditionally, an application executes on a particular processor, utilizes a specific range of physical memory on the host computer, resides on a specific disk drive and communicates using specific I/O ports. This results in resource conflicts and performance issues when multiple workloads compete for the same physical resources. Virtualization, breaks this relationship between the physical hardware and the logical resources needed for an application's execution by creating a layer of abstraction between a workload and its underlying hardware.

Virtualization roots are traced back to the 1960s when IBM Watson Research Center worked on the M44/44X Project, which had the goal to evaluate the emerging concept of time sharing systems. A number of ground breaking virtualization concepts were implemented in the M44/44X project including partial hardware sharing, memory paging and time sharing. The M44/44X project is generally accredited with the first use of the term virtual machine. The existence of virtualization as a concept went largely unremarked for the nearly two decades of the rise of client/server on x86 platforms.

The term virtualization refers to the act of creating a virtual version of a device or resource, such as a server, operating system, storage device or network resources. Devices, applications and users are able to interact with the virtual resource as if it was a real single logical resource. Virtualization is applied to a wide range of system layers, including system-level virtualization, hardware-level virtualization and application virtualization.

Hardware virtualization refers to the abstraction of computing resources from the software that uses those resources. Hardware virtualization installs a hypervisor or virtual machine manager (VMM), which creates this abstraction layer and manages the underlying hardware. Once a hypervisor is in place, software relies upon virtual representations of the computing components. Virtualized resources can be utilized be isolated instances called virtual machines (VMs) where operating systems and applications can be installed.

2.3.1 Virtual Machine

A virtual machine (VM) can be defined an operating system or application environment that is installed on software which imitates dedicated hardware. The

end user has the same experience on a virtual machine as they would have on dedicated hardware. The hypervisor emulates the host's CPU, memory, hard disk, network and other hardware resources, enabling virtual machines to share resources. Virtual machines are separated into two classes: (1) system virtual machines and (2) process virtual machines.

- System virtual machines provide a complete system platform simulating the complete system hardware stack and supporting the execution of complete operating systems. They usually emulate an existing architecture, and are built with the purpose of either providing a platform to run programs where the real hardware is not available for use, or having multiple instances of virtual machines leading to more efficient use of computing resources.

- A process virtual machine adds a layer over an operating system simulating the programming environment for the execution of an individual process. It is created when the process is started, runs as a normal application on the host OS and is destroyed when it exits. Virtual machines of this class are usually closely suited a programming language and their purpose is to provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system, therefore allowing a program to execute in the same way on any platform. A popular example of this type of VM is the Java Virtual Machine.

2.3.2 Hypervisor

A hypervisor or virtual machine monitor (VMM) is a piece of software that allows a single computer to support multiple, identical execution environments. It enables the creation and management of virtual machines. Multiple instances of different operating systems executing on virtual machines may share the virtualized hardware resources. Hypervisors can be classified into three types:

- Type-0 hypervisors which are a hardware feature and can run multiple guest operating systems, each in a separate hardware partition. This type of VMM is encoded in the firmware and loaded at boot time. It splits a system separate virtual systems, each with dedicated CPUs, memory, and I/O devices. Guest operating systems in a type-0 hypervisor are native operating systems with a subset of hardware made available to them.

- Type-1 or bare-metal hypervisors which run directly on top of hardware. They are special-purpose operating systems that run natively on the hardware, but rather than providing system calls and other interfaces for running

programs, they create, run and manage guest operating systems. Type-1 hypervisors provide better performance and greater flexibility because they operate as a thin layer designed to expose hardware resources to VMs, reducing the overhead required to run the hypervisor itself.

- Type-2 or hosted hypervisors which are installed as a software application on an existing operating system. They support guest virtual machines by coordinating calls for CPU, memory, disk, network and other resources through the physical host's operating system.

2.3.3 Benefits

Virtualization technology offers multiple benefits and has become widely adopted in the information technology industry. Most of these advantages are fundamentally related to the ability to run several different execution environments while sharing the same hardware.

A major advantage of virtual machines in data center use is system consolidation. A typical non-virtualized server usually achieves low utilization but, due to virtualization, multiple lightly used systems can be run in virtual machines on the same host system creating a more heavily used system. As a result, virtual machines can be hosted on fewer physical servers, leading to lower costs for hardware acquisition, maintenance, energy and cooling system usage.

Apart from resource utilization, virtualization can improve resource management as well. Most hypervisors implement live migration allowing a running virtual machine to move from one physical server to another without interrupting its operation or active network connections. This feature creates flexibility by decoupling workloads from the hardware and allowing load balancing. If a server is overloaded, live migration can free resources on the host while not disrupting the guest. Similarly, when host hardware must be repaired or upgraded, guests can be migrated to other servers, and then migrated back, after the host maintenance is complete.

Virtualized environments also facilitate creation and management of virtual machines. Unlike conventional systems that need an operating system, device drivers and application files in order to operate, a VM exists as a single file that can be created and duplicated as needed. This enables creating snapshots of virtual machines' state periodically and storing them. Using those snapshots recovery time after a system failures is accelerated as crashed VMs can be quickly reloaded.

Another advantage of virtual machines is that it facilitates operating system testing and development. Operating systems are complex programs, and as a result a change in one part may cause bugs in other parts. Because the operating system executes in kernel mode, a bug could cause the entire system to crash. Furthermore, because the operating system controls the entire machine, the system is unavailable to users while changes are made and tested. When system development is performed on virtual machines those problem are overcome as the system can be reverted to a previous state after a severe crash simply copying a virtual image. System operation is disrupted only when a completed and tested change is ready to be put into production.

Virtualization has laid the foundation for important advances in computer facility implementation, management, and monitoring. Cloud computing, in which resources such as CPU, memory, and I/O are provided as services to customers using Internet technologies, is made possible by virtualization. By using APIs, programs can create thousands of VMs, using a cloud computing facility, all running a specific guest operating system and application, which others can access via the Internet. This functionality is used by many multiuser games, photo-sharing sites and other web services.

2.3.4 Virtualization Techniques

Trap-and-Emulate

Operating systems provide different levels of access to resources. They offer hierarchical protection mechanisms (protection rings) in order to protect data and functionality from faults and malicious behavior. This is generally hardware-enforced by some processor architectures that provide different CPU modes at the hardware level. Typical systems offer two modes: (1) user mode and (2) kernel mode. In user mode the executing code has no ability to directly access hardware or reference memory, while in kernel mode the executing code has complete and unrestricted access to the underlying hardware.

A virtual machine which runs on a dual-mode system can execute code only in user mode. However, just as a physical machine, the virtual machine has two modes, a virtual user mode and a virtual kernel mode, both of which execute in physical user mode. Operations such as system calls, interrupts and privileged instructions that cause transfer from user to kernel mode on a physical machine must also cause this transfer in a virtual machine.

In the trap-and-emulate method, when the guest kernel attempts to execute

a privileged instruction, an error occurs because the system is in user mode and causes a trap in the physical machine. The VMM gains control, emulates the action that was attempted by the guest kernel on behalf of the guest and returns control to the virtual machine. Using this approach, non-privileged instructions run natively on the hardware providing the same performance, while privileged instructions can decrease performance of the guest system.

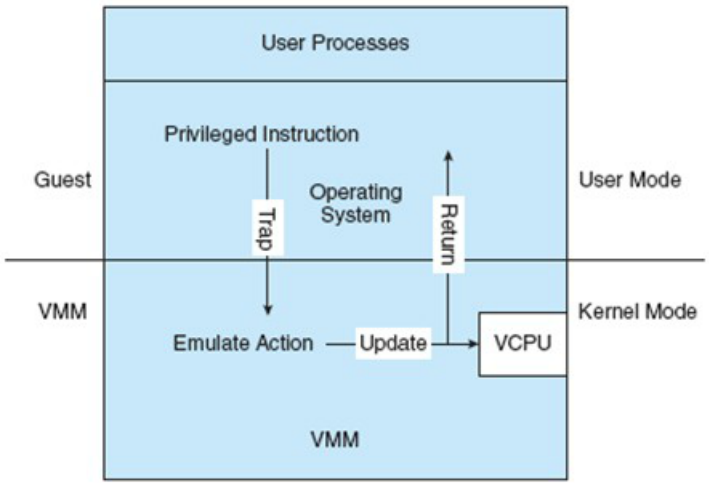

Figure 2.4: Trap-and-Emulate Implementation

Binary Translation

Some CPUs do not have a clean separation of privileged and non-privileged instructions, making the implementation of the trap-and-emulation method challenging. Apart from privileged and non-privileged instruction there is a set of instructions, referred to as special instructions, that can execute both in user and in kernel mode with different behavior. No trap is generated when those instructions are executed in user mode rendering the trap-and-emulate procedure useless.

This problem can be overcome using the binary translation technique. In binary translation, when guest virtual CPU (VCPU) is in user mode, the guest can run its instructions natively on a physical CPU. When guest VCPU is in kernel mode, the VMM examines every instruction the guest executes in virtual kernel mode by reading the next few instructions that the guest is going to execute, based on guest's program counter. Special instructions are translated into a new set of instructions that perform the equivalent task while other instructions are run natively. Binary translation is implemented by reading guest binary instructions

dynamically from the guest and generating native binary code that executes in place of the original code. Performance is improved using a translator cache where the replacement code for each instruction that needs to be translated is cached. All later executions of that instruction are fetched from the translation cache and do not need to be translated again.

Hardware Assistance

Efficiently implementing virtualization is not possible without some level of hardware support. Hardware-assisted virtualization is a platform virtualization approach that enables efficient full virtualization using help from hardware capabilities implemented in modern processors. Those CPUs define two new modes of operation, host and guest. The VMM can enable host mode, define the characteristics of each guest virtual machine, and then switch to guest mode, passing control of the system to a guest operating system. When the guest tries to access a virtualized resource, then control is passed to the VMM to manage that interaction. In addition, CPUs include memory management enhancements and hardware-assisted DMA technologies.

Paravirtualization

Paravirtualization differs from other virtualization approaches because it does not aim to give the system the impression that it is running on physical hardware. It instead presents the guest with a system that is similar but not identical to the native. Paravirtualization does not require virtualization extensions from the host CPU and thus enables virtualization on hardware architectures that do not support hardware-assisted virtualization. However, guests require kernel support and modified device drivers in oder to support this technique. This approach accomplishes more efficient use of resources and a thinner virtualization layer.

The Xen hypervisor, has implemented several paravirtualization techniques in order to optimize performance of both host and guest systems. Xen implements a different approach from other VMMs which present to guests virtual devices that appear to be real devices. Instead it presents device abstractions that allow efficient I/O, implementing communication between the guest and the hypervisor. For each device used by each guest, there is a circular buffer shared by the guest and the hypervisor via shared memory.

2.3.5 QEMU - KVM

Kernel-based Virtual Machine (KVM) [**?**] is a full virtualization solution for Linux on x86 hardware containing virtualization extensions. It is a Linux subsystem which leverages these virtualization extensions to add hypervisor capability to Linux. Using KVM, one can create and run multiple virtual machine.

KVM is structured as a Linux character device, exposing a `/dev/kvm` device node which can be used by userspace programs to create and run virtual machines through a set of system calls. Operations provided by `/dev/kvm` include creation and memory allocation to a new virtual machine, reading and writing virtual CPU registers as well as injecting an interrupt into a virtual CPU.

QEMU [11] is an open source machine emulator. It can run an unmodified target operating system and all its applications in a virtual machine. It has two operating modes: user mode emulation and computer emulation. In user mode emulation, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine by using dynamic translation. When used as a machine emulator, QEMU emulates a full computer system and can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can make use of KVM when running a target architecture that is the same as the host architecture.

The primary usage of QEMU is to run one operating system on another. QEMU is used for debugging as well, since virtual machines can be easily stopped, and their state can be inspected, saved and restored. Moreover, specific embedded devices can be simulated by adding new machine descriptions and new emulated devices.

Each virtual machine that runs using QEMU is an individual QEMU process on the host system. The guest's RAM is mapped in the QEMU process' address space and acts as the physical memory for the guest. KVM allows QEMU to execute guest code directly on the host CPU. In order to execute guest code, a QEMU process opens `/dev/kvm` and issues the `KVM_RUN ioctl()`. When the guest accesses a hardware device register, halts the guest CPU, or performs other special operations, the `ioctl()` returns to QEMU. Then QEMU can emulate the desired outcome or wait for a guest interrupt in the case of a halted guest CPU. The basic flow of a guest CPU is as follows:

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
        ioctl(KVM_RUN)
        switch (exit_reason) {
        case KVM_EXIT_IO:   /* ... */
        case KVM_EXIT_HLT:  /* ... */
        }
}
```

2.3.6 I/O Virtualization

Paravirtualization is a common technology used to virtualize I/O devices. It enables low overhead I/O device virtualization providing efficient communication between host and guest. In this approach virtual hardware is optimized for the virtualization layer and exposes a software interface to the guest, which is similar but not identical to that of the underlying hardware. It is implemented by creating communication channels between hypervisor and guest operating system. Paravirtualized frontend drivers post I/O requests to backend drivers directly, with minimal overhead. To address the issue of having a unified model for those paravirtualized drivers across different virtualization systems, virtio [12] has been proposed. Virtio provides a standardized interface for the development of virtualized devices, as well as a mechanism to support guest-to-hypervisor communication.

Using the interface defined in virtio, guest drivers communicate with the hypervisor by pushing data buffers to a shared queue. The guest posts request buffers, which are processed by the backend to produce corresponding responses. Virtio defines a virtual queue interface that can be used for guest-to-hypervisor communication. Specifically, it implements a shared ring buffer mechanism that enables guests to post buffers which host can consume. Each shared ring has a callback function associated with it, which is called when the hypervisor consumes the buffers. The communication scheme is shown in Figure 2.5. Using the virtio data transport interface, frontend drivers can enqueue buffers to the shared ring and notify the hypervisor. They can then either poll or wait to be notified when results become available through a virtual interrupt. Frontend virtio drivers, including drivers for network and block devices, have been added to the mainline

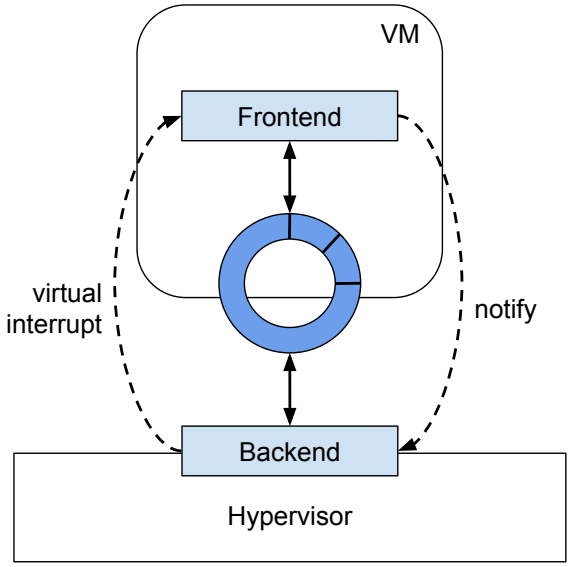

Figure 2.5: Data Transport Mechanism

Linux kernel. Additionally, backend virtio drivers have been implemented for the QEMU software. Those implementations use a data transport channel and a control mechanism which we also use in our approach.

The interface through which frontend and backend drivers communicate is implemented using the `virtqueue struct`. A `virtqueue` is a shared ring into which buffers are posted by the guest for consumption by the host. Each buffer is a scatter-gather array consisting of readable and writable parts. The frontend driver can enqueue requests destined for the backend, in the form of a scatter-gather list. The buffer abstraction is implemented using a `struct scatterlist` array. The array contains `out` entries describing data destined for the backend driver, as well as `in` entries for that driver to store data to return to the frontend driver. Virtio provides a set of functions that enable the guest driver to use the `virtqueue struct` in order to communicate with the hypervisor:

```
struct virtqueue_ops {
        int (*add_buf)(struct virtqueue *vq,
                       struct scatterlist sg[],
                       unsigned int out_num,
                       unsigned int in_num,
                       void *data);
        void (*kick)(struct virtqueue *vq);
```

```
        void *(*get_buf)(struct virtqueue *vq, unsigned int *len);
        void (*disable_cb)(struct virtqueue *vq);
        bool (*enable_cb)(struct virtqueue *vq);
        void *(*detach_unused_buf)(struct virtqueue *vq);
};
```

The basic operation is `add_buf()` which exposes buffers to the hypervisor. The backend driver is notified via the `kick()` call to start processing the buffers. When processing is completed, the guest calls the `get_buf()` function to retrieve the buffers containing results. Polling is supported as `get_buf()` can be called at any time, returning `NULL` no results are available. Guest driver can disable further callbacks using `disable_cb`. Finally, the `detach_buf()` to detach the first unused buffer from the `virtqueue`.

Chapter 3

Design and Implementation

In this chapter we describe our approach's architecture and analyze how virtualization and sharing of the GPU device is accomplished. We design our framework using a paravirtualization approach and employ API redirection in order to enable CUDA application to execute within VMs. The system consists of three software modules: a user level library, a frontend driver located at the guest OS and a backend that represents a virtual CUDA device. We accomplish virtualization by intercepting library calls and redirecting their arguments to the frontend driver. They are afterwards transfered to the backend through a communication channel and executed on the host. Results are eventually returned to the guest. GPU resource sharing is implemented by multiplexing execution requests at the backend side. Future work includes implementing a GPU resource management system that enables scheduling of execution requests posted by multiple guests. Data and control paths are depicted in Figure 3.1. Solid lines represent control path, while dashed lines represent data path.

3.1 Library

CUDA applications access GPU resources through routine calls as well as extensions to the C programming language. Routine calls are implemented in libraries provided by the CUDA SDK. Moreover, language extensions are replaced at compile time by internal function calls, not exposed to the programmer. Using our framework, CUDA applications developed with the Runtime API remain binary compatible since we expose the same interface in our library. In order to implement Runtime API routines in our library, we transfer routine arguments to the backend, where the execution occurs, and receive the execution results.

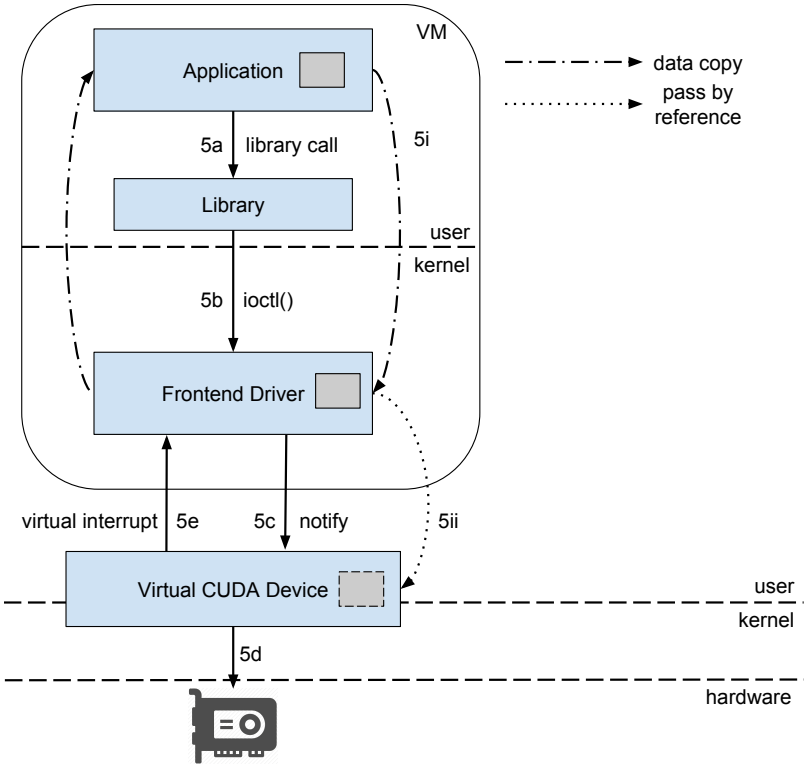

Figure 3.1: Data and Control Path

When a CUDA library call is made by an application (5a), we intercept the arguments, pack them to an execution request among with other required data, such as CUDA contexts, stored by the library, and forward the request to the frontend driver through an `ioctl()` system call (5b). When the system call returns, we unpack the results and return them to the calling process. Furthermore, we accomplish virtualization of the kernel launch syntax (`<<<...>>>`) by implementing the internal routines which replace the extension in our library. We intercept CUDA SDK routine calls and override them with our library's routines using the LD_PRELOAD environment variable.

3.2 Frontend Driver

We design the frontend driver as the intermediate component which forwards execution requests from guest to backend. We implement the frontend driver as a kernel module loaded to the guest Linux kernel. When a library routine makes an `ioctl()` call (5b), the frontend driver handles it by performing appropriate memory allocations and copying the intercepted arguments from user space to kernel space (5i). It then uses the data transport mechanism described in Chapter 2 to make a request to the virtual CUDA device (5c). When processing is complete, the frontend driver copies results back to user space.

We implement two approaches for the mechanism which awaits results from the virtual device, a polling based and an interrupt based. In the former approach the driver repeatedly checks if a buffer has been pushed to the shared ring buffer by the backend, while in the latter one the process' state is changed to interruptible sleep, until a virtual interrupt is received, indicating that a buffer has been pushed to the shared ring. The interrupt handler then pops the buffer from the ring and wakes up the process. We perform evaluation of the aforementioned implementations regarding their performance and CPU utilization.

We enable communication between frontend and backend utilizing the previously described data transport mechanism. More specifically, we issue execution requests to the virtual CUDA device by pushing buffers to the shared ring buffer and notifying the backend. This communication mechanism introduces a limitation. Each data buffer has to be allocated in a physically contiguous way, which is not always feasible, especially in large data transfers. This is a limiting issue in the case of data copies between host and device memory. We therefore develop a mechanism which falls back to a scatter-gather technique with smaller physically contiguous buffers, in case the required memory cannot be allocated contiguously. The backend can then access each piece of memory and reconstruct the original buffer.

Moreover, we implement GPU resource sharing among processes executing concurrently in the same virtual machine. We accomplish it by enabling co-executing processes to access the shared ring buffer concurrently using a synchronization scheme. Each process can independently push requests to the ring buffer and wait for them to be processed.

3.3 Virtual CUDA Device

We design the backend part of the framework as a dispatcher which handles execution requests from multiple co-located VMs. We implement the virtual CUDA device as a QEMU PCI device. The backend component operates as a request handler, receiving requests for routine execution as well as the required arguments and executing them in the host environment. The backend can directly access buffers provided through the data transport mechanism, without copying them. This is possible since the guest's physical address space is accessible from the QEMU process' virtual address space through a translation mechanism. When an execution request is received, we decode it, retrieve required arguments and trigger execution on the GPU (5d).

We eventually handle execution requests at the backend by executing appropriate CUDA Driver API routines. We choose the approach of implementing the Runtime API using Driver API routines, since the Driver API allows explicit context and module management. Explicitly managing module loading and CUDA context switching is required in order to implement GPU resource sharing. We ensure isolation and protection among CUDA applications by switching the CUDA context associated with the calling process to the current one before issuing routine execution, since each CUDA context has its own unique address space. When execution is completed, we push buffers representing execution results to the shared ring buffer and notify the guest by triggering a virtual interrupt (5e).

In order to accomplish sharing of the physical device among processes executing concurrently in co-located VMs, we implement a separate shared ring buffer between the virtual CUDA device and each VM. We treat each request interdependently and multiplex execution requests from co-located virtual machines. By multiplexing execution requests from CUDA application executing concurrently on multiple VMs we enable them to share their access to GPU resources.

3.4 Data and Control Path

The data and control paths are presented in Figure 3.1. CUDA applications allocate memory at guest user space in order to copy data to the device or pass arguments to routine calls. When a routine call is performed by the application, control passes to our library. Library routines and the frontend driver use a common data structure in order to exchange data. Library routines intercept arguments pack them to this structure and transfer control to the frontend driver by performing an `ioctl()` system call. The driver copies required data from user

to kernel space. Subsequently, the frontend driver's implementation packs the appropriate arguments to a buffer, pushes it to the virtual queue, and notifies the backend. Control is thus passed to the backend, while the frontend driver either polls or sleeps waiting for results. The backend can access the exposed buffers without copying them. It executes the appropriate routine and returns control to the frontend driver through a virtual interrupt. The driver then copies results back to the user space and returns to the library, which finally returns to the calling process.

3.5 Runtime API Implementation Details

We implement virtualization of Runtime API routines by intercepting library call arguments and forwarding them to the backend for execution. Standard library routines' implementation is fairly straightforward since there are respective Driver API routines offering the same functionality. However, implementing routine calls that replace the kernel launch extension (`<<<...>>>`) requires more effort, since those routines are not exposed to the programmer and CUDA is proprietary software not providing source code. We therefore employ reverse engineering techniques in order to accomplish virtualization of the kernel launch extension. We perform library call tracing in order to discover declaration of internal routines that implement kernel launching. More specifically, we executed CUDA applications using the `ltrace` tool in order to discover the names of routines which replace the kernel launch extension, as well as routines which perform runtime initialization. We then determined the declarations of those routines through header files provided by CUDA SDK. We implemented the internal routines in our library exposing the same interface in order to intercept their arguments and used debugging techniques to discover how arguments are used. We implement the required functionality by forwarding intercepted arguments to the backend and executing appropriate Driver API routines. Multiple internal routines are used to configure and launch the kernel. In our implementation we gather the appropriate arguments, store them in data structures, and forward them lazily on the last call.

Moreover, before launching a kernel execution, we need to load device code to the GPU from the appropriate CUDA object file. However, the Driver API routine that implements module loading requires the programmer to provide the object file name. Additionally, Runtime API implicitly manages module loading, not exposing the respective file names. We therefore develop a mechanism which, at the beginning of CUDA application execution, searches for `.cubin` files and uses symbol extraction to determine which kernels are defined in each object file. More specifically, during runtime initialization, we list all files in the current directory

using the `ls` command, search for all files with the `.cubin` extension and store them in a linked list structure. CUDA object files contain object code of device functions (kernels) declared in the respective source files. Kernel names are encoded so that they also contain the number of arguments they use as well as each argument's type. We therefore use the `nm` command to list symbols contained in `.cubin` files and employ pattern matching to decode the kernels' names from the given symbols. Subsequently, we store the mapping between object files and kernel declarations in a data structure in our library. Furthermore, kernels are internally referenced in functions using different names than the ones they are declared with. We discover and store the mapping between kernel names and internal names using arguments from the routine which registers the kernels. When a kernel is launched, referenced with its internal name, we search for the its declared name and then for the object file it is contained and use them to load the corresponding file and register the kernel.

The internal CUDA routines are:

```
void** __cudaRegisterFatBinary(void *fatCubin)

void __cudaUnregisterFatBinary()

void __cudaRegisterFunction(void **fatCubinHandle, const char *hostFun,
                            char *deviceFun, const char *deviceName,
                            int thread_limit, uint3 *tid, uint3 *bid,
                            dim3 *bDim, dim3 *gDim, int *wSize)

cudaError_t cudaConfigureCall(dim3 gridDim, dim3 blockDim, size_t sharedMem,
                              cudaStream_t stream)

cudaError_t cudaSetupArgument(const void *arg, size_t size, size_t offset)

cudaError_t cudaLaunch(const void *func)
```

`__cudaRegisterFatBinary` loads the GPU code representing the kernels used by the application. We can't use the `void *fatCubin` argument since the corresponding Driver API does not offer the same interface. We use this routine to search the current directory and create the mapping between object files and kernel declarations. We use `__cudaUnregisterFatBinary` to free allocated memory before the application terminates. `__cudaRegisterFunction` registers the kernels used by the application. We intercept the kernel names, search our mapping and load the the appropriate `.cubin` files. The `cudaConfigureCall`, `cudaSetupArgument`

and `cudaLaunch` routines implement the kernel launch extension. We intercept the required arguments from those calls and forward them to the frontend driver at the `cudaLaunch` routine execution.

3.6 Isolation and Security

Our implementation ensures protection between applications executing within a VM as well as between separate VMs. We achieve isolation using the mechanism of CUDA contexts. CUDA contexts are the equivalent of CPU processes. Each context has each own unique address space and, as a result, GPU pointer values from different contexts reference different physical memory locations. We associate a context with each application in order to ensure isolation and protection from other applications executing in the same as well as different VMs. Then, we set the current context each time at the backend side based on the calling process.

Our framework has been designed in such a way, that it can be enhanced with more advanced features as future extensions. For instance, a scheduling mechanism could be added in order to ensure fairness among separate VMs. Currently, in our prototype implementation requests are multiplexed on the host in a FIFO order. This can be extended in a future work by introducing different scheduling algorithms which can apply fairness and protect VMs from other poorly configured or malicious VMs that try to hog GPU resources.

3.7 Current Limitations

Although our framework provides transparent access to CUDA devices, it introduces two functionality restrictions. Due to undocumented internal library calls, currently only CUDA Toolkit 5.0 is supported in the guest. CUDA toolkit 7.5 can be used at the backend, where the actual execution occurs. Additionally, CUDA object files, which are used to load device code, need to be available to the host at runtime. They can be either copied before the execution or accessed through a shared file system.

Chapter 4

Experimental Evaluation

All performance evaluations are conducted on a test system consisting of two Intel Xeon X5650 CPUs (@2.66 GHz) with 48 GB of main memory. It is equipped with one Nvidia Tesla M2050 GPU. The host system is running Ubuntu Linux 14.04 distribution with kernel 3.19.0 and Nvidia driver version 352.39. The virtualization software used is QEMU-KVM 2.3. All virtual machines are configured to use one VCPU and 1 GB RAM. The guest OS is Debian 3.16.7 with kernel version 3.16.0.

In order to evaluate the performance of our prototype, we use benchmarks from the official CUDA SDK 7.5 [13] as well as the Rodinia benchmark suite [14]. We select benchmarks to represent a wide range of GPGPU applications, and use varying computational loads, data sizes, and different CUDA features.

We first use synthetic microbenchmarks to compare the two implementations that the frontend uses to wait for results by the backend. Furthermore, we perform breakdown analysis and examine the overhead introduced by the framework's software stack. Subsequently, we use an application to evaluate the corresponding performance using our framework compared to native execution. Finally, we evaluate the scalability of the system as the number of concurrently executing applications in co-located VMs increases.

4.1 Sleep and Busy Wait Implementations

We first perform an evaluation of the two aforementioned mechanism used by the frontend driver to wait for execution results. We use a microbenchmark from CUDA samples, that performs matrix multiplication which allows us to adjust the size of input data. Matrix multiplication is an important operation used in a variety of applications, such as financial and signal processing applications. We compare total execution time as well as CPU utilization for each method. We evaluate the two metrics for a range of input data sizes. Results of this experiment are depicted in Table 4.1.

Results show that for small input sizes the busy wait method performs much better than the sleep method regarding to total execution time. This is expected, since the overhead of triggering a virtual interrupt and executing the interrupt handler is higher compared to polling in a busy wait loop. However, as input size increases the difference in performance becomes negligible. Additionally, when the busy wait method is used, applications fully utilize the CPU throughout their execution, creating unnecessary load to the system. In the case of the sleep method, applications have lower CPU utilization, since they release the CPU during waiting time.

In order to benefit from advantages of both methods, we implement a hybrid approach. For small input sizes we use the busy wait method in order to achieve better performance. In this case, CPU is fully utilized for a short period of time, since backend execution does not usually last long for small input sizes. Conversely, for larger input sizes we use the sleep method in order to achieve low CPU utilization as well as high performance.

Input Size (KB)		8	16	32	64	128	256	512	1024	2048
Busy Wait	Execution Time (ms)	5.0	8.5	8.7	9.0	14.1	34.9	64.1	240.8	463.0
	CPU Usage (%)	100	100	100	100	100	100	100	100	100
Sleep	Execution Time (ms)	13.8	14.2	14.8	15.0	15.3	35.2	64.1	241.1	462.7
	CPU Usage (%)	10	12	10	10	9	9	10	6	7

Table 4.1: Comparison of Sleep and Busy Wait Implementations

4.2 Microbenchmark Performance

We conduct experiments with several benchmarks running in a native environment compared to executing in virtual machines with our framework. We measure the execution time of all CUDA operations and do not include any computation performed on the CPU as our framework introduces overhead only on CUDA operations. The normalized execution times on the native and virtualized environment are presented in Figure 4.1. Experiment results show that performance degradation of BlackScholes (BS), LU Decomposition (LUD) and Back Propagation (BB) (where we executed the GPU kernel 1000 times) benchmarks executing in a VM is negligible compared to the native execution. Their execution time in a VM is 1.74%, 3.32% and 1.56% higher than native respectively. The largest overhead in execution time is 9.23% for the fastWalshTransform (FWT) benchmark. This benchmark has a short kernel launch time (only a third of the total GPU execution time) and thus overhead from memory copy between host and device has a higher impact on its performance, due to the aforementioned copy between user and kernel space. This overhead can be alleviated by applying zero copy techniques such as memory pinning, which can be implemented as future work. Moreover, we observe a 4.40% improvement in CUDA execution time of the matrix multiplication (MM) benchmark. This improvement can be attributed to the conversion of Runtime API to Driver API at the backend.

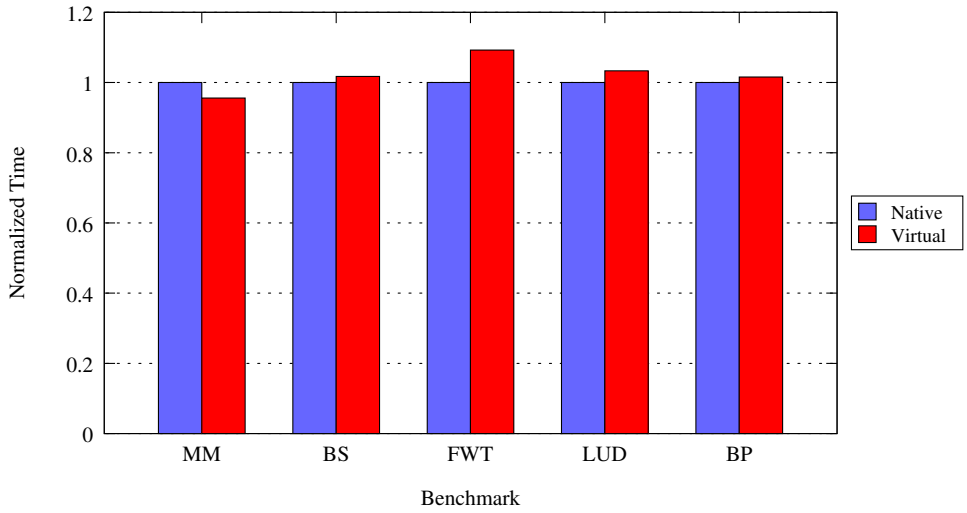

Figure 4.1: Microbenchmark Performance

4.3 Breakdown Analysis

We analyze the overhead introduced by our virtualization framework and perform breakdown analysis of individual CUDA Runtime API routines, using the bandwidthTest benchmark provided by the CUDA samples. Results are shown in Figures 4.2 and 4.3. We choose to perform measurements of the `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost` routines, as they introduce the largest virtualization overhead because of memory copy operations. We divide the execution of a routine into five phases (`lib`, `copy`, `fend`, `exec`, `bend`) representing the different components of the system's software stack as well as operations causing significant overhead. The first phase (`lib`) includes operations performed by the library except the `ioctl()` system call that transfers control to the frontend driver. `Copy` represents the overhead introduced by copying memory from user space to kernel space, while `fend` is the time consumed at the rest of frontend driver's operations. Finally, `exec` is the time of Driver API routine execution at the backend and `bend` is the time spent at the rest of the operations performed by the backend, such as memory allocations, argument unpacking etc.

Regarding the `cudaMemcpyHostToDevice` function, the figure indicates that the dominant factor of execution time is Driver API routine execution at the backend. This phase takes up to 69% of the total execution time for large memory copies. Since this phase represents the actual execution on the GPU, the rest of execution phases can be characterized as the virtualization overhead caused by our framework. Results show that this overhead remains constant at 27% of the

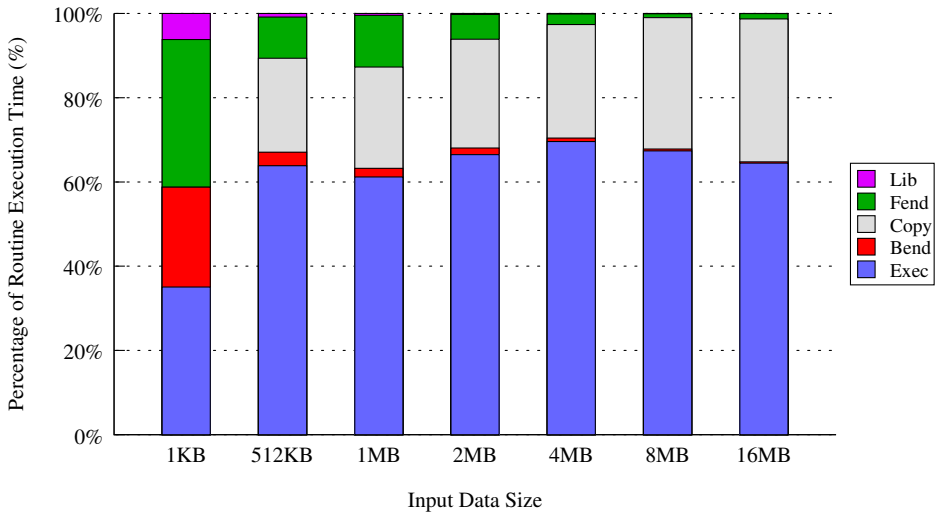

Figure 4.2: Breakdown Analysis: cudaMemcpyHostToDevice

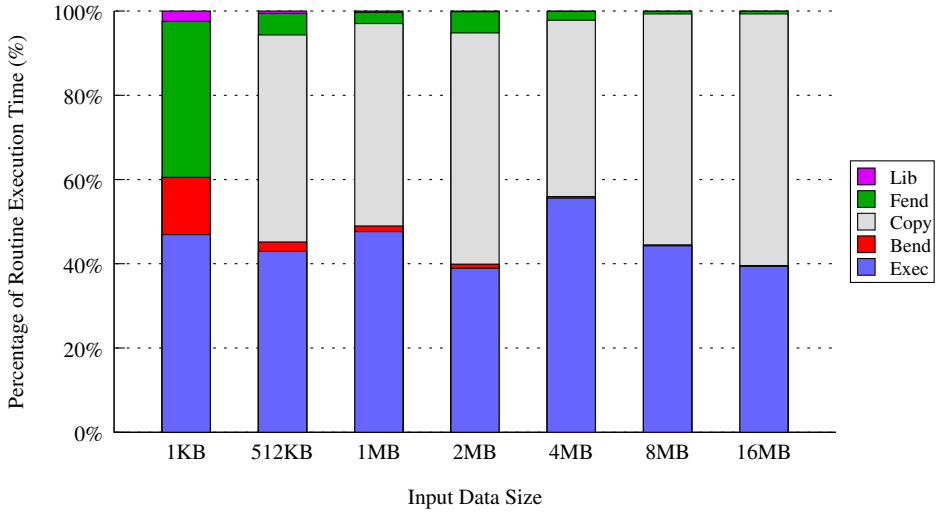

Figure 4.3: Breakdown Analysis: cudaMemcpyDeviceToHost

total execution time on average for memory copies larger than 1 MB. This favors applications in which execution time is dominated by computation on the GPU rather than memory copies. The phase of memory copy from user to kernel space consumes 30% of the total execution time on average and constitutes the major factor causing the overhead. Applying zero copy mechanisms, as mentioned earlier, could lower this overhead. Such techniques can be implemented as future work.

On the other hand, measurements on the `cudaMemcpyDeviceToHost` routine depict that it introduces a larger overhead on execution time. The `exec` phase constitutes a smaller part of the total routine execution time, as it takes up to 55%. This is because the overhead introduced by the system and more specifically the phase of memory copy from kernel to user spaces consumes 50% of the total execution time on average.

Further measurements depict that the rest of library routines introduce a constant overhead. We measure the Driver API routine execution time as well as the overhead of `cudaMalloc` (Table 4.2), `cudaFree` (Table 4.3) and kernel launch (Table 4.4) operations. Results show that the overhead introduced by our framework

Input Size	8 KB	512 KB	1 MB	2 MB	4 MB	8 MB	16 MB
cuMemAlloc (μs)	9	125	124	125	126	127	130
overhead (μs)	23	29	23	24	23	23	24

Table 4.2: cudaMalloc Overhead

remains constant as input data size increases. More specifically the overhead is 24 μs on the `cudaMalloc` routine on average, 26 μs on `cudaFree` and 45 μs on the kernel launch operation. Moreover, this overhead becomes a smaller fraction of the routine execution time as input data size increases, since execution on the GPU lasts longer.

Input Size	8 KB	512 KB	1 MB	2 MB	4 MB	8 MB	16 MB
cuMemFree (μs)	13	132	129	125	141	183	191
overhead (μs)	20	22	23	24	23	25	23

Table 4.3: cudaFree Overhead

Input Size	8 KB	512 KB	1 MB	2 MB	4 MB	8 MB	16 MB
cuLaunchKernel (μs)	14	15	14	14	17	19	22
overhead (μs)	63	57	55	57	75	66	73

Table 4.4: Kernel Launch Overhead

4.4 Impact of Input Data Size

We measure the total execution time of the matrix multiplication benchmark on native as well as virtualized environment for increasing input data sizes in order to study the impact of dataset size on the virtualization overhead by our framework. Figure 4.4 presents the results normalized over the total execution time on the native environment. As seen from the graph there is significant overhead for small input data sizes. Execution time on virtualized environment is approximately double than on native for 8 KB of input and about 40% for 16 KB and 32 KB. This is because the overhead introduced from operations performed by the virtualization framework is significant compared to the actual execution time on the GPU. However, since the execution time on the native environment is short, this overhead does not significantly affect the application's performance. As dataset size increases, execution on the GPU takes longer and the overhead introduced by virtualization consists a smaller part of the total execution time. As a result application's performance when executing on the virtualized environment is very close to the native performance. The lower overhead 2.82% input size of 4 MB.

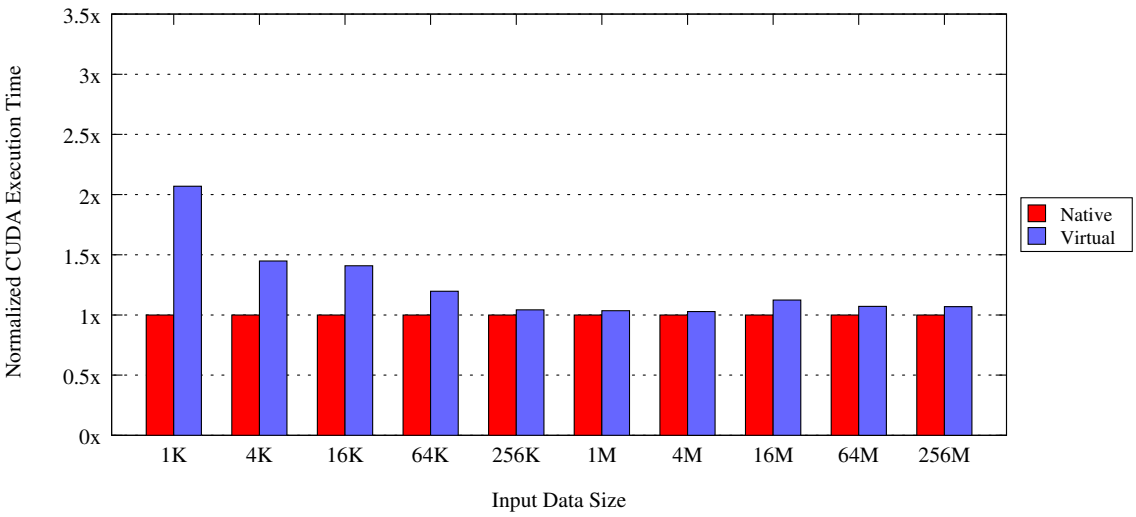

Figure 4.4: Input Size Impact

4.5 Application Performance

We evaluate the robustness and efficiency of the system when used by a higher level application. We use StoreGPU [15], a framework which enables distributed storage system designers to offload hashing-based operations to GPUs. StoreGPU application accelerates compute and data intensive primitives popular in distributed storage system implementations. We execute the application's GPU kernel 10 times and measure the total time of execution as well as execution time of CUDA operations. Figure 4.5 depicts experiment results for execution on native and virtual environment using our framework. As in previous experiments, `Cuda` represents only the CUDA related functions calls, while `Total` represents the total time of execution including both CPU and GPU processing. Results show that the total execution time of StoreGPU in a VM is 7.67% higher than native while CUDA execution time is 4.67% higher.

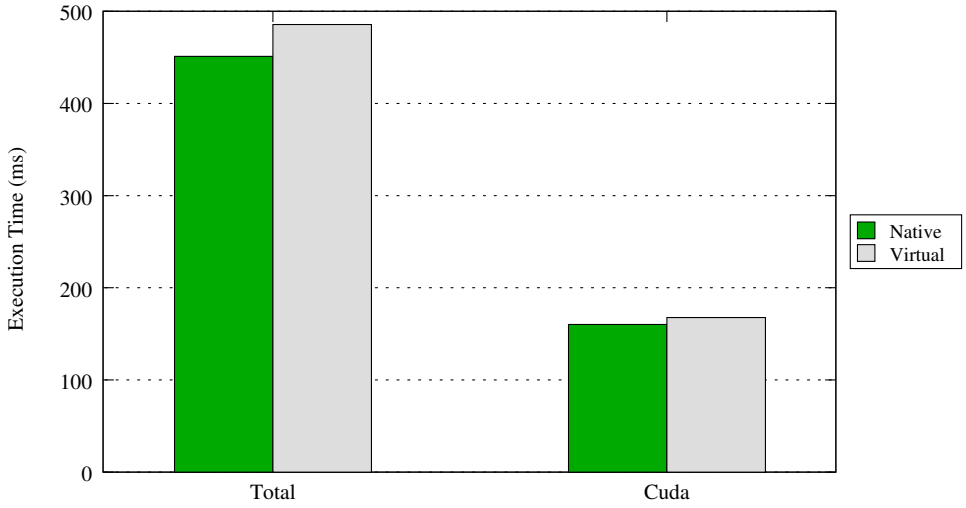

Figure 4.5: Application Performance

4.6 Performance at Scale

We evaluate the overhead the system introduces at multiple concurrently executing GPU contexts by conducting experiments in two setups: (1) we issue multiple processes on the native system executing the same application (`native`) and (2) we launch multiple VMs and execute one application per VM (`virtual`). We measure application's CUDA execution time for these settings and evaluate the overhead introduced by the system, as the number of GPU contexts and VMs increases respectively. We use the BlackScholes benchmark provided by the CUDA samples. Results are depicted in Figure 4.6.

We make two observations based on the results. One is that native execution time increases linearly as the number of GPU contexts increases. This is expected, since legacy GPUs enforce serialization of GPU tasks from different contexts. We later discuss the effect of this GPU characteristic on different types of applications. However, recent Nvidia GPUs (e.g. Kepler [16]) implement actual sharingof GPU resources between concurrently executing CUDA jobs. Another observation is that performance degradation of concurrently executing GPU applications in multiple VMs is negligible compared to native. Operations performed by our framework's software modules, such as copies between user and kernel space, are executed in multiple VMs in parallel and do not introduce additional overhead as the number of VMs increases.

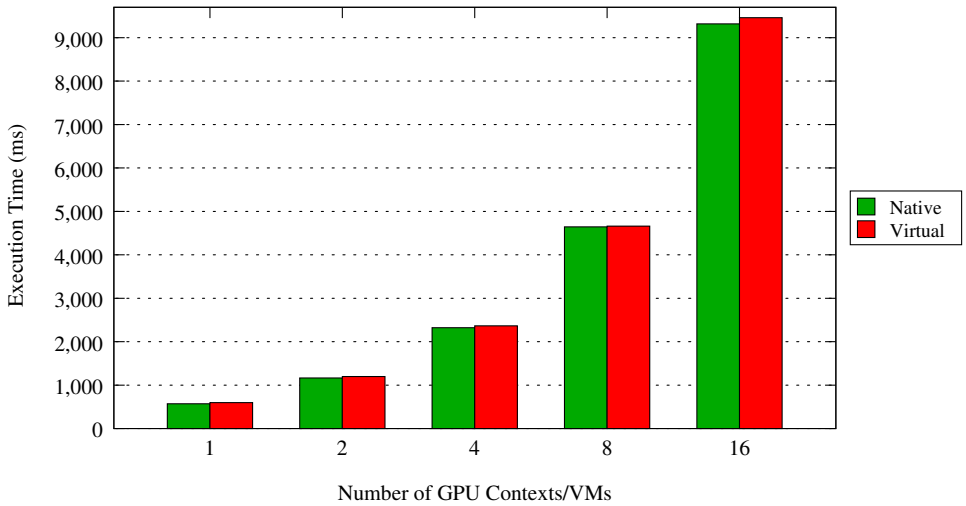

Figure 4.6: Scaling Measurements

4.6.1 Scaling Measurements

GPU applications consist of execution time both on the CPU and the GPU. However, scalability measurements require execution on the GPU to occur concurrently on all processes. We therefore implement and use a synchronization mechanism which places a barrier before execution on the GPU starts, so that after all processes have reach that point, they begin execution on the GPU at the same time.

When performing measurements on the host system this is achieved using signals. A process uses the `fork()` system call in order to create the required number of child processes, which execute the application. Each child process pauses its execution when it reaches the barrier by sending the `SIGSTOP` signal to itself. The parent process detects that all child processes have stopped through the `wait()` system call, and resumes their execution by sending them `SIGCONT` signals.

On the other hand, synchronization between processes executing on separate VMs is achieved using a server-client model and file-based synchronization. All virtual machines share a part of their file system through NFS. A server process, executing on one of the VMs, launches a process on each VM. The server process creates an empty file on a shared folder, and all processes block trying to read from the file, which acts as a barrier. Each process also writes on a separate file informing the server that it has reached the barrier. When all processes have reached the barrier the server writes on the shared file, so that the other processes can read from the file and resume their execution at the same time.

Chapter 5

Related Work

Various approaches to implement virtualization of graphic processing hardware and address GPU resource sharing among co-located virtual machines have been proposed by the research community. The virtualization schemes used by these systems can be classified into I/O pass-through, API forwarding, paravirtualization and full virtualization. API forwarding refers to the technique in which calls to API routines are intercepted and forwarded to a remote host where the actual computation occurs. In the paravirtualization scheme the guest operating system is aware that it is running on a hypervisor and includes drivers that act as frontend, while the hypervisor layer implements backend drivers that represent virtualized devices. On the other hand, in full virtualization the guest OS is unaware that it is being virtualized and requires no changes to work in this configuration. Finally, I/O pass-through provides a VM with direct access to the GPU itself.

5.1 vCuda

Shi et al. [17] propose a GPGPU computing solution that allows applications executing in virtual machines (VMs) to leverage hardware acceleration. vCuda uses a client-server architecture consisted of three user-space components: a user-level library, a data structure used by the library, that represents a virtual GPU, and a server component. The library is responsible for intercepting and redirecting API calls from client to host, where the server executes them and returns the results. Communication between client and server is implemented using the XML-RPC protocol. In addition to virtualization, vCUDA allows device multiplexing among multiple concurrently executing guest OSes by spawning one thread for each client. The framework provides support for suspend and resume as well, enabling client sessions to be interrupted or moved between computers. The system is implemented in Xen but is portable across different virtualization platforms due to

its network transmission mechanism. Experiments in [17] show that time spent in the encoding-decoding steps of the communication protocol causes a considerable negative impact on the overall performance of the solution.

5.2 rCuda

rCUDA [18] is introduced with the goal to provide remote and transparent access to GPU accelerators installed in remote nodes in HPC clusters. rCUDA can thus enhance flexibility in cluster configurations as well as permit a single node to exploit all the GPUs installed in the cluster. The idea to use rCuda as a GPGPU virtualization framework that permits execution of GPU-accelerated applications within virtual machines is explored in [19]. The systems consists of a client middleware which intercepts and forwards API calls to a server middleware. The server, which runs as a service on a computer owning a GPU, receives and executes the API calls from the clients. Communication is accomplished using a protocol based on TCP sockets. The authors also present an optimized implementation of the communication mechanism for InfiniBand interconnects, in order to take advantage of the high speed fabric.

Nanos et al. propose V4VSockets [20], a framework for efficient, low-overhead intra-node communication in the Xen hypervisor. The authors show that rCUDA can be deployed over V4Vsockets to efficiently enable GPU resource sharing among co-located VMs.

5.3 gVirtus

Giunta et al. [21] present a GPU virtualization service focusing on providing transparent access to Nvidia accelerator boards in order to accelerate applications running within VMs. gVirtuS uses a frontend/backend scheme and relies on a pluggable communication component independent of the hypervisor and the communication channel. The system is implemented on VMware and KVM hypervisors. The authors implement three different communication mechanisms. A TCP/IP based communicator was developed in order to check front end - back end interaction. VMCI (VMware Communicator Interface) and vmSocket were used as high performance communicators for VMware and KVM respectively.

5.4 GViM

In their work, GViM, Gupta et al [22] present a system designed to virtualize graphics processors implemented for the Xen hypervisor. The system is organized using a split driver model and employs Xen-specific mechanisms, including shared memory buffers, to implement the communication mechanism. The authors describe a resource management extension for managing applications' joint use of GPU resources. More specifically, GViM implements scheduling of requests destined for the GPU using a round robin as well as a XenoCredit-based scheduling scheme.

5.5 LoGV

Gottschlag et al. propose LoGV [23], an approach to virtualize GPGPUs by leveraging protection mechanisms present in modern hardware. LoGV implements virtualization at a lower level by intercepting and forwarding the API of the `pscnv` GPU driver. This framework uses Gdev CUDA runtime [24] to support the CUDA API. The framework allocates resources securely in the hypervisor and then grants applications direct access to these resources, relying on GPGPU hardware features to guarantee mutual protection between applications. Virtual machine migration is supported, in which the system the suspends access to the GPGPU to extract a consistent snapshot of GPGPU state.

5.6 Distributed-Shared CUDA

In DS-CUDA [25] the authors present a middleware with the goal to address difficulties in programming multi-node heterogeneous computers. The system implements virtualization of a cluster of computers equipped with GPUs so that they appear as if they were attached to a single node, in order to simplify the programming of multi-GPU applications. The system's architecture consists of a single client node and multiple server nodes, in which one or more CUDA devices are installed. Client-server communication uses InfiniBand Verbs, and can also use TCP sockets in case the network infrastructure does not support InfiniBand. DS-CUDA increases the reliability of GPUs by implementing a redundancy mechanism. Identical calculations are performed on multiple CUDA devices, and the results are compared between the redundant calculations. If any of the results do not match an error handler is invoked.

5.7 gVirt

gVirt [26] is a full GPU virtualization solution, implemented in Xen, which allows the native graphics driver to run in the guest system. The system implements mediated pass-through which achieves good performance, scalability and isolation by using the pass-through technique for performance critical resources and the the traps-and-emulate technique for privileged operations. This approach is implemented on Intel Processor Graphics GPU and is oriented and tested on 2D and 3D graphic applications.

5.8 GPUvm

Suzuki et al. [27] propose an architecture based on the Xen hypervisor, that implements both full virtualization and paravirtualization. The authors introduce technologies such as virtual memory-mapped I/O (MMIO), GPU shadow channels, GPU shadow page tables, and virtual GPU schedulers and employ them in the framework's implementation. Experiments in [27] show that performance of GPU paravirtualization is two or three times slower compared to the native system, and the full virtualization exhibits even higher overhead. This approach virtualizes the GPU at a lower level and uses Gdev [24] as the CUDA Runtime.

5.9 Gdev

Gdev [28] is a GPU resource management framework that allows user space as well as the OS to use GPUs as first-class computing resources. The framework implements a virtual memory manager that enables GPU contexts to allocate memory exceeding the physical size of device memory. It also provides a shared device memory functionality that allows GPU contexts to communicate with other contexts. Moreover, Gdev provides a GPU scheduling scheme to virtualize a physical GPU into multiple logical GPUs, enhancing isolation among working sets of multitasking systems.

5.10 Pass Through

A class of solutions makes use of pass-through technology to grant VMs direct access to host devices. Devices on a host PCI-express bus are virtualized using directed I/O virtualization technologies and then direct access to a VM is granted

upon request. A memory management unit is used to handle direct memory access (DMA) coordination and interrupt remapping directly to the guest VM, thus bypassing the host entirely. This approach can minimize the overhead of virtualization, as performance measurements in the Xen platform have indicated [29], but since a pass-throughed GPU is exclusively managed by the guest OS, it does not allow multiple VMs to share the same device. Nvidia GRID [30] technology enables assignment of the physical GPU to multiple VMs at the same time. Gdev [28] is able to virtualize a physical GPU into multiple logical GPUs, which can then be pass-throughed to VMs, thus enabling GPU resource sharing.

Chapter 6

Discussion

6.1 Effect of GPU Resource Sharing on Application Performance

Sharing physical hardware among multiple concurrently executing OSes is a fundamental aspect of hardware virtualization. Our framework enables sharing of GPU resources by multiplexing requests for routine execution at the hypervisor. However, multiplexing applications' accesses on the GPU introduces additional overhead and negatively impacts its performance. Moreover, applications with different execution patterns can be affected differently by sharing their access to the GPU.

GPU accelerated applications can be divided according to their characteristics into different classes. One such class contains applications that can be be characterized as batch jobs. These applications copy large amounts of data from host to device memory and then issue intensive computations without further user interaction. Results are calculated and copied back to host memory before execution is completed. Examples include HPC scientific applications from fields such as bioinformatics [31] and material science [32]. Multiplexing GPU accesses of concurrently executing applications of this class cause performance degradation, due to the inability of legacy GPUs to offer actual resource sharing. The critical performance metric is total execution time. However, the main characteristic of such applications is that their execution time is dominated by GPU resources utilization. Therefore multiplexing GPU accesses of multiple concurrently executing applications decreases performance of all applications, since execution requests from different CUDA contexts are serialized on the GPU. This is an inherent characteristic of legacy GPUs' architecture. Thus, performance degradation of each individual application is negligible in case applications are submitted to run se-

quentially, for instance on a resource scheduling system (e.g. Torque). However, even this class of applications is expected to behave better on modern GPUs.

On the other hand, a different class involves long-running interactive applications which typically begin by copying required data to the device, outside of the critical execution path, and then repeatedly receive smaller amounts of data as input which trigger computations. Examples of applications following this execution pattern include Big Data applications that perform queries on large data sets [33]. Applications of this class sometimes have low latency characteristics and even real-time requirements. The critical performance factor of this class is the response time when input is received. Their execution includes alternations between idle periods user input is awaited, and computation periods when requested results are being calculated. This execution pattern is well fitted for device sharing among concurrently executing applications. Multiplexing their accesses to the GPU is feasible as idle periods of some applications can overlap with computation periods of others.

It is thus evident that the effect of sharing GPU resources can be different according to the application's execution pattern. A GPU resource management system could distinguish between the previously described application classes and schedule their accesses to GPU resources accordingly. Future work involves applying profiling techniques in order to categorize applications and using different scheduling algorithms to multiplex accesses to the GPU. These techniques will also be evaluated on modern GPUs, which provide more advanced sharing features among concurrent tasks.

6.2 Conclusion

In this work we propose a framework for low overhead GPU resource virtualization and sharing among co-located VMs. Our implementation employs API redirection through a split driver approach, in order to allow GPGPU applications to access the physical hardware. Evaluation of our prototype shows that The system achieves near native performance for medium and large data sizes. Moreover, multiple applications executing concurrently in co-located VMs can efficiently share the host GPU.

We design our framework in a way that enables scheduling mechanisms to be easily added to the current version. We plan to implement execution request scheduling in order to achieve quality of service between VMs as well individual applications, in a future work. An extension the backend can implement GPU

resource management and ensure fairness by identifying applications' GPU execution profile and appropriately schedule their access to the GPU. To this end, the mechanism could detect and slow down VMs with high demands on GPU resources. Finally, future endeavors also include evaluating our framework on recent Nvidia GPU with enhanced features regarding resource sharing.

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Σήμερα, λόγω της ζήτησης της αγοράς για υψηλής ανάλυσης 3D γραφικά πραγματικού χρόνου, οι Μονάδες Επεξεργασίας Γραφικών (GPUs) έχουν εξελιχθεί σε υψηλών επιδόσεων πολυπύρηνους επεξεργαστές γενικού σκοπού, ικανούς για υπολογισμούς υψηλής απόδοσης και μεγάλο εύρος ζώνης μνήμης. Εκτός του ότι είναι αποτελεσματικές στον χειρισμό γραφικών και την επεξεργασία εικόνας, η παράλληλη δομή τους τις καθιστά κατάλληλες να επιλύουν προβλήματα τα οποία μπορούν να εκφραστούν με παράλληλους υπολογισμούς και έχουν υψηλή πυκνότητα υπολογισμών σχετικά με την ποσότητα δεδομένων. Ως αποτέλεσμα, οι μονάδες επεξεργασίας γραφικών χρησιμοποιούνται ως επιταχυντές με σκοπό τη βελτίωση της επίδοσης εφαρμογών οι οποίες παραδοσιακά εκτελούνται από την κεντρική μονάδα επεξεργασίας (CPU). Η προσέγγιση αυτή, γνωστή ως εκτέλεση υπολογισμών γενικού σκοπού σε GPUs (GPGPU) υιοθετείται όλο και περισσότερο σε εφαρμογές υπολογισμών υψηλών επιδόσεων (HPC). Ερευνητικές δημοσιεύσεις έχουν υποδείξει ότι εφαρμογές που πραγματοποιούν εντατικούς υπολογισμούς, από ένα ευρύ φάσμα επιστημονικών πεδίων, όπως τα οικονομικά [1], η χημική φυσική [2], η πρόβλεψη καιρού [3], η δυναμική ρευστών [4] κλπ. μπορούν να αξιοποιήσουν GPUs για να αποκτήσουν σημαντικά οφέλη στην επίδοση τους. Εκτός από τον επιστημονικό τομέα, οι GPUs χρησιμοποιούνται σε συστήματα όπως δρομολογητές υλοποιημένοι σε λογισμικό [5], κρυπτογραφημένα δίκτυα [6] καθώς και συστήματα διαχείρισης βάσεων δεδομένων [7]. Ένας από τους λόγους της καθιέρωσης των υπολογισμών γενικού σκοπού σε GPUs είναι η ανάπτυξη προγραμματιστικών περιβάλλοντων, μεταγλωττιστών και βιβλιοθηκών όπως το περιβάλλον CUDA της Nvidia.

Από την άλλη πλευρά, η εικονικοποίηση έχει αυξανόμενη επιρροή στον τρόπο

χρήσης και διαχείρισης των υπολογιστικών πόρων. Η βελτίωση των επιδόσεων του υλικού και η αυξανόμενη ζήτηση για ενοποίηση υπηρεσιών από την αγορά, οδηγεί τα εικονικοποιημένα cloud περιβάλλοντα να φιλοξενούν μία συνεχώς αυξανόμενη ποσότητα υπολογισμών. Οι εικονικές μηχανές (VMs) μπορούν να βελτιώσουν την αξιοποίηση των υπολογιστικών πόρων, καθώς διαφορετικοί πελάτες μπορούν να μοιραστούν ένα κόμβο με την ψευδαίσθηση ότι κατέχουν το σύνολο του μηχανήματος κατά αποκλειστικό τρόπο, παρέχοντας ταυτόχρονα απομόνωση διεργασιών και ευκολία διαχείρισης. Κατά συνέπεια, οι τεχνικές εικονικοποίησης είναι μία υποσχόμενη προσπάθεια για την εκτέλεση λογισμικού HPC σε περιβάλλον cloud, καθώς η δέσμευση εικονικοποιημένων πόρων στο cloud είναι μία ελαστική, αποτελεσματική ως προς τον χρόνο και το κόστος εναλλακτική στον παραδοσιακό τρόπο δέσμευσης πόρων. Με τις πρόσφατες εξελίξεις τόσο στην τεχνολογία της εικονικοποίησης όσο και στην τεχνολογία των GPUs, προκύπτει μία αυξανόμενη ανάγκη παροχής ετερογενών πόρων, ιδιαίτερα GPUs σε περιβάλλοντα cloud, με τον ίδιο επεκτάσιμο και άμεσο τρόπο όπως το παραδοσιακό εικονικοποιημένο υλικό. Οι πάροχοι cloud υπηρεσιών αντιμετωπίζουν επομένως την πρόκληση της ενσωμάτωσης GPGPU στα συστήματα τους. Για παράδειγμα, η υπηρεσία Amazon Elastic Compute Cloud (EC2) [8] προσφέρει GPUs ως υπολογιστικούς πόρους, αλλά κάθε πελάτης εκχωρείται με μία φυσική GPU κατά αποκλειστικό τρόπο. Δυστυχώς, στο πλαίσιο του cloud η εικονικοποίηση συσκευών εισόδου - εξόδου (I/O) παρουσιάζει κακή απόδοση, λόγω της επιβάρυνσης στην επίδοση από την έμμεση πρόσβαση σε φυσικούς πόρους και την ανάγκη πολύπλεξης της πρόσβασης εφαρμογών σε πόρους I/O. Η εικονικοποίηση και ο διαμοιρασμός των GPUs αντιμετωπίζουν επιπλέον προκλήσεις λόγω των χαρακτηριστικών των μονάδων επεξεργασίας γραφικών οι οποίες δεν προσφέρουν δυνατότητες διακοπτής χρονοδρομολόγησης και διαμοιρασμού χρόνου.

1.2 Προτεινόμενη Λύση

Στην εργασία αυτή προτείνουμε μία αποτελεσματική προσέγγιση στην ενσωμάτωση δυνατοτήτων GPGPU σε εικονικές μηχανές. Παρουσιάζουμε τον σχεδιασμό και την υλοποίηση ενός μηχανισμού ο οποίος επιτρέπει σε εφαρμογές που εκτελούνται σε εικονικά περιβάλλοντα να βελτιώσουν την επίδοση τους αξιοποιώντας πόρους της GPU. Χρησιμοποιώντας τον μηχανισμό μας, εικονικές μηχανές οι οποίες εκτελούνται στο ίδιο host σύστημα μπορούν να μοιραστούν τους πόρους της GPU. Για την επαλήθευση των σχεδιαστικών αρχών υλοποιούμε έναν πρωτότυπο μηχανισμό ο οποίος στοχεύει στην εικονικοποίηση μονάδων επεξεργασίας γραφικών της Nvidia, επιτρέποντας έτσι σε εφαρμογές που έχουν αναπτυχθεί χρησιμοποιώντας το περιβάλλον CUDA να εκτελούνται σε εικονικές μηχανές. Η προσέγγιση μας χρησιμοποιεί τεχνικές παρα-εικονικοποίησης καθώς και ένα μοντέλο οδηγού συσκευής (driver) χωρισμένου σε δύο μέρη. Αποτελείται από μία βιβλιοθήκη επιπέδου χρήστη, έναν

frontend driver που βρίσκεται στο guest λειτουργικό σύστημα, και έναν backend driver υλοποιημένο στον hypervisor. Η αρχιτεκτονική του συστήματος απεικονίζεται στο Σχήμα 1.1.

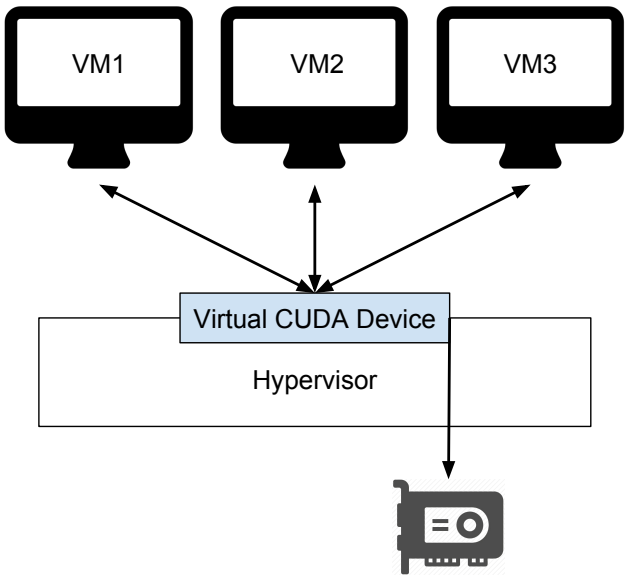

Figure 1.1: Η αρχιτεκτονική του συστήματος

Συνοπτικά, οι κύριες συνεισφορές της εργασίας είναι:

- Προτείνουμε έναν αποτελεσματικό μηχανισμό εικονικοποίησης μονάδων επεξεργασίας γραφικών ο οποίος επιτρέπει σε εφαρμογές GPGPU να εκτελούνται σε εικονικές μηχανές, και υλοποιεί διαμοιρασμό πόρων της GPU μεταξύ εικονικών μηχανών.

- Διατηρούμε συμβατότητα σε επίπεδο εκτελέσιμων αρχείων με εφαρμογές που χρησιμοποιούν το CUDA Runtime API έτσι ώστε υπάρχουσες εφαρμογές να μπορούν να χρησιμοποιούν τον μηχανισμό μας χωρίς καμία τροποποίηση στον πηγαίο κώδικα τους.

- Κατηγοριοποιούμε τις εφαρμογές που χρησιμοποιούν GPUs ως επιταχυντές με βάση το μοτίβο υπολογισμών και πρόσβασης στη μνήμη και αναλύουμε ποια είδη εφαρμογών μπορούν να επωφεληθούν από τη χρήση του μηχανισμού μας.

Η πειραματική αξιολόγηση μας δείχνει ότι ο μηχανισμός εισάγει χαμηλή επιβάρυνση, καθιστώντας την επίδοση εφαρμογών που χρησιμοποιούν επιτάχυνση από GPUs και εκτελούνται σε εικονικές μηχανές, ανταγωνιστική με την επίδοση εφαρμογών

που εκτελούνται σε πραγματικά συστήματα με πρόσβαση στους φυσικούς πόρους της GPU.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Προγραμματιστικά Περιβάλλοντα GPGPU

Τα CUDA (Compute Unified Device Architecture) [9] και OpenCL (Open Computing Language) [10] είναι δύο ευρέως χρησιμοποιούμενα περιβάλλοντα τα οποία προσφέρουν δυνατότητες υπολογισμών γενικού σκοπού σε GPUs. Και τα δύο προσφέρουν παρόμοια χαρακτηριστικά αλλά μέσω διαφορετικών προγραμματιστικών διεπαφών. Το OpenCl, το οποίο αναπτύχθηκε από την ομάδα Khronos Group, είναι ένα ανοικτό πρότυπο για παράλληλο προγραμματισμό σε ετερογενή συστήματα που αποτελούνται από CPUs, GPUs, ψηφιακούς επεξεργαστές σήματος (DSPs) και άλλους τύπους επεξεργαστών ή επιταχυντών. Το CUDA είναι ένα περιβάλλον παράλληλου προγραμματισμού που έχει αναπτυχθεί από την Nvidia. Προσφέρει μία μη ανοικτή προγραμματιστική διεπαφή (API) και ένα σύνολο επεκτάσεων σε γλώσσες προγραμματισμού τα οποία μπορούν να χρησιμοποιηθούν για την πραγματοποίηση υπολογισμών γενικού σκοπού σε GPUs.

Το CUDA προσφέρει δύο προγραμματιστικές διεπαφές: (1) το Runtime API και (2) το Driver API. Το Runtime API είναι μία διεπαφή υψηλού επιπέδου που παρέχει ένα σύνολο από συναρτήσεις και επεκτάσεις γλωσσών προγραμματισμού και προσφέρει αυτοματοποιημένη αρχικοποίηση και διαχείριση των contexts και modules της GPU. Το Driver API είναι μία διεπαφή χαμηλού επιπέδου που προσφέρει ένα επιπλέον επίπεδο ελέγχου εκθέτοντας έννοιες χαμηλότερου επιπέδου όπως CUDA contexts, το ανάλογο των διεργασιών για της CPU, και CUDA modules, το ανάλογο των δυναμικών βιβλιοθηκών. Ωστόσο, η χρήση του Driver API απαιτεί περισσότερο κώδικα και μεγαλύτερη προσπάθεια για τον προγραμματισμό και τον εντοπισμό σφαλμάτων. Το σχήμα 2.1 απεικονίζει την στοίβα λογισμικού CUDA. Οι εφαρμογές στην πλειοψηφία τους δεν χρησιμοποιούν το Driver API καθώς δεν χρειάζονται το επιπλέον

επίπεδο ελέγχου, αλλά το Runtime API ώστε να παράγουν πιο συνοπτικό κώδικα.

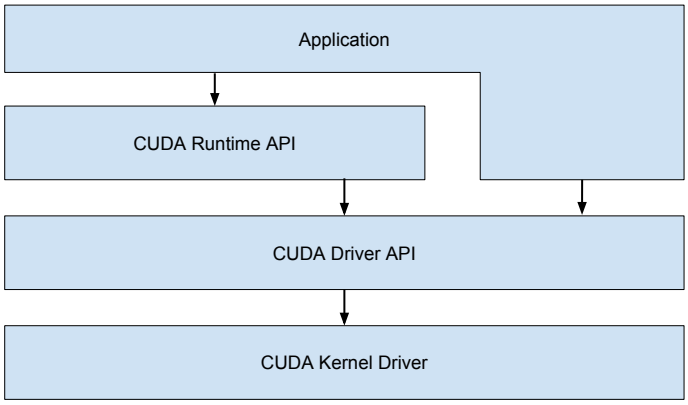

Figure 2.1: Η στοίβα λογισμικού CUDA

Το CUDA εκθέτει τα χαρακτηριστικά του μέσω μίας βιβλιοθήκης χρόνου εκτέλεσης καθώς και ενός συνόλου επεκτάσεων γλωσσών προγραμματισμού. Οι επεκτάσεις αυτές επιτρέπουν στους προγραμματιστές να δηλώνουν συναρτήσεις (kernels) και να ρυθμίζουν την εκτέλεση τους στην GPU. Οι επεκτάσεις περιλαμβάνουν στοιχεία της γλώσσας που καθορίζουν αν μία συνάρτηση εκτελείται στην CPU ή την GPU και αν μπορεί να κληθεί από την CPU ή την συσκευή, στοιχεία της γλώσσας που καθορίζουν τον τύπο της μνήμης όπου αποθηκεύεται μία μεταβλητή στην GPU και ειδικές μεταβλητές που καθορίζουν διαστάσεις και δείκτες σχετικούς με την εκτέλεση στους πολλαπλούς πυρήνες της GPU. Οι kernels ρυθμίζονται και εκτελούνται με την επέκταση διαμόρφωσης εκτέλεσης, η οποία συμβολίζεται ως `<<<...>>>`. Μία τέτοια συνάρτηση δηλώνεται ως εξής:

```
__global__ void Func(float* parameter);
```

και καλείται:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

όπου το Dg καθορίζει τη διάσταση και το μέγεθος του πλέγματος των block νημάτων, το Db τη διάσταση και το μέγεθος του κάθε block νημάτων και το Ns τον αριθμό των bytes στην μοιραζόμενη μνήμη που δεσμεύεται για την παρούσα κλήση συνάρτησης.

Τα αρχεία πηγαίου κώδικα των εφαρμογών CUDA περιέχουν κώδικα τόσο για την CPU όσο και για την GPU. Πιο συγκεκριμένα, περιέχουν επεκτάσεις γλώσσας και συναρτήσεις συσκευής οι οποίες χρειάζεται να μεταγλωττιστούν με χρήση του `NVCC` compiler. Ο `NVCC` διαχωρίζει τον κώδικα της συσκευής από εκείνον της CPU

και μεταγλωττίζει τον GPU κώδικα σε κώδικα assembly αρχιτεκτονικής συνόλου εντολών CUDA (κώδικα `PTX`) ή σε δυαδική μορφή (αρχεία αντικειμένων `cubin`). Ο κώδικας της CPU τροποποιείται αντικαθιστώντας της επέκτασης διαμόρφωσης εκτέλεσης (`<<<...>>>`) με τις απαραίτητες κλήσεις συναρτήσεων για την φόρτωση και την εκτέλεση ενός πυρήνα από `PTX` κώδικα η αρχείο αντικειμένων `cubin`. Η διαδικασία αυτή απεικονίζεται στο σχήμα 2.2. Ο κώδικας για τη GPU φορτώνεται από αρχεία `cubin` ή `PTX` κατά την αρχικοποίηση του περιβάλλοντος εκτέλεσης είτε ρητά χρησιμοποιώντας το Driver API.

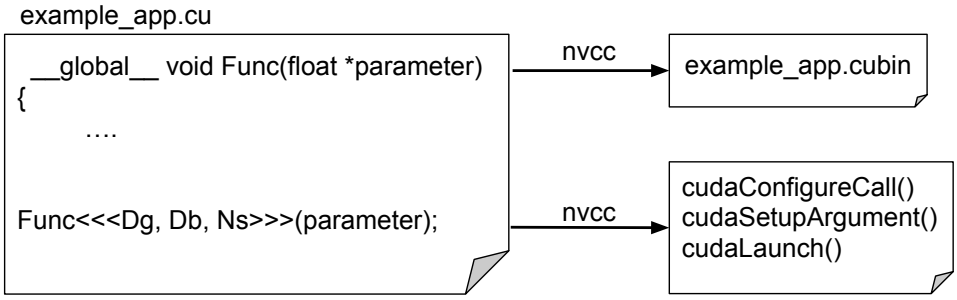

Figure 2.2: Διαδικασία Μεταγλώττισης

2.2 Εικονικοποίηση Συσκευών Εισόδου/Εξόδου

Η παρα-εικονικοποίηση είναι μία τεχνολογία που χρησιμοποιείται ευρέως στην εικονικοποίηση συσκευών εισόδου/εξόδου (I/O). Επιτρέπει την εικονικοποίηση συσκευών I/O με χαμηλή επιβάρυνση στην επίδοση, παρέχοντας αποτελεσματική επικοινωνία μεταξύ host και guest. Στην προσέγγιση αυτή, το εικονικό υλικό έχει βελτιστοποιηθεί με σκοπό την εικονικοποίηση και εκθέτει στο guest σύστημα μία διεπαφή η οποία είναι παρόμοια αλλά όχι ταυτόσημη με εκείνη του υποκείμενου υλικού. Υλοποιείται με τη δημιουργία διαύλων επικοινωνίας μεταξύ του hypervisor και του guest λειτουργικού συστήματος. Frontend drivers που υλοποιούν παρα-εικονικοποίηση αποστέλλουν αιτήματα για I/O στους backend drivers απευθείας, με ελάχιστη επιβάρυνση. Για την αντιμετώπιση του ζητήματος της ύπαρξης ενός ενιαίου προτύπου για τους drivers που εφαρμόζουν παρα-εικονικοποίηση σε διαφορετικά συστήματα εικονικοποίησης, έχει προταθεί το σύνολο drivers και μηχανισμών virtio [12]. Το virtio παρέχει μία τυποποιημένη διεπαφή για την υλοποίηση εικονικών συσκευών, καθώς και έναν μηχανισμό που υποστηρίζει επικοινωνία μεταξύ guest και hypervisor.

Χρησιμοποιώντας τη διεπαφή που ορίζεται στο virtio οι drivers του guest συστήμαστος επικοινωνούν με τον hypervisor τοποθετώντας buffers σε μία μοιραζόμενη

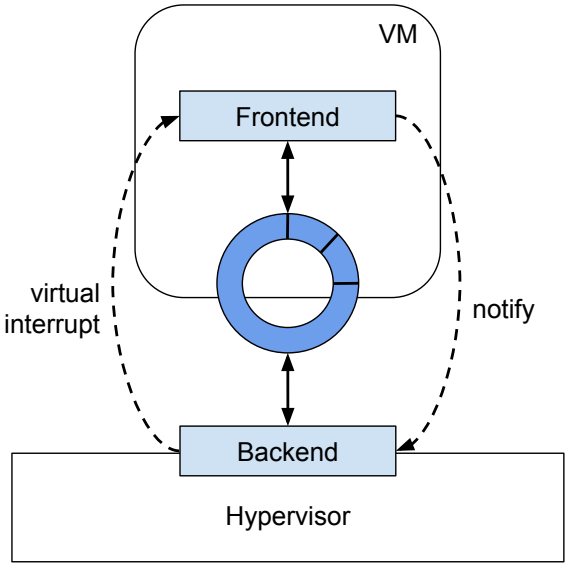

Figure 2.3: Data Transport Mechanism

ουρά. Το guest σύστημα αποστέλλει buffers που αντιπροσωπεύουν αιτήματα εκτέλεσης, τα οποία περιέχουν δεδομένα πάνω στα οποία πραγματοποιείται επεξεργασία στο backend. Το virtio ορίζει μία διεπαφή ουράς η οποία μπορεί να χρησιμοποιηθεί για επικοινωνία μεταξύ guest και hypervisor. Συγκεκριμένα, υλοποιεί έναν μηχανισμό μοιραζόμενου δακτυλίου που δίνει τη δυνατότητα στον host να τοποθετεί buffers τους οποίους ο guest μπορεί να παραλάβει. Ο δακτύλιος συνδέεται με μία συνάρτηση η οποία καλείται όταν ο hypervisor παραλαμβάνει buffers. Ο μηχανισμός επικοινωνίας απεικονίζεται στο σχήμα 2.3. Χρησιμοποιώντας την διεπαφή μεταφοράς δεδομένων του virtio, οι frontend drivers μπορούν να εισάγουν buffers στον δακτύλιο και να ενημερώνουν τον hypervisor. Στη συνέχεια μπορούν είτε να ελέγχουν επαναληπτικά για αποτελέσματα είτε να περιμένουν για ενημέρωση μέσω εικονικής διακοπής, όταν υπάρχουν διαθέσιμα αποτελέσματα. Frontend virtio drivers, συμπεριλαμβανομένων drivers για συσκευές δικτύου και συσκευές block, έχουν προστεθεί στον πυρήνα του λειτουργικού Linux. Επιπροσθέτως, backend virtio drivers έχουν υλοποιηθεί για το λογισμικό QEMU. Οι υλοποιήσεις αυτές χρησιμοποιούν έναν δίαυλο μεταφοράς δεδομένων και έναν μηχανισμό ελέγχου τους οποίους χρησιμοποιούμε και εμείς στην υλοποίηση μας.

Κεφάλαιο 3

Σχεδιασμός και υλοποίηση

Στο κεφάλαιο αυτό περιγράφουμε την αρχιτεκτονική του μηχανισμού μας και αναλύουμε πως επιτυγχάνεται η εικονικοποίηση και ο διαμοιρασμός της GPU. Για τη σχεδίαση του συστήματος χρησιμοποιούμε τις τεχνικές της παρα-εικονικοποίησης και της ανακατεύθυνσης του API έτσι ώστε να καταστήσουμε δυνατή την εκτέλεση εφαρμογών CUDA σε εικονικές μηχανές. Ο μηχανισμός αποτελείται από τρία μέρη: μία βιβλιοθήκη χώρου χρήστη, έναν frontend driver που βρίσκεται στο guest λειτουργικό σύστημα και έναν backend driver ο οποίος αντιπροσωπεύει μία εικονική συσκευή CUDA. Επιτυγχάνουμε την εικονικοποίηση παραλαμβάνοντας κλήσεις βιβλιοθήκης και ανακατευθύνοντας τα ορίσματα τους στον frontend driver. Στη συνέχεια, τα ορίσματα μεταφέρονται στο backend μέσω ενός διαύλου επικοινωνίας και οι κλήσεις βιβλιοθήκης εκτελούνται στον host. Τα αποτελέσματα επιστρέφονται τελικώς μέσω του διαύλου επικοινωνίας στην καλούσα διεργασία. Ο διαμοιρασμός πόρων της GPU υλοποιείται εφαρμόζοντας πολύπλεξη των αιτημάτων εκτέλεσης στο backend. Μελλοντικές επεκτάσεις της εργασίας περιλαμβάνουν την υλοποίηση ενός μηχανισμού διαχείρισης πόρων της GPU ο οποίος θα επιτρέπει την χρονοδρομολόγηση αιτημάτων εκτέλεσης από πολλαπλές εικονικές μηχανές. Οι ροές δεδομένων και εκτέλεσης απεικονίζονται στο σχήμα 3.1. Οι συνεχείς γραμμές αντιπροσωπεύουν τη ροή ελέγχου ενώ οι διακεκομμένες τη ροή δεδομένων.

3.1 Βιβλιοθήκη

Οι εφαρμογές CUDA έχουν πρόσβαση σε πόρους της GPU μέσω συναρτήσεων καθώς και επεκτάσεων της γλώσσας προγραμματισμού C. Οι συναρτήσεις είναι υλοποιημένες σε βιβλιοθήκες που παρέχονται από το περιβάλλον προγραμματισμού CUDA. Επιπλέον, οι επεκτάσεις γλώσσας αντικαθιστώνται κατά τη μεταγλώττιση από κλήσεις

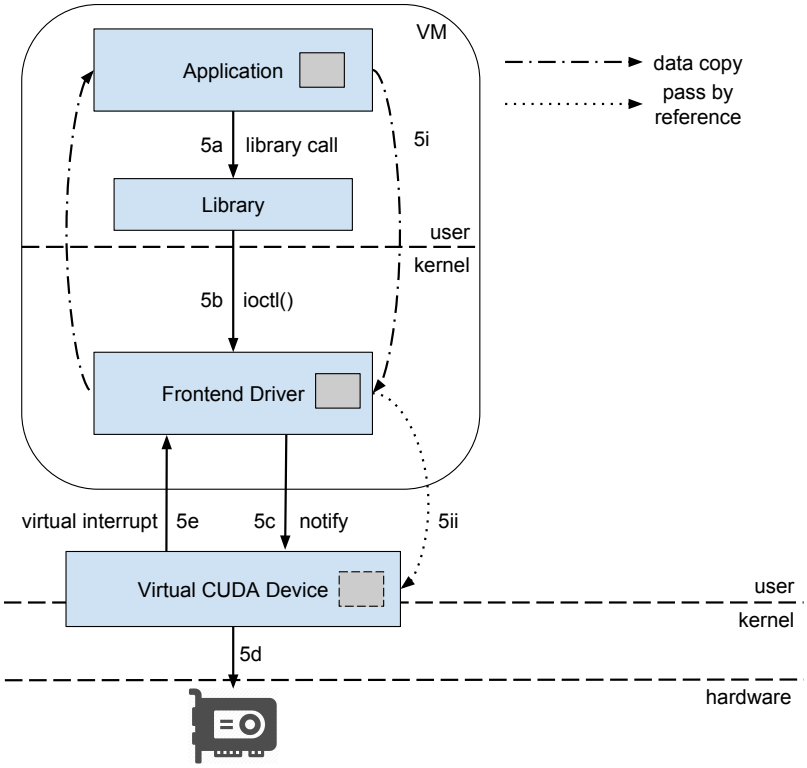

Figure 3.1: Ροές Δεδομένων και Εκτέλεσης

σε εσωτερικές συναρτήσεις οι οποίες δεν είναι διαθέσιμες στον προγραμματιστή. Χρησιμοποιώντας τον μηχανισμό μας, οι εφαρμογές CUDA που έχουν αναπτυχθεί με χρήση του Runtime API παραμένουν συμβατές για εκτέλεση καθώς η βιβλιοθήκη μας εκθέτει την ίδια διεπαφή με εκείνη της βιβλιοθήκη CUDA. Προκειμένου να υλοποιήσουμε τις συναρτήσεις του Runtime API στη βιβλιοθήκη μας, μεταφέρουμε τα ορίσματα των συναρτήσεων στο backend όπου λαμβάνει χώρα η εκτέλεση, και παραλαμβάνουμε τα σχετικά αποτελέσματα.

Όταν πραγματοποιείται μία κλήση βιβλιοθήκης CUDA από μία εφαρμογή (5a), παραλαμβάνουμε τα ορίσματα, τα συγκεντρώνουμε σε ένα αίτημα εκτέλεσης μαζί με άλλα απαιτούμενα δεδομένα, όπως CUDA contexts τα οποία αποθηκεύονται στην βιβλιοθήκη, και διαβιβάζουμε το αίτημα στον frontend driver μέσω μίας κλήσης συστήματος `ioctl()` (5b). Όταν μία κλήση συστήματος επιστρέφει, παραλαμβάνουμε τα αποτελέσματα και τα επιστρέφουμε στην καλούσα διεργασία. Επιπροσθέτως, επιτυγχάνουμε εικονικοποίηση της επέκτασης εκτέλεσης πυρήνα (`<<<...>>>`) υλοποιώντας στη βιβλιοθήκη μας τις εσωτερικές συναρτήσεις οι οποίες αντικαθιστούν την επέκταση αυτή. παραλαμβάνουμε τις κλήσεις συναρτήσεων του CUDA SDK και

τις αντικαθιστούμε με τις συναρτήσεις της βιβλιοθήκης μας χρησιμοποιώντας την μεταβλητή περιβάλλοντος LD_PRELOAD.

3.2 Frontend Driver

Σχεδιάζουμε τον frontend driver ως το ενδιάμεσο τμήμα του μηχανισμού το οποίο προωθεί τα αιτήματα εκτέλεσης από την guest διεργασία στο backend. Υλοποιούμε τον frontend driver ως ένα module το οποίο φορτώνεται στον πυρήνα του Linux. Όταν μία συνάρτηση βιβλιοθήκης πραγματοποιεί μία κλήση `ioctl()` (5b), ο frontend driver πραγματοποιεί τις κατάλληλες δεσμεύσεις μνήμης και αντιγράφει τα ορίσματα από τον χώρο χρήστη στον χώρο πυρήνα (5i). Στη συνέχεια χρησιμοποιεί τον μηχανισμό μεταφοράς δεδομένων που περιγράφεται στο κεφάλαιο 8 για να πραγματοποιήσει ένα αίτημα στην εικονική συσκευή CUDA (5c). Όταν ολοκληρωθεί η επεξεργασία ο frontend driver αντιγράφει τα αποτελέσματα πίσω στον χώρο χρήστη.

Υλοποιούμε δύο προσεγγίσεις του μηχανισμού παραλαβής αποτελεσμάτων από την εικονική συσκευή, έναν ο οποίος βασίζεται σε επαναληπτικό έλεγχο και έναν ο οποίος βασίζεται σε διακοπές. Στην πρώτη προσέγγιση ο driver ελέγχει σε βρόγχο αν κάποιος buffer έχει προστεθεί στον μοιραζόμενο δακτύλιο από το backend, ενώ στην δεύτερη η διεργασία μεταβαίνει σε κατάσταση αδράνειας έως ότου ληφθεί μία εικονική διακοπή, υποδεικνύοντας την προσθήκη buffer στον δακτύλιο. Ο χειριστής διακοπών τότε εξάγει τον buffer από τον δακτύλιο και επαναφέρει τη διεργασία σε κατάσταση εκτέλεσης. Πραγματοποιούμε πειραματική αξιολόγηση των προαναφερθέντων υλοποιήσεων σχετικά με την επίδοση τους καθώς και τη χρησιμοποίηση της CPU.

Υλοποιούμε την επικοινωνία μεταξύ frontend και backend χρησιμοποιώντας τον μηχανισμό μεταφοράς δεδομένων ο οποίος περιγράφεται παραπάνω. Πιο συγκεκριμένα, πραγματοποιούμε αιτήματα στην εικονική συσκευή CUDA προσθέτοντας buffers στον μοιραζόμενο δακτύλιο και ειδοποιώντας στη συνέχεια το backend. Ο εν λόγω μηχανισμός επικοινωνίας εισάγει έναν περιορισμό. Κάθε buffer δεδομένων πρέπει να δεσμεύεται σε διαδοχικές φυσικές θέσεις μνήμης, κάτι το οποίο δεν είναι πάντα εφικτό, ειδικά σε μεγάλες μεταφορές δεδομένων. Αυτός είναι ένας περιοριστικός παράγοντας στην περίπτωση της μεταφοράς δεδομένων μεταξύ της CPU και της μνήμης της συσκευής. Αναπτύσσουμε επομένως ένα μηχανισμό ο οποίος, στην περίπτωση που η απαιτούμενη μνήμη δεν μπορεί να δεσμευτεί με συνεχόμενο τρόπο, καταφεύγει σε μία scatter-gather τεχνική η οποία χρησιμοποιεί buffers δεσμευμένους σε διαδοχικές φυσικές θέσεις μνήμης. Το backend μπορεί στη συνέχεια να έχει πρόσβαση σε κάθε τμήμα μνήμης και να ανακατασκευάσει τον αρχικό buffer.

Επιπλέον υλοποιούμε διαμοιρασμό των πόρων της GPU μεταξύ διεργασιών που εκτελούνται ταυτόχρονα στην ίδια εικονική μηχανή. Αυτό επιτυγχάνεται επιτρέποντας σε πολλαπλές διεργασίες να έχουν πρόσβαση στον ίδιο μοιραζόμενο δακτύλιο ταυτόχρονα, χρησιμοποιώντας έναν μηχανισμό συγχρονισμού. Κάθε διεργασία μπορεί να προσθέτει αιτήματα στον δακτύλιο ανεξάρτητα και να περιμένει να υποστούν επεξεργασία.

3.3 Εικονική Συσκευή CUDA

Σχεδιάζουμε το backend μέρος του μηχανισμού μας ως έναν διεκπεραιωτή ο οποίος διαχειρίζεται τα αιτήματα εκτέλεσης από πολλαπλά VMs τα οποία εκτελούνται στον ίδιο host. Υλοποιούμε την εικονική συσκευή CUDA ως μία συσκευή QEMU PCI. Το backend μέρος λειτουργεί ο διαχειριστής αιτημάτων, δεχόμενο αιτήματα για την εκτέλεση συναρτήσεων καθώς και τα απαιτούμενα ορίσματα και εκτελώντας τες στο host περιβάλλον. Το bacend έχει απευθείας πρόσβαση σε buffers που παρέχονται μέσω του μηχανισμού μεταφοράς δεδομένων, χωρίς να απαιτείται αντιγραφή τους. Αυτό είναι δυνατό, επειδή ο χώρος φυσικών διευθύνσεων του guest είναι προσβάσιμος από τον χώρο εικονικών διευθύνσεων της διεργασίας QEMU, μέσω ενός μηχανισμού μετάφρασης. Όταν λαμβάνεται ένα αίτημα για εκτέλεση, το αποκωδικοποιούμε, ανακτούμε τα απαραίτητα ορίσματα και εκκινούμε την εκτέλεση στην GPU (5d).

Χειριζόμαστε τελικά τα αιτήματα για εκτέλεση στο backend εκτελώντας κατάλληλες CUDA Driver API συναρτήσεις. Επιλέγουμε την προσέγγιση της υλοποίησης του Runtime API χρησιμοποιώντας συναρτήσεις του Driver API καθώς έτσι μας επιτρέπεται ρητή διαχείριση των δομών CUDA. Η ρητή διαχείριση της φόρτωσης των modules και της εναλλαγής των contexts είναι απαιτούμενη προκειμένου να υλοποιηθεί ο διαμοιρασμός των πόρων της GPU. Διασφαλίζουμε απομόνωση και προστασία μεταξύ διεργασιών CUDA θέτοντας το context που συσχετίζεται με την τρέχουσα διεργασία ως το τρέχον, πριν εκκινήσουμε την εκτέλεση μίας συνάρτησης, καθώς κάθε CUDA context διαθέτει τον δικό του αποκλειστικό χώρο διευθύνσεων. Όταν η εκτέλεση ολοκληρωθεί, προσθέτουμε τους buffers που περιέχουν τα απαιτούμενα αποτελέσματα στον μοιραζόμενο δακτύλιο και ειδοποιούμε τον guest προκαλώντας μία εικονική διακοπή (5e).

Προκειμένου να επιτύχουμε διαμοιρασμό της φυσικής συσκευής σε διεργασίες οι οποίες εκτελούνται σε διαφορετικά VMs στο ίδιο host σύστημα, υλοποιούμε ένα ξεχωριστό μοιραζόμενο δακτύλιο μεταξύ της εικονικής συσκευής CUDA και κάθε VM. Αντιμετωπίζουμε κάθε αίτημα εκτέλεσης ξεχωριστά και εφαρμόζουμε πολύπλεξη των αιτήματων από διαφορετικές εικονικές μηχανές, ώστε να επιτρέπουμε διαμοιρασμό

των πόρων της GPU.

3.4 Λεπτομέρειες Υλοποίησης Runtime API

Υλοποιούμε την εικονικοποίηση του Runtime API παραλαμβάνοντας τα ορίσματα των συναρτήσεων βιβλιοθήκης και προωθώντας τα στο backend για εκτέλεση. Η υλοποίηση των τυπικών συναρτήσεων βιβλιοθήκης είναι αρκετά απλή καθώς υπάρχουν αντίστοιχες συναρτήσεις του Driver API οι οποίες προσφέρουν την ίδια λειτουργικότητα. Ωστόσο, η υλοποίηση των συναρτήσεων οι οποίες αντικαθιστούν την επέκταση εκτέλεσης kernel (`<<<...>>>`) απαιτεί περισσότερη προσπάθεια, καθώς οι συναρτήσεις αυτές δεν εκτίθενται στον προγραμματιστή και η υλοποίηση τους δεν είναι ανοιχτού κώδικα. Χρησιμοποιήσαμε ως εκ τούτου τεχνικές reverse engineering προκειμένου να επιτύχουμε εικονικοποίηση της επέκτασης αυτής. Πραγματοποιήσαμε tracing κλήσεων βιβλιοθήκης προκειμένου να καταγράψουμε τη δήλωση των εσωτερικών συναρτήσεων που υλοποιούν την επέκταση. Υλοποιούμε τις συναρτήσεις αυτές εκθέτοντας την ίδια διεπαφή έτσι ώστε να μπορούμε να παραλάβουμε τα ορίσματα εκτέλεσης. Η ρύθμιση και εκτέλεση των kernels γίνεται μέσω πολλαπλών συναρτήσεων βιβλιοθήκης. Στην υλοποίηση μας συγκεντρώνουμε τα κατάλληλα ορίσματα, τα αποθηκεύουμε σε δομές δεδομένων, και τα προωθούμε όταν είναι απαραίτητο κατά την τελευταία χρονικά κλήση.

Επιπλέον, πριν την εκτέλεση ενός kernel χρειάζεται να γίνει φόρτωση του εκτελέσιμου κώδικα στην GPU, από τα κατάλληλα αρχεία αντικειμένων CUDA. Ωστόσο, οι συναρτήσεις του Driver API, οι οποίες υλοποιούν την φόρτωση των modules, απαιτούν από τον προγραμματιστή να παρέχει το όνομα του αρχείου. Το Runtime API διαχειρίζεται αυτόματα την φόρτωση των modules, χωρίς να εκθέτει τα αντίστοιχα ονόματα αρχείων. Έχουμε αναπτύξει επομένως ένα μηχανισμό οποίος, κατά την εκκίνηση της εκτέλεσης μίας εφαρμογής CUDA, αναζητεί τα αρχεία τύπου `.cubin` και χρησιμοποιεί εξαγωγή συμβόλων για να καθορίσει ποιοι kernels ορίζονται σε κάθε αρχείο αντικειμένων. Αποθηκεύουμε την αντιστοίχιση μεταξύ των αρχείων αντικειμένων και των kernels σε μία δομή δεδομένων στη βιβλιοθήκη και αναζητούμε για το αντίστοιχο αρχείο προς φόρτωση κάθε φορά που εκτελείται ένας kernel.

3.5 Απομόνωση και Ασφάλεια

Η υλοποίηση μας εξασφαλίζει την προστασία μεταξύ διεργασιών οι οποίες εκτελούνται μία εικονική μηχανή καθώς και μεταξύ διαφορετικών εικονικών μηχανών. Επιτυγχάνουμε απομόνωση χρησιμοποιώντας τον μηχανισμό των CUDA contexts. Τα CUDA contexts είναι το ισοδύναμο των διεργασιών στη CPU. Κάθε context έχει τον δικό του χώρο διευθύνσεων και, ως αποτέλεσμα, ίδιες τιμές δεικτών σε μνήμη της GPU από διαφορετικά contexts αναφέρονται σε διαφορετικές θέσεις στη φυσική μνήμη. Συσχετίζουμε ένα context με κάθε διεργασία έτσι ώστε να διασφαλιστεί η απομόνωση και προστασία από άλλες διεργασίες που εκτελούνται στο ίδιο καθώς και σε διαφορετικά VMs. Στη συνέχεια θέτουμε το τρέχον context στο backend σύμφωνα με στην καλούσα διεργασία.

Ο μηχανισμός μας έχει σχεδιαστεί με τέτοιο τρόπο ώστε να μπορεί να εμπλουτιστεί με πιο προηγμένα χαρακτηριστικά ως μελλοντικές επεκτάσεις. Για παράδειγμα, ένας μηχανισμός δρομολόγησης θα μπορούσε να προστεθεί ώστε να εξασφαλιστεί η δικαιοσύνη μεταξύ των διαφορετικών εικονικών μηχανών. Προς το παρόν, στην πρωτότυπη υλοποίηση μας πραγματοποιείται πολύπλεξη των αιτημάτων στην μεριά του host κατά FIFO σειρά. Αυτό μπορεί να επεκταθεί σε μελλοντική επέκταση με την εισαγωγή αλγορίθμων χρονοδρομολόγησης οι οποίοι θα εφαρμόζουν δικαιοσύνη και θα προστατεύουν τις εικονικές μηχανές από λάθος ρυθμισμένες ή κακόβουλες εικονικές μηχανές οι οποίες επιχειρούν να καταχραστούν πόρους της GPU.

3.6 Λειτουργικοί Περιορισμοί

Παρά το γεγονός ότι το σύστημα μας παρέχει διάφανη πρόσβαση σε συσκευές CUDA, εισάγει δύο λειτουργικούς περιορισμούς. Εξαιτίας των εσωτερικών συναρτήσεων βιβλιοθήκης, επί του παρόντος υποστηρίζεται μόνο το CUDA Toolkit 5.0 στο guest περιβάλλον. Στο backend, όπου πραγματοποιείται η τελική εκτέλεση, μπορεί να χρησιμοποιηθεί το CUDA toolkit 7.5. Επιπλέον, τα αρχεία αντικειμένων CUDA τα οποία χρησιμοποιούνται για τη φόρτωση εκτελέσιμου κώδικα, πρέπει να είναι διαθέσιμα στον host κατά τον χρόνο εκτέλεσης. Μπορούν είτε να αντιγραφούν πριν την εκτέλεση είτε να γίνουν προσβάσιμα μέσω κοινού συστήματος αρχείων.

Κεφάλαιο 4

Πειραματική Αξιολόγηση

Η αξιολόγηση επίδοσης διεξάγεται σε ένα σύστημα αποτελούμενο από δύο επεξεργαστές Intel Xeon X5650 (@2.66 GHz) με 48 GB κύριας μνήμης, το οποίο είναι εξοπλισμένο με μία μονάδα επεξεργασία γραφικών Nvidia Tesla M2050. Το host σύστημα χρησιμοποιεί λειτουργικό σύστημα Ubuntu Linux 14.04 με έκδοση πυρήνα 3.19.0 και Nvidia driver έκδοσης 352.39. Το λογισμικό εικονικοποίησης που χρησιμοποιείται είναι το QEMU-KVM 2.3. Όλες οι εικονικές μηχανές έχουν ρυθμιστεί να χρησιμοποιούν μία εικονική CPU και 1 GB RAM. Το guest λειτουργικό σύστημα είναι Debian 3.16.7 με έκδοση πυρήνα 3.16.0.

Αρχικά χρησιμοποιούμε συνθετικά benchmarks για να συγκρίνουμε τις δύο υλοποιήσεις τις οποίες χρησιμοποιεί το frontend μέρος του συστήματος για την αναμονή αποτελεσμάτων από το backend. Επιπλέον πραγματοποιούμε ανάλυση breakdown και εξετάζουμε την επιβάρυνση στην επίδοση που εισάγει η στοίβα λογισμικού του συστήματος. Στη συνέχεια, αξιολογούμε την επίδοση μίας πραγματικής εφαρμογής η οποία χρησιμοποιεί τον μηχανισμό μας συγκρίνοντας με την εκτέλεση στο πραγματικό σύστημα. Τέλος αξιολογούμε την κλιμακωσιμότητα του μηχανισμού καθώς ο αριθμός ταυτόχρονα εκτελούμενων εφαρμογών σε VMs στον ίδιο host αυξάνεται.

Προκειμένου να αξιολογήσουμε την επίδοση του πρωτότυπου που έχουμε υλοποιήσει , χρησιμοποιούμε benchmarks από το CUDA SDK 7.5 [13] καθώς και από την συλλογή benchmark Rodinia [14]. Επιλέγουμε benchmarks που αντιπροσωπεύουν ένα ευρύ φάσμα εφαρμογών GPGPU, και χρησιμοποιούμε ποικίλα υπολογιστικά φορτία, μεγέθη δεδομένων και διαφορετικά χαρακτηριστικά του CUDA.

4.1 Υλοποιήσεις Sleep και Busy Wait

Αρχικά εκτελούμε μία αξιολόγηση των δύο προαναφερθέντων μηχανισμών οι οποίοι χρησιμοποιούνται από τον frontend driver για την αναμονή των αποτελεσμάτων εκτέλεσης. Χρησιμοποιούμε ένα microbenchmark που εκτελεί πολλαπλασιασμό πινάκων το οποίο μας επιτρέπει ρυθμίζουμε το μέγεθος των δεδομένων εισόδου. Ο πολλαπλασιασμός πινάκων είναι μία σημαντική λειτουργία που χρησιμοποιείται σε μία ποικιλία εφαρμογών, όπως χρηματοοικονομικές εφαρμογές και εφαρμογές επεξεργασίας σήματος. Συγκρίνουμε τον συνολικό χρόνο εκτέλεσης καθώς και τη χρησιμοποίηση της CPU για κάθε υλοποίηση. Αξιολογούμε τις δύο μετρικές για ένα εύρος μεγέθους δεδομένων εισόδου. Τα αποτελέσματα των πειραμάτων απεικονίζονται στον Πίνακα 4.1.

Τα αποτελέσματα δείχνουν ότι για μικρά μεγέθη δεδομένων εισόδου η υλοποίηση busy wait αποδίδει αρκετά καλύτερα στην μετρική του συνολικού χρόνου εκτέλεσης από την υλοποίηση sleep. Αυτό είναι αναμενόμενο, δεδομένου του ότι η επιβάρυνση από την ενεργοποίηση μίας εικονικής διακοπής είναι υψηλότερη σε σύγκριση με τον επαναληπτικό έλεγχο μέσα σε βρόγχο. Ωστόσο, καθώς το μέγεθος των δεδομένων εισόδου αυξάνει η διαφορά στην επίδοση γίνεται αμελητέα. Επιπλέον, όταν χρησιμοποιείται η μέθοδος busy wait, οι εφαρμογές χρησιμοποιούν πλήρως την CPU σε όλη τη διάρκεια της εκτέλεσης τους, δημιουργώντας περιττό φορτίο στο σύστημα. Στην περίπτωση της μεθόδου sleep, οι εφαρμογές έχουν χαμηλότερη χρησιμοποίηση της CPU, καθώς την απελευθερώνουν κατά τη διάρκεια του χρόνου αναμονής.

Προκειμένου να επωφεληθούμε από τα πλεονεκτήματα και των δύο μεθόδων, υλοποιούμε με υβριδική προσέγγιση. Για μικρά μεγέθη δεδομένων εισόδου χρησιμοποιούμε την υλοποίηση busy wait προκειμένου να επιτύχουμε καλύτερη επίδοση. Στην περίπτωση αυτή, η CPU χρησιμοποιείται πλήρως για ένα σύντομο χρονικό διάστημα, καθώς η εκτέλεση στο backend διαρκεί λίγο για μικρά μεγέθη δεδομένων εισόδου. Αντίθετα, για μεγαλύτερα μεγέθη δεδομένων εισόδου χρησιμοποιούμε την υλοποίηση sleep προκειμένου να επιτύχουμε τόσο χαμηλή χρησιμοποίηση της CPU όσο και καλή επίδοση.

Input Size (KB)		8	16	32	64	128	256	512	1024	2048
Busy Wait	Execution Time (ms)	5.0	8.5	8.7	9.0	14.1	34.9	64.1	240.8	463.0
	CPU Usage (%)	100	100	100	100	100	100	100	100	100
Sleep	Execution Time (ms)	13.8	14.2	14.8	15.0	15.3	35.2	64.1	241.1	462.7
	CPU Usage (%)	10	12	10	10	9	9	10	6	7

Table 4.1: Σύγκριση Υλοποιήσεων Sleep και Busy Wait

4.2 Επίδοση Microbenchmark

Διεξάγουμε μετρήσεις χρησιμοποιώντας διάφορα benchmarks συγκρίνοντας την εκτέλεση στο πραγματικό σύστημα με εκείνη σε εικονικές μηχανές με χρήση του μηχανισμού μας. Πραγματοποιούμε μέτρηση του χρόνου εκτέλεσης όλων των λειτουργιών CUDA χωρίς να συμπεριλαμβάνουμε τους υπολογισμούς που εκτελούνται στην CPU, καθώς ο μηχανισμός εισάγει επιβάρυνση μόνο στις λειτουργίες CUDA. Στο σχήμα παρουσιάζονται οι κανονικοποιημένοι χρόνοι εκτέλεσης σε πραγματικό και εικονικό περιβάλλον. Τα πειραματικά αποτελέσματα δείχνουν ότι η υποβάθμιση της επίδοσης στα benchmark BlackScholes (BS), LU Decomposition (LUD) and Back Propagation (BB) (όπου εκτελούμε τον kernel σε 1000 επαναλήψεις) όταν εκτελούνται σε VM είναι αμελητέα σε σύγκριση με την εκτέλεση στο πραγματικό σύστημα. Ο χρόνος εκτέλεσης τους είναι 1.74%, 3.32% και 1.56% υψηλότερος από εκείνον στο host περιβάλλον αντίστοιχα. Η μεγαλύτερη επιβάρυνση στον χρόνο εκτέλεσης είναι 9.23% για το benchmark fastWalshTransform (FWT). Το benchmark αυτό χαρακτηρίζεται από σύντομο χρόνο εκτέλεσης του kernel (μόλις το ένα τρίτο του συνολικού χρόνου εκτέλεσης) με αποτέλεσμα η επιβάρυνση λόγω της αντιγραφής δεδομένων μεταξύ CPU και συσκευής να έχει μεγαλύτερο αντίκτυπο στην επίδοση του, λόγω της προαναφερθείσας αντιγραφής μεταξύ χώρου χρήστη και χώρου πυρήνα. Η επιβάρυνση αυτή μπορεί να μειωθεί εφαρμόζοντας τεχνικές μηδενικής αντιγραφής όπως memory pinning, οι οποίες μπορούν να υλοποιηθούν ως μελλοντικές επεκτάσεις. Επιπλέον, παρατηρούμε βελτίωση κατά 4.40% στον χρόνο εκτέλεσης των λειτουργιών CUDA στο benchmark πολλαπλασιασμού πινάκων (MM). Η βελτίωση αυτή μπορεί να αποδοθεί στην μετατροπή του Runtime API σε Driver API στο backend.

4.3 Ανάλυση Breakdown

Αναλύουμε την επιβάρυνση που εισάγεται από τον μηχανισμό εικονικοποίησης και πραγματοποιούμε ανάλυση breakdown επιμέρους συναρτήσεων του CUDA Runtime API χρησιμοποιώντας το benchmark BandwidthTest που παρέχεται από τα CUDA samples. Τα αποτελέσματα παρουσιάζονται στο σχήμα 4.2. Επιλέγουμε να πραγματοποιήσουμε μετρήσεις στη συνάρτηση `cudaMemcpy`, καθώς εισάγει την υψηλότερη επιβάρυνση εξαιτίας των λειτουργιών αντιγραφής δεδομένων. Χωρίζουμε την εκτέλεση μίας συνάρτησης σε πέντε στάδια (`lib`, `copy`, `fend`, `exec`, `bend`) τα οποία αντιπροσωπεύουν διαφορετικά μέρη της στοίβας λογισμικού του μηχανισμού καθώς και λειτουργίες που προκαλούν σημαντική επιβάρυνση. Το πρώτο στάδιο (`lib`) περιλαμβάνει τις λειτουργίες που πραγματοποιούνται από την βιβλιοθήκη εκτός από την εκτέλεση της κλήσης συστήματος `ioctl()` η οποία μεταφέρει τον έλεγχο στον frontend driver. Το στάδιο `copy` αντιπροσωπεύει την επιβάρυνση που εισάγεται από

την αντιγραφή δεδομένων μεταξύ χώρου χρήστη και χώρου πυρήνα, ενώ το στάδιο **fend** είναι ο χρόνος που καταναλώνεται στις υπόλοιπες λειτουργίες του frontend. Τέλος, το στάδιο **exec** είναι το χρονικό διάστημα εκτέλεσης της συνάρτησης Driver API και το στάδιο **bend** το χρονικό διάστημα που καταναλώνεται στις υπόλοιπες λειτουργίες που εκτελούνται από το backend, όπως δεσμεύσεις μνήμης, ανάκτηση ορισμάτων κλπ.

Από τα αποτελέσματα προκύπτει ότι το μεγαλύτερο μέρος του χρόνου εκτέλεσης

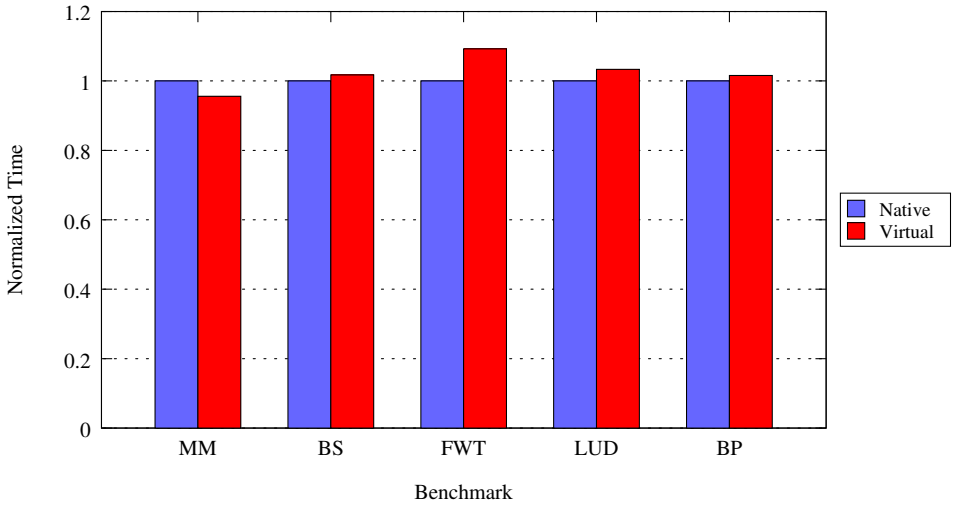

Figure 4.1: Επίδοση Microbenchmark

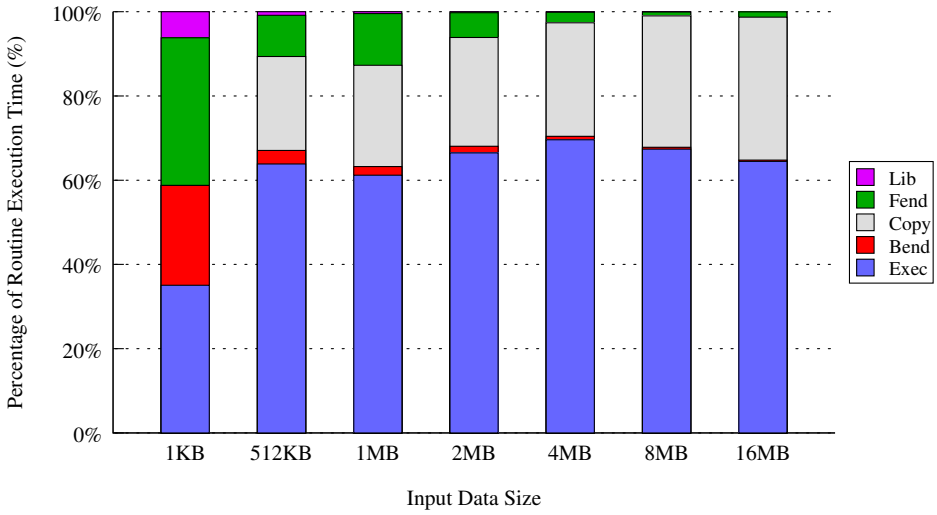

Figure 4.2: Ανάλυση Breakdown

είναι η εκτέλεση της συνάρτησης του Driver API στο backend. Το στάδιο αυτό διαρκεί έως και 69% του συνολικού χρόνου εκτέλεσης για μεγάλες αντιγραφές δεδομένων. Δεδομένου ότι η φάση αυτή αντιπροσωπεύει την τελική εκτέλεση στην GPU, τα υπόλοιπα στάδια εκτέλεσης μπορεί να χαρακτηριστεί ως η επιβάρυνση η οποία προκαλείται από το σύστημα μας. Τα αποτελέσματα δείχνουν ότι η επιβάρυνση αυτή παραμένει σταθερή στο 27% του συνολικού χρόνου εκτέλεσης κατά μέσο όρο, για αντιγραφές δεδομένων μεγαλύτερες του 1 MB. Το γεγονός αυτό ευνοεί εφαρμογές στις οποίες ο χρόνος εκτέλεσης κυριαρχείται από υπολογισμούς στην GPU παρά από αντιγραφές δεδομένων. Το στάδιο της αντιγραφής δεδομένων μεταξύ χώρου χρήστη και χώρου πυρήνα καταναλώνει κατά μέσο όρο το 30% το χρόνου εκτέλεσης και αποτελεί το κύριο μέρος της επιβάρυνσης λόγω εικονικοποίησης. Η εφαρμογή μηχανισμών μηδενικής αντιγραφής, όπως προαναφέρθηκε, μπορεί να μειώσει αυτή την επιβάρυνση. Οι τεχνικές αυτές μπορούν να υλοποιηθούν ως μελλοντικές επεκτάσεις.

Περαιτέρω μετρήσεις δείχνουν ότι οι υπόλοιπες συναρτήσεις της βιβλιοθήκης εισάγουν σταθερή επιβάρυνση. Η λειτουργιά `cudaMalloc` εισάγει επιβάρυνση 24 μs, η λειτουργία `cudaFree` 23 μs και η εκτέλεση ενός πυρήνα στην GPU 64 μs. Η επιβάρυνση στην επίδοση των εφαρμογών μειώνεται καθώς το μέγεθος των δεδομένων εισόδου αυξάνεται, επειδή το η σταθερή επιβάρυνση αποτελεί μικρότερο μέρος του συνολικού χρόνου εκτέλεσης καθώς ο χρόνος εκτέλεσης στην GPU αυξάνεται.

4.4 Επίδοση Εφαρμογής

Αξιολογούμε την αξιοπιστία και την αποδοτικότητα του συστήματος όταν χρησιμοποιείται από μία εφαρμογή υψηλότερου επιπέδου. Χρησιμοποιούμε την εφαρμογή StoreGPU [15] η οποία επιτρέπει σε σχεδιαστές κατανεμημένων συστημάτων αποθήκευσης να αναθέτουν λειτουργίες βασισμένες στον κατακερματισμό σε μονάδες επεξεργασίας γραφικών. Η εφαρμογή StoreGPU επιταχύνει υπολογιστικά και αποθηκευτικά απαιτητικές λειτουργίες, δημοφιλείς σε υλοποιήσεις κατανεμημένων συστημάτων αποθήκευσης. Εκτελούμε τον GPU kernel σε 10 επαναλήψεις και μετράμε τον συνολικό χρόνο εκτέλεσης καθώς και τον χρόνο εκτέλεσης των λειτουργιών CUDA. Το σχήμα 4.3 απεικονίζει το πειραματικά αποτελέσματα τόσο σε πραγματικό περιβάλλον όσο και σε εικονικό με τη χρήση του συστήματος μας. Όπως και σε προηγούμενα πειράματα, το `Cuda` αντιπροσωπεύει μόνο τον χρόνο εκτέλεσης των λειτουργιών CUDA, ενώ το `Total` τον συνολικό χρόνο εκτέλεσης συμπεριλαμβανομένου τόσο του χρόνου εκτέλεσης στην CPU όσο την GPU. Τα αποτελέσματα δείχνουν ότι ο συνολικός χρόνος εκτέλεσης σε μία εικονική μηχανή είναι υψηλότερος κατά 7.67% από εκείνον στο πραγματικό σύστημα ενώ ο χρόνος εκτέλεσης CUDA υψηλότερος κατά 4.67%.

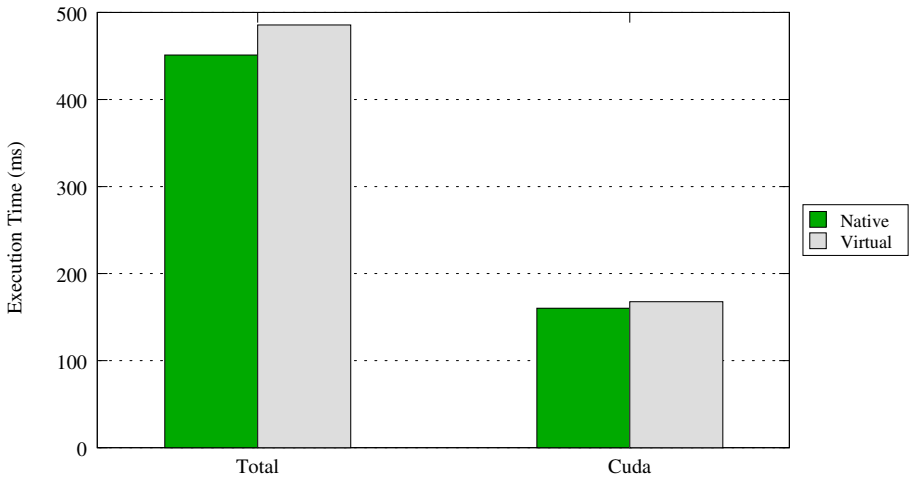

Figure 4.3: Επίδοση Εφαρμογής

4.5 Μετρήσεις Κλιμάκωσης

Αξιολογούμε την επιβάρυνση που εισάγει το σύστημα μας στην ταυτόχρονη εκτέλεση εφαρμογών που χρησιμοποιούν την GPU διεξάγντας δύο είδη μετρήσεων: (1) δημιουργούμε πολλαπλές διεργασίες στο πραγματικό σύστημα οι οποίες πραγματοποιούν εκτέλεση της ίδιας εφαρμογής (`native`) και (2) εκτελούμε την εφαρμογή σε πολλαπλά VMs (`virtual`). Μετράμε τον χρόνο εκτέλεσης των λειτουργιών CUDA για αυτές τις περιπτώσεις και αξιολογούμε την επιβάρυνση που εισάγεται από το σύστημα, καθώς ο αριθμός των διεργασιών και των εικονικών μηχανών αντίστοιχα αυξάνεται. Χρησιμοποιούμε το benchmark BlackScholes το οποίο παρέχεται από τα CUDA samples. Τα αποτελέσματα παρουσιάζονται στο σχήμα 4.4.

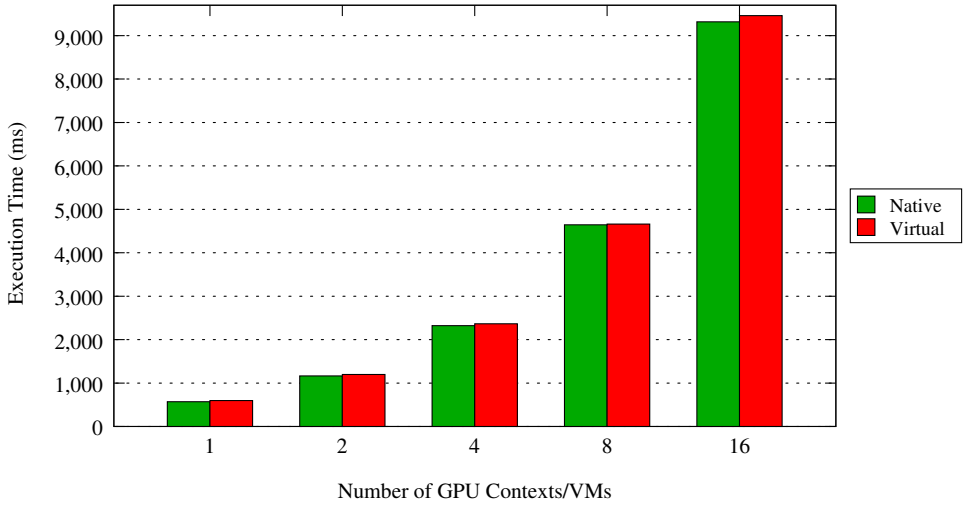

Figure 4.4: Μετρήσεις Κλιμάκωσης

Κεφάλαιο 5

Σχετικές Υλοποιήσεις

Διάφορες προσεγγίσεις για την υλοποίηση εικονικοποίησης συσκευών επεξεργασίας γραφικών και διαμοιρασμού πόρων GPU έχουν προταθεί από την ερευνητική κοινότητα. Παράλληλα, οι πάροχοι cloud υπηρεσιών όπως οι Amazon [8] και Microsoft Azure [34] κλπ. [35] προσφέρουν πόρους GPU στο πλαίσιο των υπηρεσιών τους. Μια κατηγορία προτεινόμενων λύσεων κάνει χρήση της τεχνολογίας pass-through προκειμένου να παρέχει σε εικονικές μηχανές άμεση πρόσβαση στις φυσικές συσκευές. Αυτή η προσέγγιση μπορεί να ελαχιστοποιήσει την επιβάρυνση λόγω εικονικοποίησης, όπως έχουν δείξει μετρήσεις επιδόσεων στο λογισμικό Xen [29], αλλά μέσω της τεχνολογίας pass-through η διαχείριση της GPU γίνεται αποκλειστικά από το guest λειτουργικό σύστημα, και δεν είναι δυνατό πολλαπλά VMs να διαμοιράζονται την ίδια συσκευή. Η τεχνολογία Nvidia GRID [30] επιτρέπει εκχώρηση της φυσικής GPU σε πολλαπλά VMs ταυτόχρονα. Το σύστημα Gdev [28] επιτρέπει την εικονικοποιήση μίας φυσικής GPU σε πολλαπλές λογικές, οι οποίες μπορούν στη συνέχεια να ανατεθούν σε VMs, επιτρέποντας με αυτό τον τρόπο τον διαμοιρασμό πόρων.

Το gVirt [26] αποτελεί μία προσέγγιση πλήρους εικονικοποίησης η οποία επιτρέπει στον driver της GPU να εκτελείται στο guest λειτουργικό σύστημα. Ο μηχανισμός αυτός έχει υλοποιηθεί σε επεξεργαστές γραφικών Intel Processor Graphics και η αξιολόγηση του είναι προσανατολισμένη σε εφαρμογές 2D και 3D γραφικών. Το GPUvm είναι ένας μηχανισμός ο οποίος βασίζεται στο λογισμικό Xen και υλοποιεί πλήρη εικονικοποίηση καθώς και παρα-εικονικοποίηση μονάδων επεξεργασίας γραφικών. Η προσέγγιση αυτή διαφέρει από τη δική μας, καθώς υλοποιεί εικονικοποίηση της GPU σε χαμηλότερο επίπεδο και χρησιμοποιεί την πλατφόρμα Gdev [24] για την υποστήριξη του CUDA Runtime.

Τα συστήματα vCUDA [17] και rCuda [18, 19] εφαρμόζουν την τεχνική της

ανακατεύθυνσης κλήσεων API προκειμένου να καταστήσουν δυνατή τη διαφανή πρόσβαση σε συσκευές Nvidia σε εφαρμογές που εκτελούνται σε VMs. Και τα δύο χρησιμοποιούν δικτυακά πρωτόκολλα προκειμένου να υλοποιήσουν μηχανισμούς επικοινωνίας. Παρόλο που οι δικτυακοί μηχανισμοί επικοινωνίας καθιστούν αυτές τις υλοποιήσεις ανεξάρτητες του συστήματος εικονικοποίησης, προκαλούν σημαντική επιβάρυνση στην επίδοση. rCUDA διαθέτει επίσης μία βελτιστοποιημένη υλοποίηση του μηχανισμού επικοινωνίας για διασυνδέσεις InfiniBand, προκειμένου να επωφεληθεί από την υψηλή ταχύτητα του μέσου. Επιπλέον έχει προταθεί το gVirtus [21], ένας μηχανισμός εικονικοποίησης GPU ο οποίος δίνει έμφαση στην ανεξαρτησία από τον hypervisor και βασίζεται σε έναν μηχανισμό επικοινωνίας ανεξάρτητο από τον δίαυλο μέσω του οποίου υλοποιείται η επικοινωνία. Σύγκριση με τις μετρήσεις που παρουσιάζονται στο [21] υποδεικνύει ότι το σύστημα μας εισάγει χαμηλότερη επιβάρυνση από αυτή την προσέγγιση. Αυτό μπορεί να αποδοθεί στην εκτέλεση συναρτήσεων του Runtime API στο backend. Το GViM [22] χρησιμοποιεί την προσέγγιση της παρα-εικονικοποίησης προκειμένου να υλοποιήσει την εικονικοποίηση και διαχείριση των πόρων της GPU. Χρησιμοποιεί μηχανισμούς του Xen, συμπεριλαμβανομένων και μοιραζόμενων buffers, με σκοπό την υλοποίηση του μηχανισμού επικοινωνίας. Η υλοποίηση αυτή εισάγει μεγαλύτερη επιβάρυνση από τον μηχανισμό μας, όπως δείχνει η σύγκριση μεταξύ των μετρήσεων στο [22] και αυτής της εργασίας. Ο μηχανισμός LoGV [23] υλοποιεί εικονικοποίηση λειτουργιών GPGPU σε χαμηλότερο επίπεδο χρησιμοποιώντας τους μηχανισμούς προστασίας των σύγχρονων GPUs. Εικονικοποιεί την διεπαφή του `pscnv` GPU driver και χρησιμοποιεί λογισμικό ανοικτού κώδικα προκειμένου να προσφέρει υποστήριξη του CUDA API. Επιπλέον, η προσέγγιση αυτή δεν διασφαλίζει την προστασία μεταξύ εφαρμογών που εκτελούνται στην ίδια εικονική μηχανή, παρά μόνο μεταξύ εικονικών μηχανών. Στην υλοποίηση DS-CUDA [25] οι συγγραφείς παρουσιάζουν έναν μηχανισμό με στόχο την αντιμετώπιση των δυσκολιών του προγραμματισμού σε ετερογενή κατανεμημένα συστήματα. Ο μηχανισμός υλοποιεί εικονικοποίηση ενός cluster υπολογιστών εξοπλισμένων με GPUs έτσι ώστε να εμφανίζεται ότι οι GPUs είναι συνδεδεμένες σε έναν μόνο κόμβο, με σκοπό την διευκόλυνση του προγραμματισμού εφαρμογών που εκτελούνται σε πολλαπλές GPUs. Στο πλαίσιο του διαμοιρασμού πόρων της GPU μεταξύ εικονικών μηχανών έχει προταθεί το V4VSockets [20], ένας μηχανισμός που προσφέρει αποδοτική επικοινωνία μεταξύ κόμβων, υλοποιημένος στο λογισμικό Xen. Οι συγγραφείς δείχνουν ότι το rCUDA μπορεί να εκτελεστεί με χρήση του V4Vsockets επιτρέποντας τον αποδοτικό διαμοιρασμό πόρων της GPU μεταξύ VMs.

Κεφάλαιο 6

Σύνοψη

6.1 Ανάλυση Επίδρασης του Διαμοιρασμού Πόρων στην Επίδοση Εφαρμογών

Ο διαμοιρασμός του υλικού μεταξύ πολλαπλών ταυτόχρονα εκτελούμενων λειτουργικών συστημάτων αποτελεί μία θεμελιώδη πτυχή της εικονικοποίησης υλικού. Ο μηχανισμός μας επιτρέπει τον διαμοιρασμό πόρων της GPU εφαρμόζοντας πολύπλεξη των αιτημάτων εκτέλεσης στο επίπεδο του hypervisor. Ωστόσο, η πολύπλεξη της πρόσβασης εφαρμογών στην GPU εισάγει επιπρόσθετη επιβάρυνση και επιδρά αρνητικά στην επίδοση τους. Επιπλέον, εφαρμογές με διαφορετικά μοτίβα εκτέλεσης μπορεί να επηρεάζονται διαφορετικά από τον διαμοιρασμό των προσβάσεων τους στην GPU.

Οι εφαρμογές που χρησιμοποιούν επιτάχυνση από την GPU μπορούν διαχωριστούν ανάλογα με τα χαρακτηριστικά τους σε διάφορες κατηγορίες. Μία από αυτές περιλαμβάνει εφαρμογές οι οποίες μπορούν να χαρακτηριστούν ως εφαρμογές μαζικής επεξεργασίας. Οι εφαρμογές αυτές αντιγράφουν μεγάλες ποσότητες δεδομένων από τον επεξεργαστή στην GPU και στη συνέχεια πραγματοποιούν εντατικούς υπολογισμούς χωρίς περαιτέρω αλληλεπίδραση με τον χρήστη. Τα αποτελέσματα υπολογίζονται και αντιγράφονται στη μνήμη της CPU κατά την ολοκλήρωση της εκτέλεσης. Παραδείγματα τέτοιων εφαρμογών αποτελούν HPC επιστημονικές εφαρμογές από πεδία όπως η βιοπληροφορική [31] και η επιστήμη των υλικών [32]. Η πολύπλεξη των προσβάσεων στην GPU εφαρμογών αυτής της κατηγορίας μπορεί να προκαλέσει υποβάθμιση της επίδοσης, καθώς οι παλαιότερες GPUs δεν προσφέρουν πραγματικό διαμοιρασμό πόρων. Η κρίσιμη μετρική επίδοσης είναι ο συνολικός χρόνος εκτέλεσης. Ωστόσο, το κύριο χαρακτηριστικό των εφαρμογών αυτών είναι ότι ο χρόνος εκτέλεσης τους κυριαρχείται από χρησιμοποίηση πόρων της GPU. Ως εκ

τούτου, η πολύπλεξη των προσβάσεων στην GPU ταυτόχρονα εκτελούμενων εφαρμογών επιβαρύνει την επίδοση όλων των εφαρμογών, καθώς αιτήματα για εκτέλεση από διαφορετικά CUDA contexts σειριοποιούνται κατά την εκτέλεση στην GPU. Αυτό είναι ένα εγγενές χαρακτηριστικό της αρχιτεκτονικής των παλαιότερων GPUs. Αντιθέτως, στην περίπτωση που οι εφαρμογές υποβάλλονται για εκτέλεση σειριακά, όπως για παράδειγμα σε ένα σύστημα χρονοπρογραμματισμού πόρων (Torque) η υποβάθμιση της επίδοσης κάθε εφαρμογής είναι αμελητέα. Ωστόσο, ακόμη και αυτή η κατηγορία εφαρμογών αναμένεται να επιτυγχάνει καλύτερη επίδοση στις σύγχρονες GPUs.

Από την άλλη πλευρά, μία διαφορετική κατηγορία περιλαμβάνει διαδραστικές εφαρμογές μακράς εκτέλεσης, οι οποίες συνήθως εκκινούν την εκτέλεση τους αντιγράφοντας τα απαραίτητα δεδομένα στη συσκευή, εκτός του κρίσιμου μονοπατιού εκτέλεσης, και λαμβάνουν διαδοχικά μικρότερες ποσότητες δεδομένων ως είσοδο, οι οποίες εκκινούν υπολογισμούς. Παραδείγματα εφαρμογών οι οποίες ακολουθούν αυτό το μοτίβο εκτέλεσης αποτελούν εφαρμογές Big Data οι οποίες εκτελούν ερωτήματα σε μεγάλα σύνολα δεδομένων [33]. Εφαρμογές αυτής της κατηγορίας έχουν συνήθως απαιτήσεις πραγματικού χρόνου. Ο κρίσιμος παράγοντας της επίδοσης τους είναι η ταχύτητα απόκρισης όταν λαμβάνεται κάποια είσοδος. Η εκτέλεση τους περιλαμβάνει εναλλαγές μεταξύ περιόδων αδράνειας, όπου αναμένεται είσοδος από τον χρήστη, και περιόδων επεξεργασίας, όπου υπολογίζονται τα ζητούμενα αποτελέσματα. Αυτό το μοτίβο εκτέλεσης επιτυγχάνει καλή επίδοση στον διαμοιρασμό της συσκευής μεταξύ ταυτόχρονα εκτελούμενων εφαρμογών. Η πολύπλεξη των προσβάσεων τους στην GPU είναι εφικτή καθώς περίοδοι αδράνειας κάποιων εφαρμογών μπορεί να συμπίπτουν με περιόδους υπολογισμών άλλων.

Προκύπτει επομένως ότι η επίδραση του διαμοιρασμού πόρων της GPU μπορεί να είναι διαφορετική ανάλογα με το μοτίβο εκτέλεσης της εφαρμογής. Ένα σύστημα διαχείρισης της GPU θα μπορούσε να κατατάσσει τις εφαρμογές στις κατηγορίες που περιγράφηκαν παραπάνω και να δρομολογεί τις προσβάσεις τους στους πόρους της GPU ανάλογα. Μελλοντική επέκταση του μηχανισμού μπορεί να περιλαμβάνει την εφαρμογή τεχνικών profiling με σκοπό την κατηγοριοποίηση εφαρμογών καθώς και τη χρήση διάφορων αλγορίθμων χρονοδρομολόγησης των προσβάσεων στην GPU. Οι τεχνικές αυτές μπορούν ακόμη να αξιολογηθούν σε σύγχρονες GPUs οι οποίες παρέχουν πιο προηγμένα χαρακτηριστικά διαμοιρασμού μεταξύ ταυτόχρονων εκτελέσεων.

6.2 Συμπεράσματα και Μελλοντικές Κατευθύνσεις

Στην εργασία αυτή παρουσιάζουμε έναν μηχανισμό ο οποίος πραγματοποιεί αποδοτική εικονικοποίηση πόρων GPU καθώς και διαμοιρασμό τους μεταξύ εικονικών μηχανών. Η υλοποίηση μας εφαρμόζει ανακατεύθυνση API μέσω ενός driver χωρισμένου σε δύο μέρη, προκειμένου να επιτρέπει σε εφαρμογές GPGPU να έχουν πρόσβαση στην φυσική συσκευή. Η αξιολόγηση της πρωτότυπης υλοποίησης μας δείχνει ότι ο μηχανισμός επιτυγχάνει επίδοση πού κοντά σε εκείνη του πραγματικού συστήματος για μεσαία και μεγάλα μεγέθη δεδομένων εισόδου. Επιπλέον, εφαρμογές που εκτελούνται ταυτόχρονα σε διαφορετικά VMs μπορούν να μοιράζονται αποδοτικά την GPU.

Σχεδιάζουμε τον μηχανισμό με τρόπο ο οποίος να επιτρέπει επεκτάσεις που πραγματοποιούν χρονοδρομολόγηση να προστεθούν εύκολα στην παρούσα υλοποίηση. Ως μελλοντική προέκταση, σχεδιάζουμε να υλοποιήσουμε χρονοδρομολόγηση των αιτημάτων εκτέλεσης προκειμένου να επιτυγχάνεται δικαιοσύνη στην εξυπηρέτηση μεταξύ τόσο των εικονικών μηχανών όσο και των εφαρμογών. Μπορεί ακόμη να υλοποιηθεί διαχείριση των πόρων της GPU στο backend έτσι ώστε να διασφαλίζεται δικαιοσύνη, αναγνωρίζοντας τα μοτίβα εκτέλεσης των GPU εφαρμογών και προγραμματίζοντας κατάλληλα την πρόσβαση τους στην GPU. Για το σκοπό αυτό, ο μηχανισμός θα μπορούσε να ανιχνεύει και να επιβραδύνει εικονικές μηχανές με υψηλές απαιτήσεις σε πόρους της GPU. Τέλος, μελλοντικές επεκτάσεις περιλαμβάνουν επίσης την αξιολόγηση του μηχανισμού μας σε σύγχρονες Nvidia GPUs με βελτιωμένα χαρακτηριστικά που αφορούν τον διαμοιρασμό των πόρων.

Bibliography

[1] Abhijeet Gaikwad and Ioane Muni Toke. Gpu based sparse grid technique for solving multidimensional options pricing pdes. In *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 6:1–6:9, New York, NY, USA, 2009. ACM.

[2] D.P. Playne and K.A. Hawick. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09)*, pages 104–110, Las vegas, USA, 13-16 July 2009. WorldComp.

[3] J. Michalakes and M. Vachharajani. Gpu acceleration of numerical weather prediction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7, April 2008.

[4] Everett H. Phillips, Yao Zhang, Roger L. Davis, and John D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, number AIAA 2009-565, jan 2009.

[5] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[6] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and Kyoungsoo Park. Sslshader: cheap ssl acceleration with commodity processors. In *In Proceedings of the 8th USENIX conference on Networked systems and implementation, NSDI'11*. USENIX Association, 2011.

[7] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of*

the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[8] Amazon GPU Instances. http://aws.amazon.com/ec2/instance-types/.

[9] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[10] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[11] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[12] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[13] NVIDIA. NVIDIA CUDA SDK code samples. http://developer.download.nvidia.com/compute/cuda/5_0/rel-update-1/installers/cuda_5.0.35_linux_64_ubuntu11.10-1.run.

[14] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[15] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. Storegpu: Exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM.

[16] Nvidia. Nvidia's Next Generation CUDA Compute Architecture Kepler GK110. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[17] Lin Shi, Hao Chen, and Jianhua Sun. vcuda: Gpu accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, May 2009.

[18] Antonio J. Pena, Carlos Reano, Federico Silla, Rafael Mayo, Enrique S. Quintana-Orti, and Jose Duato. A complete and efficient cuda-sharing solution for {HPC} clusters. *Parallel Computing*, 40(10):574 – 588, 2014.

[19] J. Duato, A.J. Pena, F. Silla, J.C. Fernandez, R. Mayo, and E.S. Quintana-Orti. Enabling cuda acceleration within virtual machines using rcuda. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011.

[20] Anastassios Nanos, Stefanos Gerangelos, Ioanna Alifieraki, and Nectarios Koziris. V4vsockets: Low-overhead intra-node communication in xen. In *Proceedings of the 5th International Workshop on Cloud Data and Platforms*, CloudDP '15, pages 1:1–1:6, New York, NY, USA, 2015. ACM.

[21] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 379–391, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] 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.

[23] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1721–1726, 2013.

[24] S. Kato. Gdev CUDA Runtime. https://github.com/shinpei0208/gdev.

[25] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi. Distributed-shared cuda: Virtualization of large-scale gpu systems for programmability and reliability. In *FUTURE COMPUTING 2012, The Fourth International Conference on Future Computational Technologies and Applications*, page 7–12, 2012.

[26] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, Philadelphia, PA, June 2014. USENIX Association.

[27] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual*

Technical Conference (USENIX ATC 14), pages 109–120, Philadelphia, PA, June 2014. USENIX Association.

[28] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[29] A.J. Younge, J.P. Walters, S. Crago, and G.C. Fox. Evaluating gpu passthrough in xen for high performance cloud computing. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 852–859, May 2014.

[30] Nvidia. NVIDIA GRID Virtual GPU Technology. http://www.nvidia.com/object/grid-technology.html.

[31] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda - a fast short read sequence aligner using graphics processing units. *BMC Research Notes*, 5(1):1–7, 2012.

[32] Stefan Maintz, Bernhard Eck, and Richard Dronskowski. Speeding up plane-wave electronic-structure calculations using graphics-processing units. *Computer Physics Communications*, 182(7):1421 – 1427, 2011.

[33] GPUdb. GPUdb The first GPU accelerated In-Memory Distributed database. http://www.gpudb.com/.

[34] Microsoft Azure. https://azure.microsoft.com/en-us/.

[35] NVIDIA. GPU Cloud Computing. http://www.nvidia.com/object/gpu-cloud-computing-services.html.