UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática
de Sistemas y Computadores

Improving Performance and Energy Efficiency of Heterogeneous Systems with rCUDA

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Computer Engineering)

Author

Javier Prades Gasulla

Advisor

Prof. Federico Silla Jiménez

January 2021

Doctoral Committee

- Prof. Ignacio Blanquer Espert

  Universitat Politècnica de València, València, Spain

- Dr. Francisco J. Andújar Muñoz

  Universidad de Valladolid, Valladolid, Spain

- Dr. José Cano Reyes

  University of Glasgow, Glasgow, Scotland

Agraïments

Corria l'estiu de l'any 2009, jo recent havia acabat els estudis d'Enginyeria Tècnica en Informàtica de Sistemes a la Universitat Jaume I de Castelló i la crisi del 2008, la de la "bombolla immobiliària", havia deixat en un estat molt precari el teixit empresarial de tota la província, molt vinculat amb el sector tauleller. Amb aquest panorama, trobar feina de qualitat era tota una utopia, i a més a mi encara em rondava pel cap la idea de seguir amb els estudis i aconseguir el títol d'Enginyer en Informàtica.

Després d'unes poques entrevistes en treballs que no m'agradaven massa i que, per sort, no van anar molt bé, un gest totalment altruista de M.ª Angeles em va canviar la vida. M.ª Angeles em va oferir la seua casa a València per al que necessités, i jo, bé que ho vaig aprofitar. A partir d'aquell moment tot va anar rodat, em vaig poder matricular en quart curs d'Enginyeria Informàtica a la Universitat Politècnica de València i poc després vaig aconseguir una feina d'investigador a la mateixa universitat i sota la direcció d'un tal Federico Silla. Aquell primer contracte, de mes i mig, va anar allargant-se renovació rere renovació, saltant de projecte en projecte, tot amb contractes una mica precaris (tot s'ha de dir), però que m'han permés arribar fins on estic avui i complir el somni que és aquesta Tesi Doctoral. Al mateix temps, aquell desconegut Federico Silla va anar convertint-se en Fede, una de les persones més influents de la meua vida. Gràcies a ell vaig descobrir el que era un "paper" i un "deadline" (que encara que espanta molt, sempre s'acaben allargant un parell de setmanes). També vaig descobrir el que era treballar de veritat i de forma incansable, fos el diumenge de Pasqua o les vacances d'estiu. I la lliçó més important, confiar en un mateix, ser valent, no rendir-se i lluitar pel que et pertany i pels teus somnis.

La investigació és una cosa molt bonica que consisteix a agafar una cullereta de café i anar rascant a poc a poc fins a desfer la roca més gran que et pugues imaginar (això també m'ho va ensenyar Fede), el dia que aconsegueixes desfer la roca completament estàs molt content, però la majoria dels dies no són tan bons i molts poden arribar a ser frustrants. D'aquests dies no tan bons sap molt María, María sempre està ahí, per al que siga, sempre de bon humor i a més, al ser una persona totalment optimista (massa

segons el meu punt de vista), aconsegueix que inclús els pitjors dies acaben sent bons per una raó o l'altra.

Este últim any ha sigut molt dur, el coronavirus de Wuhan ens ha limitat molt les relacions personals i aquesta situació m'ha portat a ser plenament conscient de la importància que tenen aquest tipus de relacions en la meua vida. Durant el desenrotllament d'una Tesi Doctoral, o qualsevol gran projecte en la vida, hi ha un gran creixement personal, en el cas de la Tesi Doctoral un pot pensar que aquest creixement es dóna principalment per les vivències experimentades en els viatges a reu del món, per escoltar conferències de grans "gurús" en la matèria, etc. Però en el meu cas, estic segur al 100% que a mi el que més m'ha fet créixer com a persona han sigut totes les persones amb les quals compartisc treball i amb les que he participat, quasi diàriament, de nombrosíssimes xarrades en acabant de dinar. Aquestes xarrades a l'hora del café, molt interessants algunes i altres totalment banals, comprenien temes relacionats amb la informàtica en alguns casos, però principalment estaven relacionades amb temes totalment diferents, com poden ser la política, el futbol, el lliure albir, la física o la figura del "dictador bo" per anomenar alguns.

Finalment, m'agradaria agrair als meus pares tota l'ajuda que m'han donat en la meua formació acadèmica i sobretot la gran educació i esperit de superació que m'han inculcat des de molt xicotet, fent-me anar sempre un pas més enllà d'on jo creia que podia arribar.

Contents

List of Figures **xiii**

List of Tables **xxi**

Abstract **xxiii**

Resumen **xxv**

Resum **xxvii**

1 Introduction **1**
1.1 Background . . . 2
1.1.1 GPGPU . . . 2
1.1.2 Main Concerns of Using GPUs . . . 3
1.1.3 Remote GPU Virtualization . . . 3
1.1.4 rCUDA . . . 4
1.2 Objectives of the Thesis . . . 5
1.3 Main Contributions of the Thesis . . . 6
1.4 Thesis Outline . . . 7
References . . . 9

2 On the Effect of using rCUDA to Provide CUDA Acceleration to Xen Virtual Machines **13**
2.1 Introduction . . . 14
2.2 Providing CUDA GPUs to Virtual Machines . . . 16
2.3 rCUDA: Remote CUDA . . . 24
2.4 Testbeds Used in The Experiments . . . 25
2.5 Network Performance Observed by Xen VMs . . . 29
2.6 Performance of rCUDA within Xen VMs . . . 31
2.7 Impact of Xen VMs on Real Applications . . . 36
2.7.1 Applications Using One GPU . . . 37
2.7.2 Applications Using Multiple GPUs . . . 42
2.8 Conclusions . . . 46
References . . . 47

3 Made-to-Measure GPUs on Virtual Machines with rCUDA **53**
3.1 Introduction . . . 54
3.2 Motivation . . . 56

3.3 Background on GPU Virtualization 60
3.4 Performance Evaluation 62
3.5 Conclusions 67
References 68

4 Multi-tenant virtual GPUs for optimising performance of a financial risk application 71
4.1 Introduction 72
4.2 Related work 74
4.3 rCUDA 77
4.4 Financial risk application 81
4.4.1 Input and Output Data 82
4.4.2 Algorithm and GPU Implementation 84
4.5 Evaluation 86
4.5.1 Platform 87
4.5.2 Application Scalability 87
4.5.3 Reducing Execution Time Using rCUDA 90
4.5.4 Mitigating the Impact of Data Transfers in rCUDA 92
4.5.4.1 Concurrent vs Sequential Data Transfers 93
4.5.4.2 Multi-tenancy Approach 94
4.5.5 Performance Analysis Using Multi-tenancy 97
4.5.6 Modelling Multi-tenancy for Performance and Energy Estimation . 98
4.5.6.1 Performance Model 98
4.5.6.2 Energy Model 102
4.6 Conclusions 105
References 105

5 Maximizing resource usage in Multi-fold Molecular Dynamics with rCUDA 111
5.1 Introduction 112
5.2 Background 114
5.2.1 MD in Drug Discovery 114
5.2.2 rCUDA (remote CUDA) 115
5.3 System Configurations for Drug Discovery 116
5.4 Flavonoids as a Working Example 119
5.5 System Performance and Throughput 120
5.5.1 Test bed: Hardware and Software Environment 121
5.5.2 Performance Characterization 121
5.5.3 Throughput for Each Case Study 125
5.5.4 Overall System Throughput 131
5.5.5 Analysis of obtained MD results in terms of biological validation . 134
5.6 Conclusions and Future Work 134
References 137

6 Turning GPUs into Floating Devices over The Cluster: The Beauty of GPU Migration 141
6.1 Introduction 142

6.2 About Remote GPU Virtualization . . . 145
6.3 Implementing GPU Migration . . . 147
6.4 First results of GPU Migration within rCUDA . . . 149
6.5 Conclusions . . . 158
References . . . 158

7 GPU-Job Migration: the rCUDA Case **163**
7.1 Introduction . . . 164
7.2 About Remote GPU Virtualization . . . 166
7.3 Related Work on GPU Migration . . . 167
7.4 Implementing GPU Migration in rCUDA . . . 169
7.5 Performance Evaluation of GPU Migration with rCUDA . . . 172
7.5.1 Synthetic Application . . . 173
7.5.2 Real Applications . . . 177
7.5.3 Use Cases for GPU-Job Migration with rCUDA . . . 184
7.5.3.1 GPU Server Consolidation . . . 184
7.5.3.2 GPU Load Balancing . . . 187
7.5.3.3 Improved Management of User Priorities . . . 189
7.6 Conclusions . . . 190
References . . . 191

8 Conclusions **195**
8.1 Contributions . . . 196
8.2 Future Work . . . 198
8.2.1 GPU-Job Scheduler . . . 198
8.2.2 Quantification of the Economic Impact of Applying the Mechanisms Developed in this Thesis . . . 199
8.3 Publications . . . 200
8.3.1 Main Publications . . . 200
8.3.2 Main Collaborations . . . 202
References . . . 205

List of Figures

1.1 Architecture of the rCUDA middleware. 4

2.1 Typical architecture used by GPU virtualization solutions. 17

2.2 Performance comparison among three different GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. 20

2.3 Typical configuration of a Xen-based system showing how the Ethernet adapter and the GPU available in the host are provided to VMs. 25

2.4 Testbeds used in the experiments presented in this paper, which make use of rCUDA to provide GPU access to VMs. 26

2.5 Bandwidth attained by the virtual network among Xen VMs. 29

2.6 InfiniBand bandwidth tests using ConnectX-3 network cards executed in the different scenarios under study. 31

2.7 Bandwidth tests for copies between host and device memory, using CUDA and the rCUDA middleware. 32

2.8 Performance of a synthetic application where the percentage of execution time devoted to data transfers to/from the GPU and the percentage of execution time used for computations in the GPU are set by the user. . . 35

2.9 Execution time of several applications when executed in different local and remote scenarios. 38

2.10 Average overhead with respect to executions with CUDA in a native domain for the four applications depicted in Figure 2.9. 39

2.11 Histograms showing the percentage of transferred data according to message size. 41

2.12 Average overhead experienced by applications with respect to executions with CUDA using the PCI passthrough from the inside of a VM. 42

2.13 Configuration of a Xen-based system showing two GPUs assigned to one of the VMs. 43

2.14 Testbeds used with rCUDA. 44

2.15 Performance of two applications when executed in different local and remote scenarios involving Xen VMs. 45

3.1 Assignment of GPUs to VMs when using the PCI passthrough technique, which causes that GPUs are assigned to VMs in an exclusive way. 55

3.2 Assignment of GPUs to VMs when using the remote GPU virtualization mechanism, which allows GPUs to be concurrently shared among VMs. . 56

3.3 Evolution of GPU utilization and memory occupancy during the execution of the four applications considered in this study. 58

3.4 Evolution of GPU utilization and memory occupancy during the execution of a sequence of instances of the four applications considered in this paper for one hour time interval. 60

3.5 Architecture of the rCUDA middleware. 61

3.6 Execution time of the four applications under consideration in this study. 62

3.7 Test beds used in the experiments in this section. 63

3.8 Average execution time for each of the applications considered in this study when executed in the scenarios depicted in Figure 3.7. 64

3.9 Total amount of jobs executed (system throughput) for each of the applications considered in this study when executed in the scenarios depicted in Figure 3.7. 65

3.10 Average overhead depending on system load for the different test beds depicted in Figure 3.7. 65

3.11 Average system throughput depending on system load for the different test beds depicted in Figure 3.7. 66

3.12 Energy consumption for each of the scenarios depicted in Figure 3.7. . . . 66

4.1 Execution time of the financial application on multiple local GPUs. 72

4.2 Distributed acceleration architecture facilitated by rCUDA. 76

4.3 rCUDA client and server software/hardware stack. 77

4.4 Communication sequence between a client and the rCUDA server daemon. 79

4.5 Comparison of bandwidth for pinned memory and pageable memory of rCUDA, DS-CUDA and gVirtuS using CUDA as a baseline reference (DS-CUDA does not support pinned memory). 80

4.6 Computation and data transfer times for the financial risk application when executed on single and multiple GPUs with CUDA. 88

4.7 Amount of data transferred during the execution of the financial risk application. 89

4.8 Attained bandwidth when concurrent data transfers to GPUs are performed. 90

4.9 Scalability of the financial risk application when executed with rCUDA. . 91

4.10 Bandwidth attained for multiple data transfers concurrently to different remote GPUs using rCUDA. 91

4.11 Communication approaches for transferring data to GPUs. 92

4.12 GPU utilisation, power and energy consumption of concurrent and sequential data transfers to GPUs considered in Figure 4.11. 94

4.13 Sequential data copies with several vGPUs per GPU. 94

4.14 GPU utilisation, power and energy consumption of the multi-tenancy approach considered in Figure 4.13. 96

4.15 Application performance for different combinations of pGPUs and vGPUs using QDR InfiniBand. 97

4.16 Application performance for different combinations of pGPUs and vGPUs using FDR InfiniBand. 98

4.17 Results from performance model for QDR InfiniBand. 101

4.18 Results from performance model for FDR InfiniBand. 102

4.19 Results from energy model for QDR InfiniBand. 103

4.20 Results from energy model for FDR InfiniBand. 104

4.21 Combined space of energy and execution time using QDR InfiniBand. . . 104

4.22 Combined space of energy and execution time using FDR InfiniBand. . . 105

5.1 Architecture of the rCUDA middleware. 115

5.2 Hardware configurations for each of the baseline case studies considered in this paper. 118

5.3 Performance of the MD simulations when 3, 5, 10 and 20 threads are leveraged. 121

5.4 Energy per simulated ns required by the MD simulations when 3, 5, 10 and 20 threads are leveraged. 123

5.5 Average power required by the GPU and by the rest of the system in the CUDA scenario. 124

5.6 Throughput of the CPU-only MD simulations when several instances are concurrently executed in the same node. 125

5.7 Energy per simulated ns required by GROMACS when several CPU-only instances are concurrently executed in the same node. 126

5.8 GPU memory and GPU utilization along the execution time of the GROMACS simulator configured to use 10 threads with the molecules under study. 127

5.9 Instant power and accumulated energy along the execution time of the GROMACS simulator configured to use 10 threads with the molecules under study. 128

5.10 Throughput and GPU utilization when several instance of GROMACS share the GPU in the rCUDA server by leveraging the rCUDA middleware. 129

5.11 Energy per simulated ns required by GROMACS when several simulator instances share the GPU in the rCUDA server by leveraging the rCUDA middleware. 130

5.12 Aggregated throughput projection for a hybrid cluster composed of n nodes where half of the nodes own a GPU whereas the other half of the nodes do not leverage any accelerator. 133

5.13 Aggregated throughput projection for a homogeneous cluster composed of n nodes where all the nodes own one GPU. 133

5.14 RMSD over time for the DNA structure. 135

5.15 Average DNA(center of mass) to DEPHBC distance over time. 135

5.16 Superposition of first and last frame of the DNA-DEPHBC MD simulation. 135

6.1 Comparison, from a logical point of view, of two cluster configurations: (a) remote GPU virtualization is not leveraged; (b) remote GPU virtualization is used. 143

6.2 Usage of GPU migration in a cluster in order to consolidate GPU jobs and reduce energy. 144

6.3 General organization of remote GPU virtualization frameworks. 146

6.4 Execution time using CUDA and rCUDA without migration. 150

6.5 Overhead introduced in executions of Figure 6.4 because of executing the applications with rCUDA using a remote GPU instead of using a local one with CUDA. 152

6.6 Execution time using CUDA and rCUDA. 153

6.7 Overhead introduced because of carrying out one live migration while executing the applications with rCUDA using a remote GPU. 154

6.8 Estimated average time required to perform one live migration while applications are in execution. 156

6.9 Overhead with respect to CUDA when applications are live migrated up to five times during their execution. 157

7.1 General organization of remote GPU virtualization frameworks. 166

7.2 Migration modules inside rCUDA client and server. 168

7.3 Complete operation of the migration module implemented within the rCUDA middleware. 170

7.4 Bandwidth attained for several network configurations using different transfer sizes. 174

7.5 Time required to migrate a job among two P100 GPUs located in different nodes. 176

7.6 Memory configuration, in terms of total memory allocated and number of memory regions, for each of the applications considered. 179

7.7 Service downtime for each of the applications considered. 180

7.8 Evolution of memory occupancy and GPU utilization during execution time of two of the applications considered in this study. 182

7.9 Total migration time for the five applications considered in this study. . . 183

7.10 GPU migration used to consolidate servers. 186

7.11 Example of applying the GPU-job migration mechanism within rCUDA in order to balance the load among GPUs in the cluster. 188

List of Tables

2.1 Data transfers in the applications under analysis. 40

4.1 Scalability of the financial risk application when executed using CUDA. . 87

4.2 Time in seconds for GPU memory allocation and data transfer tasks of the financial risk application. 100

5.1 Performance achieved by several GROMACS configurations. 132

7.1 Amount of seconds required for management tasks in Figure 7.5(b). . . . 177

7.2 Characterization of the real applications used to analyze the migration mechanism. 178

7.3 Execution time of the CloverLeaf application in different GPUs. 190

Abstract

In the last decade the use of GPGPU (General Purpose computing in Graphics Processing Units) has become extremely popular in data centers around the world. GPUs (Graphics Processing Units) have been established as computational accelerators that are used alongside CPUs to form heterogeneous systems. The massively parallel nature of GPUs, traditionally intended for graphics computing, allows to perform numerical operations with data arrays at high speed. This is achieved thanks to the large number of cores GPUs integrate and the large bandwidth of memory access. Consequently, applications of all kinds of fields, such as chemistry, physics, engineering, artificial intelligence, materials science, and so on, presenting this type of computational patterns are benefited by drastically reducing their execution time.

In general, the use of computing acceleration provided by GPUs has meant a step forward and a revolution, but it is not without problems, such as energy efficiency problems, low utilization of GPUs, high acquisition and maintenance costs, etc.

In this PhD thesis we aim to analyze the main shortcomings of these heterogeneous systems and propose solutions based on the use of remote GPU virtualization. To that end, we have used the rCUDA middleware, developed at Universitat Politècnica de València. Many publications support rCUDA as the most advanced remote GPU virtualization framework nowadays.

The results obtained in this PhD thesis show that the use of rCUDA in Cloud Computing environments increases the degree of freedom of the system, as it allows to create virtual instances of the physical GPUs fully tailored to the needs of each of the virtual machines. In HPC (High Performance Computing) environments, rCUDA also provides a greater degree of flexibility in the use of GPUs throughout the computing cluster, as it allows the CPU part to be completely decoupled from the GPU part of the applications. In addition, GPUs can be on any node in the cluster, regardless of the node on which the CPU part of the application is running. In general, both for Cloud Computing and in

the case of HPC, this greater degree of flexibility translates into an up to 2x increase in system-wide throughput while reducing energy consumption by approximately 15%.

Finally, we have also developed a job migration mechanism for the GPU part of applications that has been integrated within the rCUDA middleware. This migration mechanism has been evaluated and the results clearly show that, in exchange for a small overhead of about 400 milliseconds in the execution time of the applications, it is a powerful tool with which, again, we can increase productivity and reduce energy foot print of the computing system.

In summary, this PhD thesis analyzes the main problems arising from the use of GPUs as computing accelerators, both in HPC and Cloud Computing environments, and demonstrates how thanks to the use of the rCUDA middleware these problems can be addressed. In addition, a powerful GPU job migration mechanism is being developed, which, integrated within the rCUDA framework, becomes a key tool for future job schedulers in heterogeneous clusters.

Resumen

En la última década la utilización de la GPGPU (General Purpose computing in Graphics Processing Units; Computación de Propósito General en Unidades de Procesamiento Gráfico) se ha vuelto tremendamente popular en los centros de datos de todo el mundo. Las GPUs (Graphics Processing Units; Unidades de Procesamiento Gráfico) se han establecido como elementos aceleradores de cómputo que son usados junto a las CPUs formando sistemas heterogéneos. La naturaleza masivamente paralela de las GPUs, destinadas tradicionalmente al cómputo de gráficos, permite realizar operaciones numéricas con matrices de datos a gran velocidad debido al gran número de núcleos que integran y al gran ancho de banda de acceso a memoria que poseen. En consecuencia, aplicaciones de todo tipo de campos, tales como química, física, ingeniería, inteligencia artificial, ciencia de materiales, etc. que presentan este tipo de patrones de cómputo se ven beneficiadas, reduciendo drásticamente su tiempo de ejecución.

En general, el uso de la aceleración del cómputo en GPUs ha significado un paso adelante y una revolución. Sin embargo, no está exento de problemas, tales como problemas de eficiencia energética, baja utilización de las GPUs, altos costes de adquisición y mantenimiento, etc.

En esta tesis pretendemos analizar las principales carencias que presentan estos sistemas heterogéneos y proponer soluciones basadas en el uso de la virtualización remota de GPUs. Para ello hemos utilizado la herramienta rCUDA, desarrollada en la Universitat Politècnica de València, ya que multitud de publicaciones la avalan como el framework de virtualización remota de GPUs más avanzado de la actualidad.

Los resutados obtenidos en esta tesis muestran que el uso de rCUDA en entornos de Cloud Computing incrementa el grado de libertad del sistema, ya que permite crear instancias virtuales de las GPUs físicas totalmente a medida de las necesidades de cada una de las máquinas virtuales. En entornos HPC (High Performance Computing; Computación de Altas Prestaciones), rCUDA también proporciona un mayor grado de flexibilidad de uso de las GPUs de todo el clúster de cómputo, ya que permite desacoplar

totalmente la parte CPU de la parte GPU de las aplicaciones. Además, las GPUs pueden estar en cualquier nodo del clúster, independientemente del nodo en el que se está ejecutando la parte CPU de la aplicación. En general, tanto para Cloud Computing como en el caso de HPC, este mayor grado de flexibilidad se traduce en un aumento hasta 2x de la productividad de todo el sistema al mismo tiempo que se reduce el consumo energético en un 15 %.

Finalmente, también hemos desarrollado un mecanismo de migración de trabajos de la parte GPU de las aplicaciones que ha sido integrado dentro del framework rCUDA. Este mecanismo de migración ha sido evaluado y los resultados muestran claramente que, a cambio de una pequeña sobrecarga, alrededor de 400 milisegundos, en el tiempo de ejecución de las aplicaciones, es una potente herramienta con la que, de nuevo, aumentar la productividad y reducir el gasto energético del sistema.

En resumen, en esta tesis se analizan los principales problemas derivados del uso de las GPUs como aceleradores de cómputo, tanto en entornos HPC como de Cloud Computing, y se demuestra cómo a través del uso del framework rCUDA, estos problemas pueden solucionarse. Además se desarrolla un potente mecanismo de migración de trabajos GPU, que integrado dentro del framework rCUDA, se convierte en una herramienta clave para los futuros planificadores de trabajos en clusters heterogéneos.

Resum

En l'última dècada la utilització de la GPGPU(General Purpose computing in Graphics Processing Units; Computació de Propòsit General en Unitats de Processament Gràfic) s'ha tornat extremadament popular en els centres de dades de tot el món. Les GPUs (Graphics Processing Units; Unitats de Processament Gràfic) s'han establert com a elements acceleradors de còmput que s'utilitzen al costat de les CPUs formant sistemes heterogenis. La naturalesa massivament paral·lela de les GPUs, destinades tradicionalment al còmput de gràfics, permet realitzar operacions numèriques amb matrius de dades a gran velocitat degut al gran nombre de nuclis que integren i al gran ample de banda d'accés a memòria que posseeixen. En conseqüència, les aplicacions de tot tipus de camps, com ara química, física, enginyeria, intel·ligència artificial, ciència de materials, etc. que presenten aquest tipus de patrons de còmput es veuen beneficiades reduint dràsticament el seu temps d'execució.

En general, l'ús de l'acceleració del còmput en GPUs ha significat un pas endavant i una revolució, però no està exempt de problemes, com ara poden ser problemes d'eficiència energètica, baixa utilització de les GPUs, alts costos d'adquisició i manteniment, etc.

En aquesta tesi pretenem analitzar les principals mancances que presenten aquests sistemes heterogenis i proposar solucions basades en l'ús de la virtualització remota de GPUs. Per a això hem utilitzat l'eina rCUDA, desenvolupada a la Universitat Politècnica de València, ja que multitud de publicacions l'avalen com el framework de virtualització remota de GPUs més avançat de l'actualitat.

Els resultats obtinguts en aquesta tesi mostren que l'ús de rCUDA en entorns de Cloud Computing incrementa el grau de llibertat del sistema, ja que permet crear instàncies virtuals de les GPUs físiques totalment a mida de les necessitats de cadascuna de les màquines virtuals. En entorns HPC (High Performance Computing; Computació d'Altes Prestacions), rCUDA també proporciona un major grau de flexibilitat en l'ús de les GPUs de tot el clúster de còmput, ja que permet desacoblar totalment la part CPU de la part GPU de les aplicacions. A més, les GPUs poden estar en qualsevol node del

clúster, sense importar el node en el qual s'està executant la part CPU de l'aplicació. En general, tant per a Cloud Computing com en el cas del HPC, aquest major grau de flexibilitat es tradueix en un augment fins 2x de la productivitat de tot el sistema al mateix temps que es redueix el consum energètic en aproximadament un 15%.

Finalment, també hem desenvolupat un mecanisme de migració de treballs de la part GPU de les aplicacions que ha estat integrat dins del framework rCUDA. Aquest mecanisme de migració ha estat avaluat i els resultats mostren clarament que, a canvi d'una petita sobrecàrrega, al voltant de 400 mil·lisegons, en el temps d'execució de les aplicacions, és una potent eina amb la qual, de nou, augmentar la productivitat i reduir la despesa energètica de sistema.

En resum, en aquesta tesi s'analitzen els principals problemes derivats de l'ús de les GPUs com acceleradors de còmput, tant en entorns HPC com de Cloud Computing, i es demostra com a través de l'ús del framework rCUDA, aquests problemes poden solucionar-se. A més es desenvolupa un potent mecanisme de migració de treballs GPU, que integrat dins del framework rCUDA, esdevé una eina clau per als futurs planificadors de treballs en clústers heterogenis.

Chapter 1

Introduction

Abstract

This PhD thesis has been prepared by compiling a compendium of the most important publications that have emerged from the research carried out during its execution. In this first chapter, the key concepts on which all the research is based will be introduced. More specifically, the concept of GPGPU will be introduced, as well as the advantages and problems that the use of these heterogeneous computing systems bring to modern data centers. Next, the main motivation for this PhD thesis will be exposed: the use of remote virtualization of GPUs, by using the rCUDA middleware, in order to address most of the problems generated by the use of these heterogeneous computing systems. Finally, the objectives and main contributions of this PhD thesis will be shown.

1.1 Background

1.1.1 GPGPU

Given the high computational requirements of many of today's applications, both academia and industry are widely using GPUs (Graphics Processing Units) to accelerate the execution time of applications. GPUs are devices with a large number of cores and a large memory access bandwidth. These features provide them with a high level of parallelism and efficiency to work with numerical arrays. For this type of operations, GPUs are much faster than current CPUs. Moreover, this trend has been exacerbated by the fact that the GPU technology has traditionally been linked to the video game market. Thus achieving large production volumes over time, in addition to large computational power. The end result is that GPUs have become a widely accepted and efficient way to reduce the execution time of many applications. Therefore, an adequate use of GPUs together with CPUs allows a notable reduction in the execution time of many applications. This has led to the use of these devices in areas as diverse as computational algebra [1], finance [2], artificial intelligence [3], fluid dynamics [4], chemical physics [5] or image analysis [6], among others.

To accelerate applications using GPUs, these devices run the computationally intensive parts of the application. This offloading on the GPUs of the computationally intensive part of an application requires the programmer to explicitly specify which parts of the application code will run on the traditional CPU and which parts on the GPU. To help the programmer in this task there are different solutions. Two of the best known and most used solutions today are CUDA [7] and OpenCL [8]. Using GPUs in order to accelerate applications makes up what is known as GPGPU (General Purpose Computing on GPUs). Although CUDA is a proprietary solution from the company NVIDIA and OpenCL is an open standard, in the field of GPUs the use of CUDA has become much more widespread than the use of OpenCL due, among other reasons, to the better features it offers and the support available from a large company like NVIDIA.

1.1.2 Main Concerns of Using GPUs

The common way in which GPUs are used in high-performance supercomputers and data centers is to install one or more of these accelerators on each computer in the cluster. However, although this configuration is interesting from a performance point of view, from a power consumption point of view it is not efficient, since a single GPU can easily consume 25% of the total power of a computer at the same time that GPUs are typically never used 100% of the time, no matter how high the level of parallelism of an application is. In fact, it is common that average utilization of GPUs is not greater than 10% or 20%. Therefore, the configuration used today to exploit the computational resources of GPUs is very inefficient, both at an energy level and at the level of the acquisition cost of the equipment. This idea is supported by the recent advances of NVIDIA in the area of GPU virtualization. In this regard, NVIDIA provides the vGPU [9] and MIG [10] solutions in order to address these concerns.

A more interesting computer cluster configuration would be to reduce the number of GPUs installed in it and to concurrently share the installed GPUs among applications. This would result in a higher utilization of the installed GPUs, a lower acquisition cost and also a lower energy consumption. This is the main purpose of the vGPU and MIG solutions recently provided by NVIDIA. However, this new configuration entails great difficulty when scheduling the use of computers that include a GPU, since mapping tasks to cluster computers becomes much more complex. In this regard, notice that making it possible to simultaneously share a GPU among several applications would further increase the complexity of task scheduling, which would often result in an unbalanced distribution of jobs where computers with GPUs would typically be saturated while computers without GPUs would have much lower utilization. A better approach is to make use of the remote GPU virtualization mechanism.

1.1.3 Remote GPU Virtualization

A possible solution to achieve an efficient cluster configuration with fewer GPUs than computers is the virtualization of the accelerators. In general, virtualization techniques (used for example to create virtual machines) allow to reduce the costs of acquisition, maintenance, administration, space and energy consumption in data centers. The usual

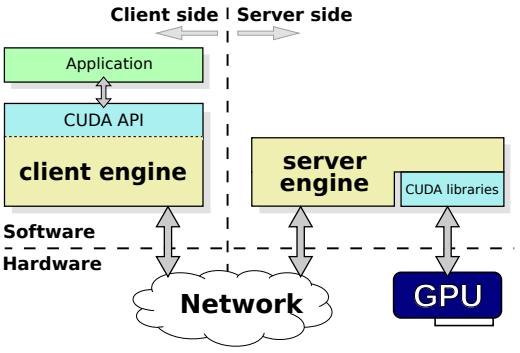


FIGURE 1.1: Architecture of the rCUDA middleware.

approach in these data centers is to use virtual machines, which provide the user with the illusion of being dedicated computers exclusively for him. In the case of remote virtualization of GPUs, these would be installed only in some cluster computers, which would act as servers for the rest of the nodes, which would use the GPUs concurrently, thus increasing their utilization and reducing costs due to both acquisition and energy consumption.

In order to provide a solution based on this idea, several GPU virtualization environments have been developed over the last few years. Among others we can name, for example, rCUDA [11, 12], vCUDA [13], GViM [14], GVirtuS [15], V-GPU [16] and GridCUDA [17] which pursue the virtualization of the CUDA runtime API (Application Programming Interface). OpenCL, VCL [18] and SnuCL [19] present work environments with similar characteristics to the environments created for CUDA. Regarding the environments focused on the CUDA API, it should be noted that, with the exception of rCUDA, the rest of the existing solutions only provide partial support for obsolete CUDA versions, which makes them non-interesting from a practical point of view and, at an industrial level, are not usable. The exception to this is GVirtuS, which provides very limited support to modern CUDA versions. In any case, this limited support makes GVirtuS not useful for industry.

1.1.4 rCUDA

rCUDA (remote CUDA) fully supports the CUDA API and is compatible with its latest versions. It has been created within the Parallel Architectures Group of Universitat Politècnica de València, thanks to the funding provided by Generalitat Valenciana

through three research Prometeo projects as well as other funding sources. The rCUDA technology allows the use of remote GPUs compatible with the CUDA library. For this reason it is said that the rCUDA environment implements the remote GPU virtualization mechanism. Figure 1.1 shows the architecture of the rCUDA middleware, which follows a client-server approach. The rCUDA server runs on the cluster computers where GPUs are installed, which serve the requests that come from the rCUDA clients. rCUDA clients run on the cluster computers that do not have a GPU and are presented to applications as the CUDA library. In this way, accelerated applications can be executed in any of the cluster nodes and when they need the services of a GPU they use the rCUDA client of their computer, in a transparent way from the application point of view, to send their requests to the actual GPU that is installed on another computer in the cluster. It should be noted that the application is not aware that it is interacting with a virtual remote GPU, but believes that it has exclusive access to a real GPU installed on the local computer. Also, the additional overhead introduced by rCUDA is minimal. In the case of using a high-performance network such as InfiniBand, rCUDA presents an overhead in the execution times of the applications around 3% compared to the usual configuration in which the GPU is used locally with CUDA. rCUDA can also work with Ethernet networks and its overhead will depend on the performance of the underlying capabilities of the network.

1.2 Objectives of the Thesis

The main objective of this PhD thesis is to demonstrate how, thanks to the use of the remote GPU virtualization middleware rCUDA, we can be able to improve the performance of current data centers, which are based on heterogeneous configurations where the computation is distributed between the CPU and the GPU.

Virtualization techniques are widespread and widely used in many data centers intended for Cloud Computing since this type of systems rely on a dynamic and highly adaptable infrastructure to meet varying demands by offering different resources as services. A first objective of this PhD thesis will be to evaluate both the feasibility of using rCUDA in virtualized environments and the possible advantages that this technology can offer compared to classic configurations.

High performance computing is one of the fields where the use of GPUs as computing accelerators is more widespread due to the great improvement in performance offered by these types of configurations. A second objective of this thesis will be to design techniques based on remote GPU virtualization in order to improve the performance of HPC systems while not increasing overall energy consumption.

The last major objective of this PhD thesis is to create a mechanism that allows migrating the computation that is being carried out in a GPU to another GPU in the cluster, without affecting the part of the application that is being executed in the CPU. That is, when we use a GPU to accelerate a given application, this application will not necessarily be anchored to using the same GPU throughout its life time, but, thanks to the migration mechanism, the GPU will be able to change according to different criteria based on several metrics, such as throughput, energy efficiency, quality of service, etc.

1.3 Main Contributions of the Thesis

The major contributions of this dissertation are described below:

- A first contribution has been the analysis of the evolution of GPU utilization over time in traditional clusters. As a result of this analysis, we have learned how to efficiently share a GPU between different applications.
- The study of using GPU computing acceleration in virtualized environments shows that the average utilization of these devices is, in general, very low. We propose to use remote GPU virtualization in this environment. By using remote GPU virtualization, two great advantages over traditional configurations in this type of environments are achieved: (i) we can use local or remote GPUs with respect to the node hosting the virtual machines. (ii) We can create virtual instances of the physical GPUs completely tailored to the needs of the different virtual machines, thus being able to assign different virtual instances of a physical GPU to one or more virtual machines at the same time. The study carried out in this PhD thesis, with real hardware and applications, shows that these two benefits have a real impact and provide a considerable improvement in the utilization ratio of

the GPUs. This ends up having an effect on better energy efficiency and higher system throughput.

- In HPC environments we have a very similar panorama to that described for virtualized Cloud Computing environments. The analysis carried out in the studies in this thesis shows a low utilization of GPUs due to the limitations of traditional job schedulers. We propose to use remote GPU virtualization in HPC environments. Actually, with rCUDA, in this thesis we exploit the concept of multi-tenancy from two different perspectives: (i) we assign to an application several virtual GPUs on the same physical GPU. In this way we manage to overlap memory transfers and computation and therefore we increase the utilization of the GPU at the same time that we reduce the execution time of the application. (ii) We create different virtual GPUs on the same physical GPU but in this case we assign them to different applications. With this approach what we achieve is to increase the amount of GPU-accelerated applications that a data center can run in a period of time and consequently increase the throughput of the entire system.

- We propose to enrich job schedulers with the capability of GPU job migration. To that end, we developed a GPU job migration mechanism and evaluated it. The obtained results show that the impact on the execution time of applications of this mechanism is minimal when a high-performance interconnection network is used. Moreover, it has also been shown that the use of this mechanism, by future job schedulers, will offer a great flexibility to the system. This will provide a better use of GPUs.

1.4 Thesis Outline

This PhD thesis is divided into eight chapters and is provided as a compendium of the main publications generated during the research. This thesis follows the regulations of Universitat Politècnica de València: each of the six central chapters corresponds to each of the six main publications generated in the course of this thesis, with changes only in format but not in contents. Therefore each of these six chapters has its own references according to the original publications. Chapters 1 and 8, introduction and conclusions, have been drafted to provide consistency and coherency to the document. In order to

follow the same structure as the other six chapters, the references appeared in these chapters are also included at the end of the chapter. Chapters 2 and 3 are related to the first objective: the use of rCUDA in Cloud Computing environments. Chapters 4 and 5 evaluate the impact of using remote GPU virtualization in HPC configurations, that is, the second objective. Finally, Chapters 6 and 7 are related to the third objective: in these chapters the GPU-job migration mechanism implemented in this PhD thesis is presented and evaluated.

The content of each of the chapters in this manuscript is the following:

- Chapter 1 has introduced the thesis, objectives, and contributions.
- Chapter 2 includes the publication *On the effect of using rCUDA to provide CUDA acceleration to Xen virtual machines*. In this chapter we demonstrate the feasibility of using rCUDA in a virtual machine environment using the Xen hypervisor. `https://doi.org/10.1007/s10586-018-2845-0`
- Chapter 3 includes the publication *Made-to-Measure GPUs on Virtual Machines with rCUDA*. In this chapter the research on virtual machines using the KVM hypervisor is expanded. In addition several virtual instances of a physical GPU are concurrently shared among different virtual machines. `https://doi.org/10.1145/3229710.3229741`
- Chapter 4 includes the publication *Multi-tenant virtual GPUs for optimising performance of a financial risk application*. In this chapter we use rCUDA in order to overlap data transfers and computation in the GPUs. The results obtained show that this produces a reduction in both execution time and energy consumed by a financial risk analysis application. `https://doi.org/10.1016/j.jpdc.2016.06.002`
- Chapter 5 includes the publication *Maximizing resource usage in multifold molecular dynamics with rCUDA*. In this chapter the virtualization of GPUs is exploited in order to increase the number of concurrent molecular simulations in a data center. `https://doi.org/10.1177/1094342019857131`
- Chapter 6 includes the publication *Turning GPUs into Floating Devices over the Cluster: The Beauty of GPU Migration*. This chapter shows the preliminary results

of the GPU-job migration mechanism that we have developed in this PhD thesis. https://doi.org/10.1109/ICPPW.2017.30

- Chapter 7 includes the publication *GPU-Job Migration: The rCUDA Case*. In this chapter the migration mechanism is shown in detail and it is evaluated in depth. Several use cases are also shown in order to demonstrate its potential. https://doi.org/10.1109/TPDS.2019.2924433

- Chapter 8 summarizes this thesis, discusses future work, and enumerates the related publications.

References

[1] Ichitaro Yamazaki, Tingxing Dong, Raffaele Solcà, Stanimire Tomov, Jack Dongarra, and Thomas Schulthess. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, 26(16):2652–2666, 2014.

[2] Vladimir Surkov. Parallel option pricing with Fourier space time-stepping method on graphics processing units. *Parallel Computing*, 36(7):372–380, 2010.

[3] Guo-Heng Luo, Sheng-Kai Huang, Yue-Shan Chang, and Shyan-Ming Yuan. A parallel Bees Algorithm implementation on GPU. *Journal of Systems Architecture*, 60(3):271–279, 2014.

[4] Everett H. Phillips et al. Rapid aerodynamic performance prediction on a Cluster of graphics processing units. In *AIAA*, 2009.

[5] 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.

[6] Yuancheng Luo and R. Duraiswami. Canny edge detection on nvidia cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, June 2008.

[7] NVIDIA. CUDA C Programming Guide. Design Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2020. Accessed 18 December 2020.

[8] Khronos OpenCL Working Group. *OpenCL 2.0 Specification*, 2013.

[9] NVIDIA Virtual GPU Software User Guide. `https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html`, 2021. Accessed 10 January 2021.

[10] NVIDIA Multi-Instance GPU User Guide. `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html`, 2021. Accessed 14 January 2021.

[11] Carlos Reaño, Federico Silla, and Jose Duato. Enhancing the rCUDA Remote GPU Virtualization Framework: From a Prototype to a Production Solution. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 695–698. IEEE Press, 2017.

[12] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, pages 4:1–4:7. ACM, 2015.

[13] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proc. of the IEEE Parallel and Distributed Processing Symposium, IPDPS*, pages 1–11, 2009.

[14] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt*, pages 17–24, 2009.

[15] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proc. of the Euro-Par Parallel Processing, Euro-Par*, pages 379–391, 2010.

[16] V-GPU: GPU virtualization. `https://github.com/zillians/platform_manifest_vgpu`, 2015. Accessed 14 January 2021.

[17] Tyng Yeu Liang and Yu Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Proc. of the IEEE Advanced Information Networking and Applications Workshops, WAINA*, pages 141–146, 2011.

[18] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, 2010.

[19] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012.

Chapter 2

On the Effect of using rCUDA to Provide CUDA Acceleration to Xen Virtual Machines

Javier Prades, Carlos Reaño, Federico Silla. **Cluster Computing The Journal of Networks, Software Tools and Applications** - Volume: 22 - March. 15 2019 - Pages 185 - 204
https://doi.org/10.1007/s10586-018-2845-0

Abstract

Nowadays, many data centers use virtual machines (VMs) in order to achieve a more efficient use of hardware resources. The use of VMs provides a reduction in equipment and maintenance expenses as well as a lower electricity consumption. Nevertheless, current virtualization solutions, such as Xen, do not easily provide graphics processing units (GPUs) to applications running in the virtualized domain with the flexibility usually required in data centers (i.e., managing virtual GPU instances and concurrently sharing them among several VMs). Therefore, the execution of GPU-accelerated applications within VMs is hindered by this lack of flexibility. In this regard, remote GPU virtualization solutions may address this concern.

In this paper we analyze the use of the remote GPU virtualization mechanism to accelerate scientific applications running inside Xen VMs. We conduct our study with six different applications, namely CUDA-MEME, CUDASW++, GPU-BLAST, LAMMPS, a triangle count application, referred to as TRICO, and a synthetic benchmark used to emulate different application behaviors. Our experiments show that the use of remote GPU virtualization is a feasible approach to address the current concerns of sharing GPUs among several VMs, featuring a very low overhead if an InfiniBand fabric is already present in the cluster.

Keywords: Virtualization, CUDA, Xen, InfiniBand, HPC, Performance.

2.1 Introduction

Virtual machines (VMs) have demonstrated to provide economic savings to data centers, the main reason being that several VMs can be concurrently executed in a single cluster node thus sharing its CPUs as well as other subsystems and, therefore, increasing overall resource utilization. Acquisition and maintenance costs are therefore reduced because a smaller amount of servers is required to address the same workload, thus reducing also energy consumption needs. In this way, the use of VMs is the basis for cloud computing services like the ones provided by Amazon and other PaaS (Platform as a Service) providers.

The benefits provided by VMs have caused that virtualization solutions such as KVM [1], Xen [2], VMware [3], or VirtualBox [4] become very popular. Actually, the benefits reported by the use of VMs have motivated that leading processor manufacturers such as Intel or AMD have increasingly incorporated more support for virtualization into their chip designs [5]. Moreover, although VMs were known in the past for reducing application performance with respect to executions in the native (or real) domain, the virtualization features included in current CPUs allow VMs to execute applications with a negligible overhead [6]. This has led some authors to suggest using VMs in the context of high-performance computing (HPC) [7].

However, despite the many advances accomplished in the field of VMs, they still do not support efficiently the current trend of using the CUDA[1] compute platform to use the graphics processing units (GPUs) as accelerators, which allows significantly reducing the time required to execute applications from areas as different as data analysis (Big Data) [8], chemical physics [9], computational algebra [10], image analysis [11], finance [12], biology [13], and artificial intelligence [14], to name just a few. In this regard, there have been several recent achievements in order to virtualize GPUs, like the new GRID K1 GPU by NVIDIA [15], which can be shared among up to eight VMs and is mainly intended for desktop virtualization, although it can also be used for executing CUDA programs. Nevertheless, given that this device only features 192 CUDA cores

[1]CUDA (Compute Unified Device Architecture) is a technology created by NVIDIA which comprises a parallel compute platform (CUDA-enabled graphics processing units) as well as an application programming interface (API) and a compiler.

per GPU, its applicability to scientific computing is very limited[2]. Other examples of including virtualization support into GPUs are the recent KVMGT[3] technology by Intel [16] and the new Multiuser GPU by AMD [17], which provide virtualization support for Intel and AMD GPUs, respectively. Unfortunately, these solutions do not support CUDA acceleration. Therefore, the lack of efficient support for CUDA-compatible GPUs in current virtualization solutions makes that applications running in the virtualized domain cannot easily access these GPUs for acceleration purposes. From the point of view of cloud computing providers such as Amazon, this means that the investment made in GPUs cannot be amortized as fast as possible as it will be further described in next section.

In this paper we explore the use of the remote GPU virtualization mechanism in order to provide CUDA acceleration to applications running inside Xen VMs. The main motivation is that GPU virtualization solutions such as V-GPU [18], DS-CUDA [19], rCUDA [20, 21], vCUDA [22], GridCuda [23], GVirtuS [24], GViM [25], Shadowfax [26], or Shadowfax II [27] may be used in VM environments in order to address their current limitations with respect to GPUs. These GPU virtualization frameworks detach GPUs from nodes, thereby allowing applications to access virtualized GPUs regardless of the exact computer where they are being executed. Thus, the detaching features of remote GPU virtualization solutions may turn them into an easy and efficient way to overcome the current limitations of VMs regarding the use of GPUs as accelerators.

The aim of this study is to assess the overhead that applications experience when accessing GPUs outside their Xen VM by using the remote GPU virtualization approach. To that end, we investigate two different scenarios. In the first one, an application within a VM accesses a GPU located in the same computer hosting it. In the second scenario we assume that a high performance network fabric such as InfiniBand (IB) is available in the cluster and the application running inside the VM accesses a GPU located in another cluster node. In this study we use the rCUDA remote GPU virtualization middleware

[2]In addition to the GRID K1 GPU, NVIDIA has also brought to market the GRID K2 model, which features 1536 CUDA cores per GPU and 4 GB of memory. However, this amount of resources per GPU is still noticeable smaller than the ones available in current NVIDIA Tesla K20 and K40 GPUs, featuring, respectively, 2496 and 2880 CUDA cores and 5GB and 12GB of memory. Therefore, using the GRID K2 device for providing acceleration to scientific applications instead of providing desktop virtualization would deliver a significantly lower performance than current mainstream GPUs used in HPC servers, such as the K20 or K40 GPUs.

[3]KVMGT is the open source implementation of Intel's GPU Virtualization Technology for KVM VMs.

because it was the sole solution able to run the applications considered in our analysis. As can be seen, the main contribution of this paper is testing the performance of the rCUDA remote GPU virtualization middleware in the context of Xen VMs.

The rest of the paper is organized as follows. Section 2.2 thoroughly reviews previous efforts to provide GPU acceleration to applications being executed inside VMs and further motivates the use of general GPU virtualization frameworks to provide CUDA acceleration to VMs. Later, Section 2.3 introduces rCUDA in more detail whereas Section 2.4 presents the experimental setup used in this paper. Section 2.5 analyzes the network performance observed by Xen VMs when making use of the virtual network connecting VMs within a host as well as the performance of the InfiniBand interconnect. Section 2.6 uses these results to analyze the performance of rCUDA when used from the inside of Xen VMs. Next, Section 2.7 addresses the main goal of this paper: studying the throughput of real GPU-accelerated applications when executed within Xen VMs. A synthetic benchmark is also leveraged in order to emulate several interesting features of application behavior. Finally, Section 2.8 summarizes the main conclusions of our work.

2.2 Providing CUDA GPUs to Virtual Machines

Providing CUDA acceleration to VMs can be accomplished by making use of the PCI passthrough technique [28, 29]. This mechanism is based on the use of the virtualization extensions widely available in current HPC servers, which allow assigning a GPU, in an exclusive way, to one of the VMs running at the host. Furthermore, when making use of this mechanism, the performance attained by accelerators is very close to that obtained when using the GPU in a native domain. Unfortunately, as this approach assigns GPUs to VMs in an exclusive way, only one of the VMs can access the GPU. This means, for instance, that for the CG1 VM instances of Amazon, which make use of the M2050 NVIDIA GPU, only one of the VMs being executed in a given host can be assigned the GPU at the same time. This exclusive assignment of the GPU to a single VM causes an underutilization of resources because the computation capabilities of the GPU cannot be leveraged by other VMs when the VM that owns the GPU does not use it. Furthermore, this exclusive assignment means that the amount of CG1 VM

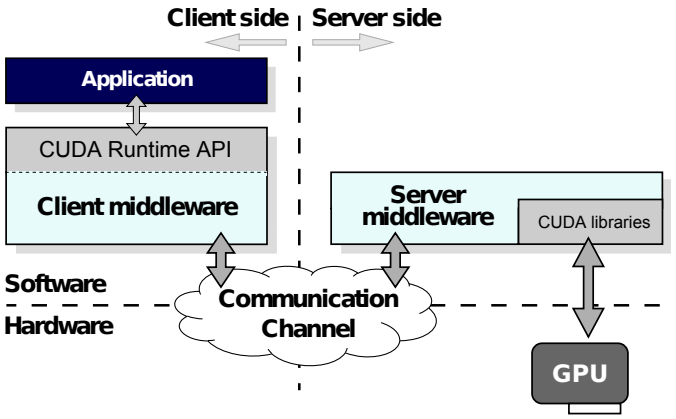


FIGURE 2.1: Typical architecture used by GPU virtualization solutions.

instances that can be concurrently in execution in a given node is limited by the amount of GPUs installed in that node. This limits the economic profit that a data center can obtain from the underlying hardware, causing that the initial investment requires more time to be amortized.

To address the concern about the exclusive assignment of GPUs to VMs, there have been several attempts, like the one proposed in [30], which dynamically changes on demand the GPUs assigned to VMs. However, these techniques present two important concerns: (1) a high time overhead is generated given that, in the best case, two seconds are required to change the assignment between GPUs and VMs; (2) These techniques do not address the impossibility of sharing GPUs among several VMs simultaneously.

For these reasons, several software-based GPU sharing mechanisms have appeared, such as, for example, V-GPU, DS-CUDA, rCUDA, vCUDA, and GridCuda. Basically, these middleware proposals share a GPU by virtualizing it, so that these middleware systems provide applications (or VMs) with virtual instances of the real device, which can therefore be concurrently shared. Usually, these GPU sharing solutions place the virtualization boundary at the API level[4] (CUDA [31] in the case of NVIDIA GPUs). In general, CUDA-based virtualization frameworks aim to offer the same API as the NVIDIA CUDA Runtime API [32] does.

[4]In order to interact with the virtualized GPU, some kind of interface is required so that the application can access the virtual device. This interface could be placed at different levels. For instance, it could be placed at the driver level. However, GPU drivers usually employ low-level protocols which, additionally, are proprietary and strictly closed by GPU vendors. Therefore, a higher-level boundary must be used. This is why the GPU API is commonly selected for placing the virtualization boundary, given that these APIs are public.

Figure 2.1 depicts the architecture usually deployed by these GPU virtualization solutions, which follow a distributed client-server approach. The client part of the middleware is installed in the domain (either native or virtual)[5] executing the application requesting GPU services, whereas the server side runs in the domain owning the actual GPU. Communication between client and server may be based on shared-memory mechanisms or on the use of a network fabric, depending on the exact features of the GPU virtualization middleware and the underlying system configuration.

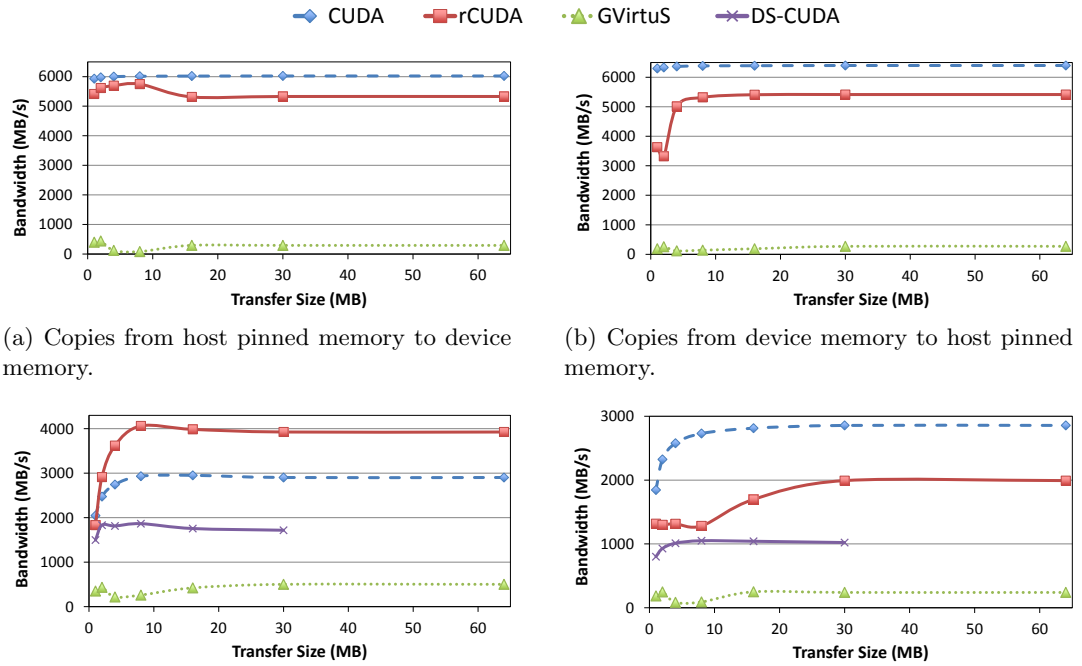
The architecture depicted in Figure 2.1 is used in the following way: the client middleware receives a CUDA request from the accelerated application and appropriately processes and forwards it to the server middleware. In the server side, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client middleware, which forwards them to the accelerated application. Notice that GPU virtualization solutions provide GPU services in a transparent way and, therefore, applications are not aware that their requests are actually serviced by a virtual GPU instead of by a local one. The following piece of code shows an example of a CUDA program that we will use to further explain how the architecture in Figure 2.1 works.

```
#include <cuda.h>
#include <stdio.h>
const int N = 8;

// Function that will be executed in the GPU
__global__ void my_gpu_function(int *a, int *b)
{
  b[threadIdx.x] = a[threadIdx.x] * 2;
}

int main()
{
  int a[N] = {0, 1, 2, 3, 4, 5, 6, 7};
  int *ad, *bd;
  const int isize = N*sizeof(int);

  // Perform some computations in the CPU
  CPU code 1
  CPU code 2
```

[5]The native domain refers to a scenario where virtualization is not used, that is, a real computer is leveraged. On the other hand, the virtual domain refers to the virtual machine.

```
  ...


  // Allocate GPU memory
  cudaMalloc( (void**)&ad, isize );
  cudaMalloc( (void**)&bd, isize );

  // Copy data to GPU memory
  cudaMemcpy( ad, a, isize, cudaMemcpyHostToDevice );

  // Run function in the GPU
  my_gpu_function<<<1, N>>>(ad, bd);

  // Copy results from GPU memory
  cudaMemcpy( b, bd, isize, cudaMemcpyDeviceToHost );

  // Free GPU memory
  cudaFree( ad );
  cudaFree( bd );

  return 0;
}
```

When the previous program is executed, not using a GPU virtualization framework, the CUDA library is loaded. However, when the program is executed to make use of a virtual GPU, the original CUDA library by NVIDIA is not loaded but another library with the same name is loaded. This other library contains a set of wrappers to the original CUDA functions that take care of the virtualization process. In this way, all the CPU code is executed in the same way as before but as soon as a call to a CUDA function is performed, the appropriate wrapper in the second library is called. For example, when the cudaMalloc function in line 24 is called, the wrapper for that function receives the arguments and forwards them to the middleware server, along with the function code assigned to the cudaMalloc function. Once the function code and the arguments arrive at the middleware server, the actual cudaMalloc function is executed in the real GPU by making use of the received arguments and the result of the call (status code) is collected. This status code is sent back to the client middleware, which is waiting for it. Upon reception of the status code, the client middleware delivers it to the application which continues with the execution of the program. The rest of CUDA calls shown in the example code in lines 25, 28, 31, 34, 37, and 38 are processed in a similar way.



(a) Copies from host pinned memory to device memory.

(b) Copies from device memory to host pinned memory.

(c) Copies from host pageable memory to device memory.

(d) Copies from device memory to host pageable memory.

FIGURE 2.2: Performance comparison among three different GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. The comparison is performed in terms of attained bandwidth. The performance of CUDA is also depicted. Tests have been carried out in native domains with the hardware and software settings described in Section 2.4.

CUDA-based GPU virtualization frameworks may be classified into two types: (1) those intended to be used in the context of VMs and (2) those devised as general purpose virtualization solutions, to be used in native domains (notice that these latter solutions may also be used within VMs). Middleware systems in the first category usually make use of shared-memory mechanisms in order to transfer data from main memory inside the VM to the GPU in the native domain, whereas the general purpose virtualization solutions in the second type make use of the network fabric in the cluster to transfer data from main memory in the client side to the remote GPU located in the server. This is why these latter solutions are commonly known as remote GPU virtualization middleware systems.

Regarding the first type of GPU virtualization solutions mentioned above, several frameworks have been developed, as for example vCUDA, GViM, gVirtuS, and Shadowfax. The vCUDA technology, intended for Xen VMs, only supports the old CUDA version 3.2 and implements an unspecified subset of the CUDA Runtime API. Moreover, its

communication protocol presents a considerable overhead, because of the cost of the encoding and decoding stages, which causes a noticeable drop in overall performance. GViM, targeting Xen VMs, is based on the obsolete CUDA version 1.1 and, in principle, does not implement the entire CUDA Runtime API. gVirtuS is based on the old CUDA version 6.5 and implements only a small portion of its API. Despite being designed for VMs, it also provides TCP/IP communications for remote GPU virtualization, thus allowing applications in a non-virtualized environment to access GPUs located in other nodes. Regarding Shadowfax, this solution allows Xen VMs to access the GPUs located at the same node, although it may also be used to access GPUs at other nodes of the cluster. It supports the obsolete CUDA version 1.1. Notice that among the virtualization frameworks decribed in this group, only the gVirtuS solution is publicly available.

In the second type of virtualization solutions mentioned above, which provide general purpose GPU virtualization, one can find rCUDA, V-GPU, GridCuda, DS-CUDA, and Shadowfax II. rCUDA, further described in Section 2.3, features CUDA 8.0 and provides specific communication support for TCP/IP compatible networks as well as for InfiniBand and RoCE fabrics. V-GPU is a recent tool supporting CUDA 4.0. Unfortunately, the information provided by the V-GPU authors is fuzzy and there is no publicly available version that can be used for testing and comparison. GridCuda also offers access to remote GPUs in a cluster, but supports the old CUDA version 2.3. Moreover, there is currently no publicly available version of GridCuda that can be used for testing. Regarding DS-CUDA, it integrates an old version of CUDA (4.1) and includes specific communication support for InfiniBand. However, DS-CUDA presents several strong limitations, such as not allowing data transfers with pinned memory. Finally, Shadowfax II is still under development, not presenting a stable version yet and its public information is not updated to reflect the current code status. Among these remote GPU virtualization solutions, only the DS-CUDA and rCUDA frameworks are publicly available.

In order to provide a comprehensive comparison among the different GPU virtualization solutions described in this section, Figure 2.2 presents a performance comparison of the three publicly available GPU virtualization solutions: DS-CUDA, rCUDA, and gVirtuS. This figure also shows the performance of CUDA as the baseline reference. The widely used `bandwidthTest` benchmark from the NVIDIA CUDA Samples [33] has

been employed. The reason for using bandwidth for measuring performance is that, when transferring data between main memory and GPU memory, data copy sizes are, in general, large (in the order of MB) as it will be shown in Section 2.7. These large data transfers are mostly influenced by attained bandwidth, which turns out to be the most limiting factor regarding the performance of these solutions. Consequently, other metrics such as latency are less relevant in this context.

The testbed employed for carrying out the performance experiments is the one described in Section 2.4, although no virtual machine has been used in order to simplify the experiments. In this way, the bandwidth test was run in a native domain whereas the server side of the middleware systems was executed in a remote computer. The InfiniBand FDR network technology was used to connect both computers. Therefore, both the rCUDA and DS-CUDA middleware systems made use of the InfiniBand Verbs API. In the case of gVirtuS, given that it is not able to take advantage of the InfiniBand Verbs API, TCP/IP over InfiniBand was used.

One additional consideration to be made regarding the experiments shown in Figure 2.2 is that the three GPU virtualization middleware systems analyzed support different versions of CUDA. Thus, each of the frameworks has been analyzed with the respective version of CUDA supported. In this regard, it is important to remark that, in order to avoid introducing additional noise in this particular test, we have previously compared the bandwidth achieved by the three versions of CUDA used and the result is that differences in performance for the bandwidth test are negligible from one CUDA version to another.

Results in Figure 2.2 deserve some discussion. First, it can be seen that CUDA achieves the highest performance when pinned memory is used (Figures 2.2(a) and 2.2(b)), attaining a bandwidth around 6000 MB/s. Notice that this bandwidth is reduced for copies using pageable memory (Figures 2.2(c) and 2.2(d)). Second, Figure 2.2 shows that rCUDA outperforms the other two remote GPU virtualization solutions. Actually, for copies from host to device memory using pageable memory rCUDA also performs better than CUDA. This is a well-known effect thoroughly described in previous works on rCUDA [21] and is due to the use of an efficient pipelined communication based on the use of internal pre-allocated pinned memory buffers. On the other hand, notice that both rCUDA and DS-CUDA make use of the InfiniBand Verbs API, thus having

access to the large bandwidth available in this interconnect. However, although rCUDA is able to struggle an important fraction of the available bandwidth, DS-CUDA presents a relatively poor performance. Therefore, it must be assumed that the difference in bandwidth is due to the different way that both GPU virtualization solutions manage the InfiniBand interconnect. Also notice that DS-CUDA supports neither memory copies larger than 32MB nor the use of pinned memory. Futhermore, notice that the performance of gVirtuS is extremely low. One may think that this is due to the fact that gVirtuS is using TCP/IP over InfiniBand, which clearly achieves lower performance than the InfiniBand Verbs API. However, according to our measurements with the iperf tool [34], when TCP/IP is used over InfiniBand FDR, a bandwidth around 1190 MB/s is achieved, which is a noticeably larger bandwidth than the one attained by gVirtuS. Hence, the low performance of this middleware is not only due to the use of TCP/IP over InfiniBand but also to the way it internally manages communications.

As a final consideration for this review section, it is important to remark that although remote GPU virtualization has traditionally introduced a non-negligible overhead, given that applications do not access GPUs attached to the local PCI Express (PCIe) link but rather access devices that are installed in other nodes of the cluster (traversing a network fabric with a lower bandwidth), this performance overhead has significantly been reduced thanks to the recent advances in networking technologies. For example, the rCUDA middleware is able to achieve 98% [35] of the native bandwidth of the Tesla K40 GPU when making use of FDR dual-port network adapters (providing 12.5GB/s of effective bandwidth) [36]. In the case of using the previous generation of these technologies, NVIDIA Tesla K20 GPUs and InfiniBand FDR single-port network adapters (6GB/s of effective bandwidth) [37], Figure 2.2 shows that bandwidth attained by rCUDA is very close to that of CUDA, except for copies from device to host memory using pageable memory, which still need some refinement. Therefore, when using remote GPU virtualization solutions, the path communicating main memory in the computer executing the application and the remote accelerator presents, in general, a bandwidth similar to that initially attained by the original CUDA approach of using local GPUs.

2.3 rCUDA: Remote CUDA

As already mentioned in the introduction section, we use in this study the rCUDA middleware given that it was the only one able to run the applications analyzed in this paper, as well as being the most up-to-date solution, providing also the best performance among the different publicly available GPU virtualization solutions. In this section we introduce rCUDA in more detail.

The rCUDA middleware supports version 8.0 of CUDA, being binary compatible with it, which means that CUDA programs do not need to be modified for using rCUDA. Furthermore, it implements the entire CUDA Runtime API (except for graphics functions). rCUDA also provides support for the CUDA Driver API. Additionally, is also supports the libraries included within CUDA, such as cuFFT, cuBLAS, or cuSPARSE. Moreover, the rCUDA middleware allows a single rCUDA server to concurrently deal with several remote clients that simultaneously request GPU services. This is achieved by creating independent GPU contexts, each of them being assigned to a different client [20]. These independent GPU contexts also provide robustness against the failure of one the clients.

rCUDA additionally provides specific support for different interconnects [20]. Support for different underlying network fabrics is achieved by making use of a set of runtime-loadable, network-specific communication modules, which have been specifically implemented and tuned in order to obtain as much performance as possible from the underlying interconnect. Currently, three modules are available: one intended for TCP/IP compatible networks, another one specifically designed for InfiniBand, and a third one intended for RoCE networks.

Regarding the InfiniBand and RoCE communications modules, they are based on the InfiniBand Verbs (IBV) API. This API offers two communication mechanisms: the channel semantics and the memory semantics. The former refers to the standard send/receive operations typically available in any networking library, while the latter offers RDMA operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy transfers with minimum involvement of the CPUs. rCUDA employs both IBV mechanisms, selecting one or the other depending on the exact task to be carried out [20].

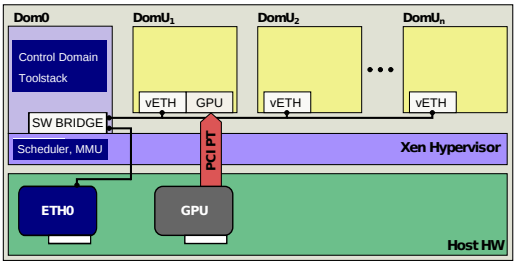


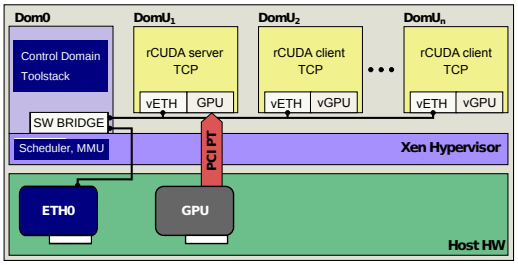
FIGURE 2.3: Typical configuration of a Xen-based system showing how the Ethernet adapter and the GPU available in the host are provided to VMs. The GPU is exclusively assigned to a single VM by making use of the PCI passthrough mechanism. Network connectivity among VMs and between VMs and the external network is provided by means of a software bridge that connects the internal virtual network to the real Ethernet adapter.

Moreover, regardless of the exact network used, data exchange between rCUDA clients and GPUs managed by rCUDA servers is pipelined so that higher bandwidth is achieved, as explained in [21]. Internal pipeline buffers within rCUDA use pre-allocated pinned memory given the higher throughput of this type of memory.

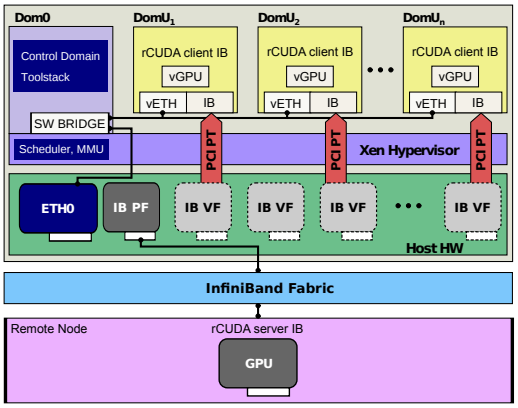
Finally, notice that previous works such as [21, 35] have already measured the bandwidth attained by rCUDA. However, the results presented in this paper are new, mainly because the context of the performance measurements is different. In previous works the focus was on assessing the performance of rCUDA on native domains, whereas in this work the focus is on stating the throughput of rCUDA on Xen VMs. Notice also that the analysis presented in Section 2.7 regarding real applications is, again, focused on the use of rCUDA within Xen VMs, what was not previously studied in other works. Finally, notice that although in this paper we focus on analyzing the effect of using rCUDA in Xen VMs, rCUDA can also be used with other hypervisors. For instance, in [38] the rCUDA middleware was used in the context of the KVM hypervisor. Other VM environments such as VMware or VirtualBox could also be leveraged.

2.4 Testbeds Used in The Experiments

In this work we consider several scenarios in order to provide Xen VMs with access to CUDA accelerators by using the rCUDA middleware. Figure 2.3 depicts a typical Xen configuration, showing a computer hosting several VMs. It can be seen in the figure that the host hardware comprises, among other devices, an Ethernet network adapter and a



(a) Testbed using the virtual network within Xen.



(b) Testbed using InfiniBand to access a remote GPU.

FIGURE 2.4: Testbeds used in the experiments presented in this paper, which make use of rCUDA to provide GPU access to VMs. **(a)** In a single-node testbed, VMs employ the virtual network to access the rCUDA server by means of the TCP/IP protocol stack. **(b)** When an InfiniBand fabric is available, VMs use such interconnect to access a remote rCUDA server.

GPU. On top of the hardware, a thin software layer (the Xen hypervisor) is installed. Above the hypervisor we can find the VMs (Dom0 and DomU$_i$). Notice that the Dom0 VM is a predefined VM using the Xen Linux kernel and behaves as the configuration and management interface to the hypervisor. The rest of VMs (from DomU$_1$ to DomU$_n$) are unprivileged VMs that can be provided to users. Figure 2.3 shows how the Ethernet adapter and the GPU are provided to VMs. On the one hand, the Ethernet adapter is owned by the Dom0 VM, which provides connectivity to the rest of VMs by using a software Ethernet bridge, thus creating a virtual network among the VMs. On the other hand, the GPU is assigned, in an exclusive way, to one of the VMs by making use of the PCI passthrough (PT) mechanism. In this manner, this VM is the only one that may access the GPU, as mentioned in Section 2.2. It is noteworthy the small flexibility that this configuration provides regarding the use of GPUs, given that only one of the VMs can access the GPU.

Once revisited the typical configuration of a Xen-based system, we can describe the testbeds used in the experiments presented in this paper. Notice that we are considering the use of the rCUDA remote GPU virtualization solution in two different scenarios: one where VMs access a GPU located at the same host executing the VMs and another one where the InfiniBand fabric is already present in the cluster and therefore VMs access a GPU installed in another cluster node by making use of this high performance interconnect. Figure 2.4(a) depicts the first scenario whereas Figure 2.4(b) presents the second one.

In the first scenario, one of the VMs will have exclusive access to the GPU by making use of the PCI passthrough mechanism. This VM will grant GPU access to the other VMs by using the rCUDA middleware: the rCUDA server will be executed in the VM owning the GPU whereas the other VMs will use the rCUDA client to access the GPU across the virtual Xen network. TCP/IP based communications will be used in this scenario to communicate the rCUDA clients with the rCUDA server. Accordingly, VMs running the rCUDA client will have one or several virtual instances (vGPU) of the real GPU, which is physically connected to the VM DomU_1. Moreover, the VM DomU_1 will be able to use either the real GPU or its virtual instances. Finally, notice that the real GPU can only be assigned to a DomU_i VM because NVIDIA does not provide support for the Xen Linux kernel used in the Dom0 VM.

Regarding the second scenario, shown in Figure 2.4(b), which uses the InfiniBand fabric already present in the cluster to access a GPU in another node, the firmware in the InfiniBand adapter has been changed, according to the directions in Mellanox User's Guide [39], in order to provide several virtual instances (virtual functions, VF) of the InfiniBand adapter, in addition to the real instance (physical function, PF). Each of these virtual functions will be provided, in an exclusive way, to a Xen VM by using the PCI passthrough mechanism. Moreover, given that an InfiniBand network is available, communication between the rCUDA clients in the VMs and the remote rCUDA server will be based on the use of the high performance InfiniBand Verbs API. Notice that in all the experiments involving the InfiniBand fabric, the remote GPU server is executed in a remote computer which has not been virtualized and also whose InfiniBand network adapter makes use of the original firmware which does not provide virtualization features. Similarly to the scenario shown in Figure 2.4(a), VMs will have one or several virtual

instances of the real GPU, which is physically located in the remote node. Finally, it is important to remark that, although in this work we only consider sharing a single GPU, the rCUDA middleware also allows sharing multiple GPUs.

In addition to the two scenarios depicted in Figure 2.4, a third scenario that could also be considered would consist of a remote rCUDA server accessed through the 1Gbps Ethernet network usually available in the cluster instead of leveraging the InfiniBand interconnect. Notice, however, that although this configuration is also valid, and VMs would have access to GPUs, the low performance of the 1Gbps Ethernet network would significantly increase the execution time of applications being executed inside VMs. Actually, as shown in [40], the performance of applications using remote GPUs across the 1Gbps Ethernet interconnect is noticeably reduced with respect to the use of a local GPU with CUDA. Therefore, in this work we will not consider this third scenario.

The testbed used in this paper to explore the use of the remote GPU virtualization mechanism inside Xen VMs is composed of three 1027GR-TRF Supermicro nodes. One of them will host the Xen VMs whereas the other two nodes will not make use of VMs. In one of the native domains we will execute the rCUDA server as shown in Figure 2.4(b) and the other native domain will be used for several comparison purposes. Each of the servers includes two Intel Xeon E5-2620 v2 processors (six cores with Ivy Bridge architecture) operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1600 MHz. They also have a Mellanox ConnectX-3 VPI single-port InfiniBand adapter connected to a Mellanox Switch SX6025 (InfiniBand FDR compatible) to exchange data at a maximum rate of 56 Gb/s. Furthermore, an NVIDIA Tesla K20 GPU is installed at each node.

Regarding the software configuration, SUSE Linux Enterprise Server 11 SP3 (x86_64) was used in the three servers, with kernel version 3.0.76-0.11. Additionally, in the node hosting the VMs, Xen version 4.2.2 was used. The same kernel version was used in the Dom0 and all the DomU domains, although for Dom0 the kernel was recompiled in order to activate the Xen options. Moreover, the Mellanox OFED 2.3-1.0.1 (InfiniBand drivers and administrative tools) was used, along with CUDA 6.5 and NVIDIA driver 340.29. Finally, VMs were configured to have 4 cores and 12 GB of RAM memory.

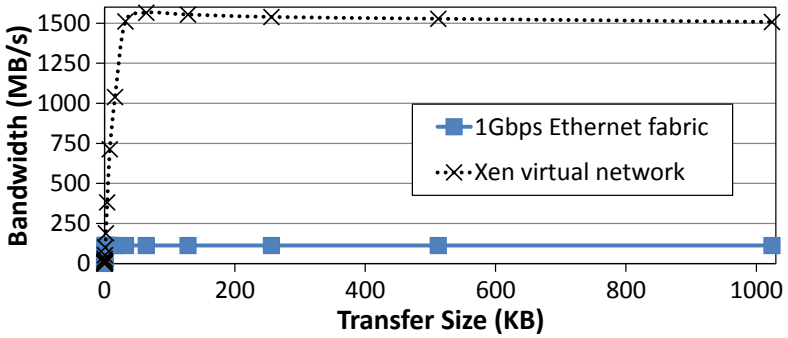


FIGURE 2.5: Bandwidth attained by the virtual network among Xen VMs.

2.5 Network Performance Observed by Xen VMs

When making use of remote GPU virtualization solutions, the bandwidth characteristics of the communication path between main memory, in the client computer, and GPU memory, in the GPU server, greatly influence the performance of data transfers between them. In this section we present the bandwidth numbers achieved by the interconnects used in our study. These results will help us to better understand the behavior of rCUDA within Xen VMs when used in conjunction with GPU-accelerated applications, later analyzed in Sections 2.6 and 2.7.

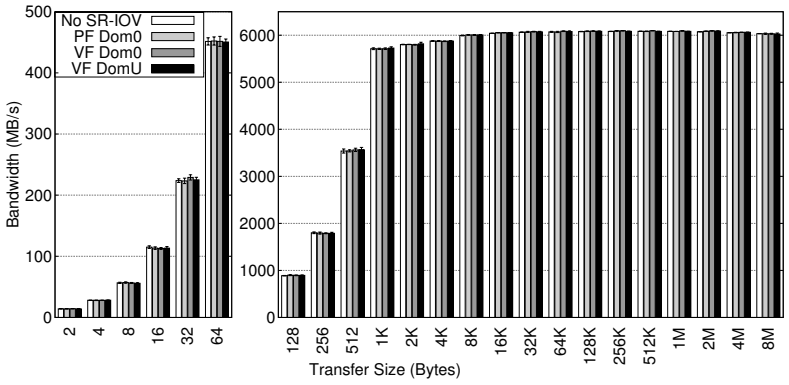
In this paper we consider the two scenarios shown in Figure 2.4. In the first one, see Figure 2.4(a), the virtual network among VMs is used to exchange data among rCUDA clients and servers using TCP/IP. We have analyzed the performance of this network by using the `iperf` tool [34]. Figure 2.5 shows the bandwidth attained by this network as transfer size increases. The figure also includes, for comparison purposes, the performance of the widely available 1 Gbps Ethernet when used from the inside of a VM. In this case, the virtualization features included in the Xen framework have been leveraged in order to provide the Ethernet adapter to the VM. It can be seen in the figure that the virtual network provides much higher bandwidth than the Ethernet one, achieving even higher bandwidth than the 10Gbps Ethernet. In this regard, notice that starting from transfer sizes equal to 32KB, the performance of the virtual network almost reaches 1600 MB/s.

With respect to the second scenario, shown in Figure 2.4(b), a wider analysis is required, given the different possibilities that the use of the InfiniBand cards brings in this context. In this scenario, InfiniBand Verbs are used over virtual instances of the InfiniBand

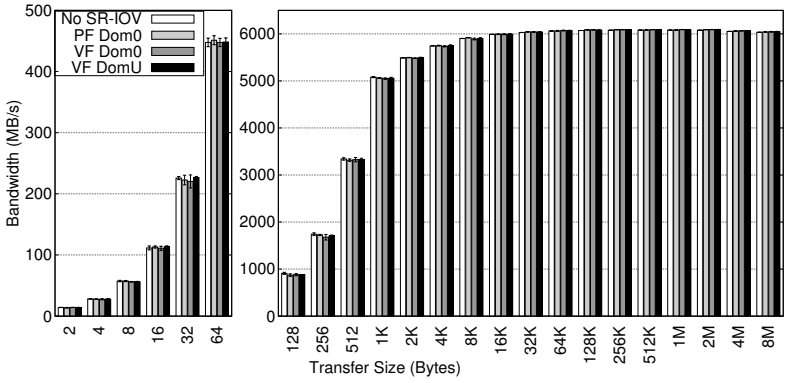
network card in order to communicate the rCUDA client and server. For this reason, the performance of the virtualized InfiniBand network card is next compared to the performance of the non-virtualized one. The `ib_read_bw`, `ib_write_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED software distribution were used in order to mimic the use that the rCUDA framework makes of the InfiniBand fabric [21]. In this regard, the `ib_read_bw` and `ib_write_bw` tests use the memory semantics (i.e., RDMA read and RDMA write, respectively), whereas the `ib_send_bw` test makes use of the channel semantics (i.e., send/receive).

Figure 2.6 shows the bandwidth achieved by the InfiniBand card for different transfer sizes in the scenario depicted in Figure 2.4(b). The behavior of the memory semantics (RDMA) is shown in Figure 2.6(a), where only results for the RDMA write case are presented, given that the RDMA read benchmark provided very similar performance. Figure 2.6(b) shows the channel semantics bandwidth (non-RDMA) when using the `ib_send_bw` benchmark. For comparison purposes, in the experiments carried out with this scenario, we have also considered the performance attained when the tests are executed in the Dom0 VM using both the physical function of the InfiniBand adapter, labeled as *"PF Dom0"*, and also one of the virtual functions, labeled as *"VF Dom0"*. In a similar way, results labeled as *"VF DomU"* refer to the use of a virtual function of the InfiniBand adapter card from the inside of a regular DomU VM. Finally, results labeled as *"No SR-IOV"* have been included for comparison purposes and refer to the use of a non-virtualized InfiniBand card from a native domain. Notice that the bars shown in the figure represent the average of 10 executions of the bandwidth tests configured to perform 20,000 repetitions at each run. Furthermore, this information is complemented, for each transfer size, with the 95% confidence intervals (although these intervals are quite small and can be only distinguished for some of the transfer sizes).

As we can see in Figure 2.6, the shapes of the bandwidth attained in all the cases under study are, in general, quite similar. Therefore, we can conclude that both the verbs using channel semantics (non-RDMA) and the ones using memory semantics (RDMA) provide similar bandwidth regardless of whether the network card is virtualized or not and also regardless of whether the network card is used from a Dom0 VM or from a DomU_i VM.



(a) InfiniBand RDMA write bandwidth.



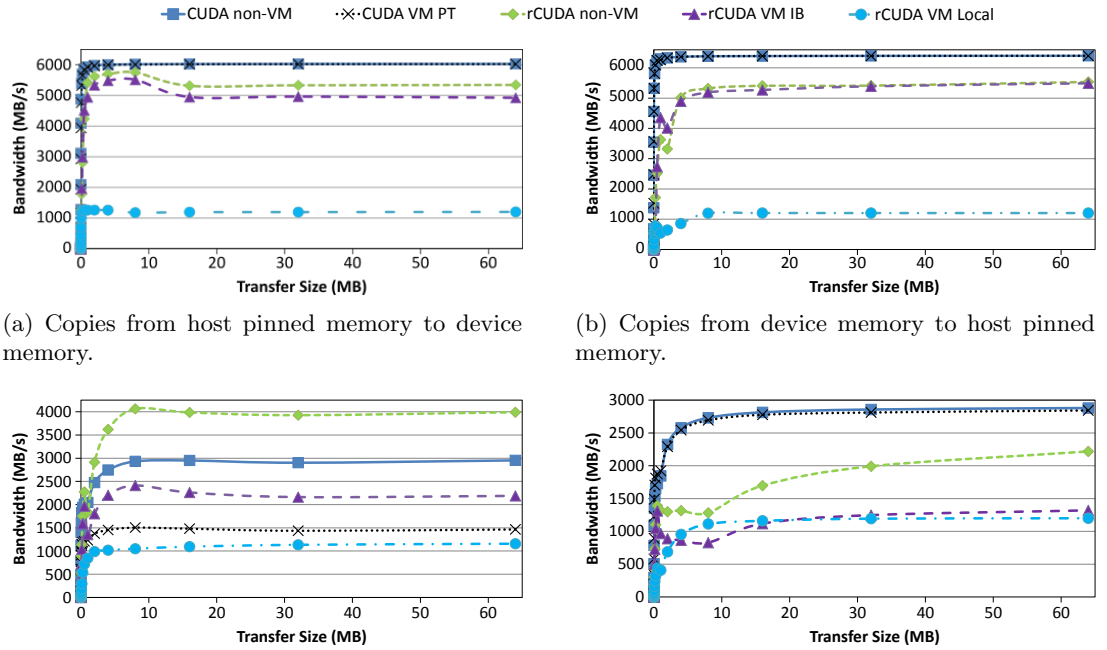
(b) InfiniBand send bandwidth.

FIGURE 2.6: InfiniBand bandwidth tests using ConnectX-3 network cards executed in the different scenarios under study.

2.6 Performance of rCUDA within Xen VMs

In this section we explore the performance of the rCUDA middleware when used in the context of Xen VMs.

We employ the performance of CUDA as the baseline reference in this analysis, since minimizing the overhead with respect to the performance of CUDA is the goal of any remote GPU virtualization solution. Therefore, we will make use of the `bandwidthTest` benchmark from the NVIDIA CUDA Samples [33] to transfer data from main memory in the client VM to the Tesla K20 GPU (located either in other VM or in a remote real server). In order to use the proper hardware configuration for the baseline CUDA reference, we made use of a configuration which uses the GPU local to the node executing the benchmark, in the traditional way and within a native domain (no Xen VM). Results for this case are referred to as *"CUDA non-VM"* in Figure 2.7. In a similar



(a) Copies from host pinned memory to device memory.

(b) Copies from device memory to host pinned memory.

(c) Copies from host pageable memory to device memory.

(d) Copies from device memory to host pageable memory.

FIGURE 2.7: Bandwidth tests for copies between host and device memory, using CUDA and the rCUDA middleware. Tests have been carried out in the different scenarios depicted in Figure 2.4 as well as in native domains.

way, when CUDA is used in DomU_1 in the scenario depicted in Figure 2.3 by using the PCI passthrough mechanism, the label *"CUDA VM PT"* is used. Regarding the performance of rCUDA, the label *"rCUDA non-VM"* refers to the performance of the rCUDA middleware when used between native domains (no Xen VM involved) making use of the InfiniBand network. These curves are included for comparison purposes. When Xen VMs are involved in the tests, the label *"rCUDA VM IB"* refers to the performance of rCUDA when used in the scenario shown in Figure 2.4(b). Finally, the performance of rCUDA in the scenario depicted in Figure 2.4(a) is denoted by the label *"rCUDA VM Local"*.

Figure 2.7 presents bandwidth results for copies in the host-to-device[6] direction and also for the opposite direction, using both pinned and pageable host memory. Results in Figure 2.7 are the average from 10 executions of the CUDA `bandwidthTest` test configured to perform 1000 repetitions at each execution. The 95% confidence intervals

[6]In this work, we will refer to main memory as *host memory* or just *host*, while GPU memory will be referred as *device memory* or simply *device*, according to the well-established usage defined in the CUDA ecosystem.

were also computed, although in this case the variability is very small and thus the value of these intervals is, in average, 1.95%, what suggested not to include them in the figures given that these small confidence intervals were going to be hardly visible.

The bandwidth results for pinned memory, presented in Figures 2.7(a) and 2.7(b), show that the bandwidth attained for CUDA copies in the native domain and in the Xen VM using PCI passthrough present almost the same performance. In the case of rCUDA using InfiniBand to communicate with a remote GPU server, a slightly smaller bandwidth is achieved. Finally, when rCUDA is used employing the virtual network among VMs, maximum bandwidth for the CUDA memory copies is slightly lower than the one obtained when using the `iperf` tool, shown in previous section.

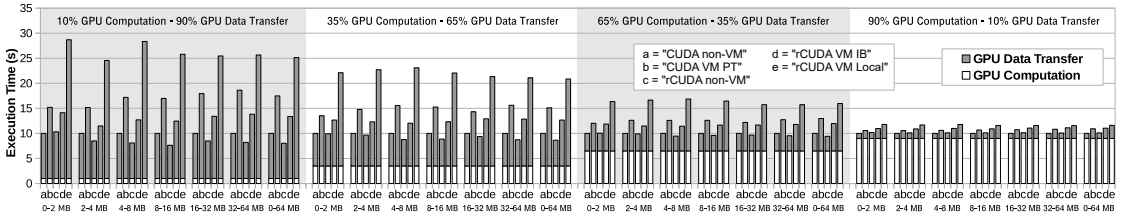
Regarding the use of pageable memory, it can be seen in Figures 2.7(c) and 2.7(d) that in the case of copies from host memory to device memory, there is an important difference between the performance achieved by CUDA in the native domain and that obtained in the Xen VM using the PCI passthrough mechanism, since performance in the former doubles the bandwidth in the latter. Nevertheless, this effect does not appear in the opposite direction (Figure 2.7(d)), where both cases present almost the same performance. Regarding the use of rCUDA when the InfiniBand network is leveraged, the ratio between the performance obtained in the native domain and that in the VM follows the same trend as for CUDA: the native domain attains twice the performance achieved in the VM. With respect to the performance of rCUDA when the virtual network is used along with TCP/IP based communications, Figure 2.7(c) shows that this scenario achieves the lowest bandwidth, as it was expected from the results shown in Figure 2.5. On the other hand, when the device-to-host direction is considered, results are quite different. First, the performance of the baseline CUDA and that of CUDA when used within a VM with PCI passthrough are very similar. Second, the performance of rCUDA in the native and virtualized domains follow the same trend as for the host-to-device direction, but now performance is noticeably reduced. Third, the bandwidth results of rCUDA when the virtual network is used are similar to the performance achieved in the opposite direction.

In summary, we can conclude that the bandwidth attained by PCI passthrough is almost identical to the one achieved by CUDA, except for copies from host pageable memory to device memory, where the bandwidth is reduced to the half. On the other hand, rCUDA

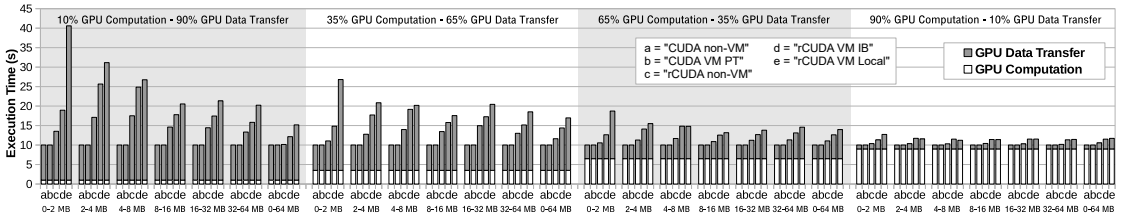
over the Xen virtual network results in a very stable behavior in all the scenarios, the bandwidth being limited by the network performance (see Figure 2.5). Finally, the bandwidth obtained by rCUDA over an InifiniBand network is very close to that of CUDA when using pinned host memory, regardless of whether accessing the remote GPU from a VM or from a native domain. In the case of pageable host memory, the bandwidth when accessing the GPU from a VM is reduced to the half of the one obtained by rCUDA without using VM. This reduction in the performance when involving the VM is more evident in the case of copies from device memory to host memory, where the bandwidth obtained by rCUDA using the VM is the same, regardless of using the Xen virtual network or the InfiniBand one.

Next we analyze the effect of using rCUDA within Xen VMs by making use of a synthetic application. The purpose of using a synthetic application is to be able to modulate the amount of data transferred between host and device as well as controlling the amount of computations carried out in the GPU. In this way, it is possible to analyze the effect of rCUDA on application performance when the application features different percentages of communications and computations. For instance, it is possible to mimic communication intensive applications by setting the amount of data transfers to last 90% of the application execution time while keeping computations to only 10% of execution time. On the contrary, it is feasible to model compute intensive applications by setting the percentage of time devoted by the application to data transfers to only 10% whereas setting the percentage of time used for computations to 90% of execution time. An intermediate case where 35% of the execution time is devoted to computations in the GPU whereas 65% of the execution time is used for data transfers is also possible. The opposite case, with 65% of execution time used for computations and 35% of execution time employed in data transfers would complete a thorough analysis with such a synthetic application.

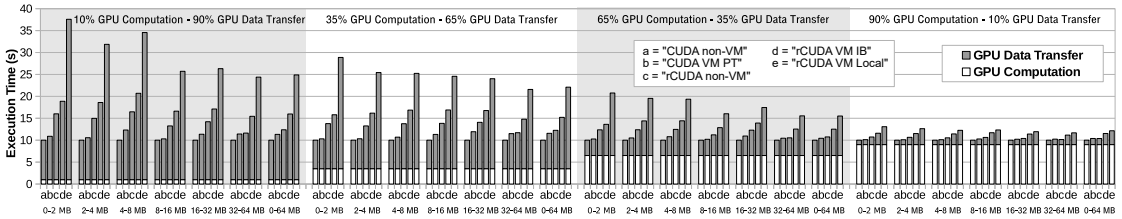
Figure 2.8 shows the performance results when the synthetic application is used with several computation and transfer percentages in the same scenarios previously described for Figure 2.7 (namely, *"CUDA non-VM"*, *"CUDA VM PT"*, *"rCUDA non-VM"*, *"rCUDA VM IB"*, and *"rCUDA VM Local"*). Figure 2.8(a) depicts the performance results when the data transfers performed by the application follow the host-to-device direction. Figure 2.8(b) shows the results when data transfers are carried out in the opposite direction.



(a) Data transfers are carried out from host pageable memory to device memory.



(b) Data transfers are carried out from device memory to host pageable memory.



(c) Data transfers are carried out in both directions: host to device and device to host using host pageable memory.

FIGURE 2.8: Performance of a synthetic application where the percentage of execution time devoted to data transfers to/from the GPU and the percentage of execution time used for computations in the GPU are set by the user. Notice that these percentages are initially established for the executions using CUDA with a local GPU (case "a") by defining the amount of data to be transferred. For the rest of scenarios, this initial amount of data to be transferred is kept constant, thus producing a deviation of the initial percentages. Furthermore, for each size interval, the exact size of data transfers is randomly set.

Figure 2.8(c) presents results when data transfers in both directions are used. Furthermore, results in Figures 2.8(a), 2.8(b), and 2.8(c) take into account the size of the data transfer, given that, as shown in Figure 2.7, attained bandwidth depends on the exact data transfer size. This is why different size intervals are used for each figure. Each of the size intervals shown in the figures (each of the bars) is the average of 5 repetitions. Each of the repetitions makes use of a randomly chosen data transfer size. Finally, notice that the percentages of application execution time devoted to data transfers and GPU computations are initially set for the native CUDA scenario using a local GPU (case "a" in the figures). This is achieved by setting the amount of data to be transferred to/from the GPU. In the rest of scenarios, that very same amount of data is transferred.

However, given that the underlying communication channel to/from the GPU is different, a deviation from the initial percentages is produced. This deviation will allow us to determine the overhead introduced by each scenario.

It can be seen in Figures 2.8(a), 2.8(b), and 2.8(c) that the overhead of rCUDA greatly depends on the percentage of data transfers carried out during the execution of the application. In this regard, the overhead introduced by rCUDA when the application devotes 90% of its execution time to data transfers is much higher than when only 10% of the execution time is devoted to data transfers. Additionally, as transfer sizes become larger, the overhead introduced by rCUDA is reduced. This result is consistent with the results shown in Figure 2.7. In a similar way, Figure 2.8(a) shows the effect of having less bandwidth in the *"CUDA VM PT"* scenario than in the *"CUDA non-VM"* configuration, as already pointed out in Figure 2.7(c). Another interesting remark about Figure 2.8(a) is that application performance is better for the *"rCUDA non-VM"* case than for the *"CUDA non-VM"* scenario. The reason is that rCUDA achieves higher bandwidth than CUDA in the native domains, as already shown in Figure 2.7(c). Finally, the lower bandwidth of rCUDA for the device-to-host data copies, shown in Figure 2.7(d), is also visible in Figure 2.8(b).

2.7 Impact of Xen VMs on Real Applications

In previous sections we have studied how the performance of the Xen virtual network and that of the InfiniBand ConnectX-3 network cards, when used from the inside of Xen VMs, influence the performance of the rCUDA remote GPU virtualization middleware. In order to do so, we used synthetic benchmarks that allowed us to focus on specific characteristics of the virtualization solution. In this section we study how the performance of these interconnects, along with the use of Xen VMs, influence the execution time of real applications. Remember that this is the actual goal of our work: to explore the use of the remote GPU virtualization mechanism in order to provide CUDA acceleration to applications running inside Xen VMs, and characterizing such exploration by using as a metric the overhead that applications experience when accessing GPUs outside their Xen VM by using a remote GPU virtualization framework. We consider two different types of applications in this section: those making use of a single GPU and those that

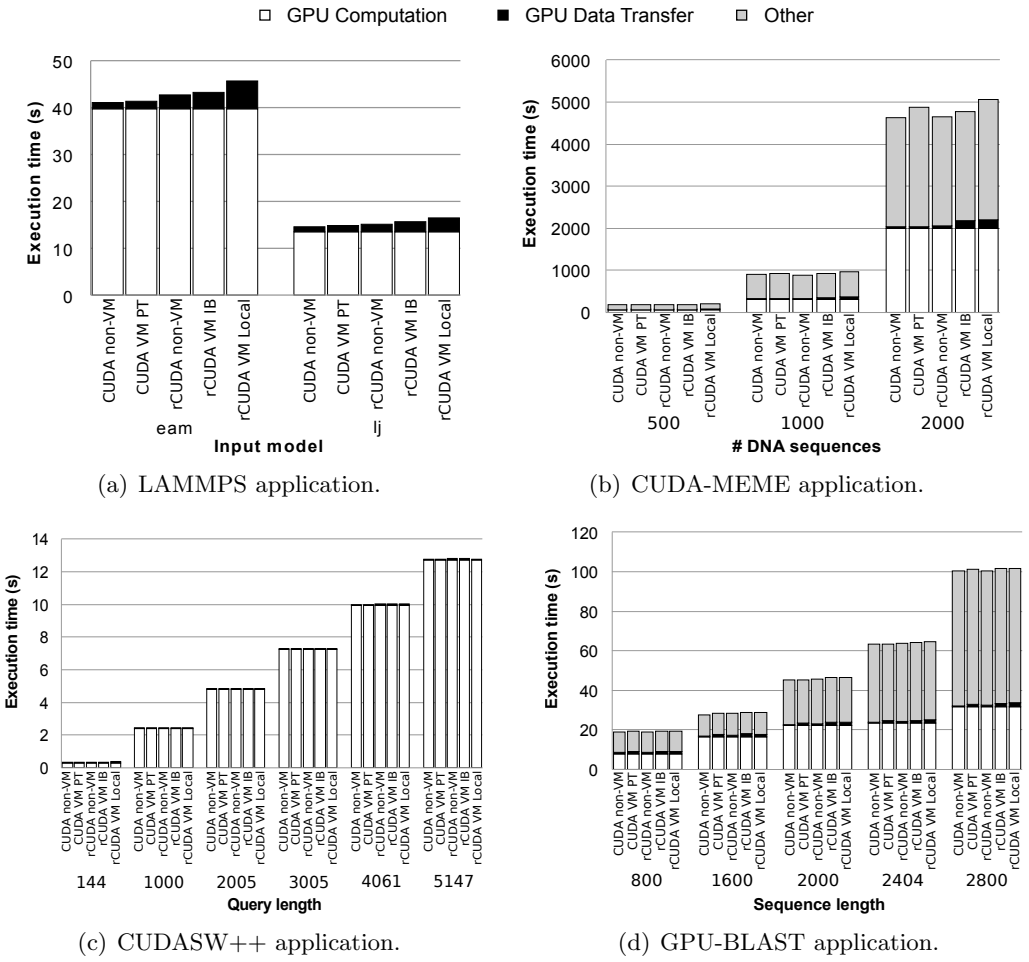
offload computations to more than one GPU. In next section we analyze the first kind of applications. In Section 2.7.2 we will present performance results for the second type of applications.

2.7.1 Applications Using One GPU

The applications analyzed in this section are LAMMPS [41], CUDA-MEME [42], CUDASW++ [43], and GPU-BLAST [44], listed in the NVIDIA GPU-Accelerated Applications Catalog [45]:

- LAMMPS is a molecular dynamics simulator that can be used as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For our tests we use the release from Dec. 9, 2014, and benchmarks `in.eam` and `in.lj`, with a factor scale of 5 in all three dimensions.
- CUDA-MEME is a parallel formulation and implementation of the MEME motif discovery algorithm using the CUDA programming model. In particular, we have used its latest release, version 3.0.15, for our study, along with the test cases available in the application website [46].
- CUDASW++ is a bioinformatics software for Smith-Waterman protein database searches that takes advantage of the massively parallel CUDA architecture of NVIDIA Tesla to perform sequence searches. In particular, we have used its latest release, version 3.1, with the latest Swiss-Prot database and the example query sequences available in the application's website.
- GPU-BLAST has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST implementation using GPUs. It is integrated into the NCBI-BLAST code and produces identical results. We use the release 1.1 in our experiments, where we have followed the installation instructions for sorting a database and creating a GPU database. We then use the query sequences that come with the application package to search the database.

Figure 2.9 shows the execution time of these four applications when executed in the same scenarios as in the previous section: execution with CUDA with a local GPU in



(a) LAMMPS application.

(b) CUDA-MEME application.

(c) CUDASW++ application.

(d) GPU-BLAST application.

FIGURE 2.9: Execution time of several applications when executed in different local and remote scenarios. Execution time is broken down into three components: GPU computation, GPU data transfer, and Other.

a native domain (*"CUDA non-VM"*) and within a Xen VM accessing the GPU in the host by making use of PCI passthrough (*"CUDA VM PT"*). In the case of rCUDA, the three scenarios considered (*"rCUDA non-VM"*, *"rCUDA VM IB"*, and *"rCUDA VM Local"*) refer to the ones already described in the previous section. Every experiment has been performed 10 times, so that the figures show the averaged results. Furthermore, the 95% confidence intervals were computed, but they are so small that their inclusion in the figures provided no additional important information. In addition to execution time, the plots in the figure also include a breakdown of the execution time, which is split into three different components: (1) time required to transfer data to/from the GPU (*"GPU Data Transfer"*), (2) time spent carrying out computations in the GPU

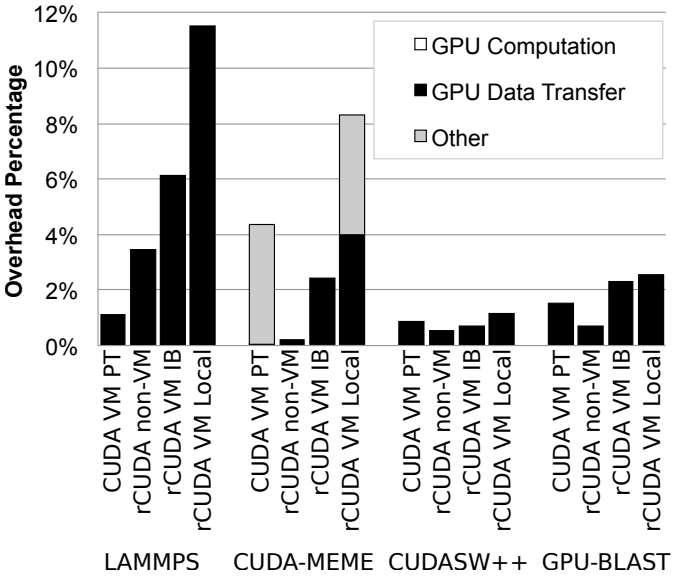


FIGURE 2.10: Average overhead with respect to executions with CUDA in a native domain for the four applications depicted in Figure 2.9.

(*"GPU Computation"*), and (3) time spent in tasks not involving the GPU, such as CPU computations and I/O (*"Other"*).

Execution times presented in Figure 2.9 show that the four applications have a similar behavior, spending a very small portion of time for transferring data to the GPU, and spending the rest of the time performing computations either in the CPU or in the GPU. More specifically, in the case of GPU-BLAST and CUDA-MEME applications, they present periods of time in which the GPU is not used. On the contrary, both LAMMPS and CUDASW++ keep the GPU busy for almost all the execution time.

Figure 2.10 shows the average overhead with respect to executions with CUDA in a native domain for the four applications. This figure shows that rCUDA overhead in LAMMPS, CUDASW++ and GPU-BLAST applications is mainly due to data transfers between main memory and GPU memory. Additionally to the overhead of transfers, the CUDA-MEME application also presents a performance decrease when using a VM that makes use of the PCI passthrough technique. As we can see, this additional overhead is not due to the increase of GPU data transfer time, but to the time spent in other tasks by the PCI passthrough technique (referred to as *"Other"* in the figure), which are out of the scope of this paper.

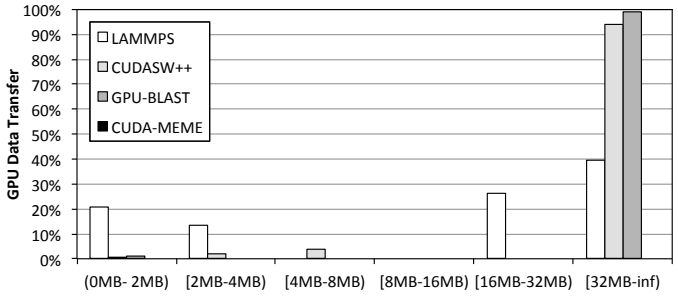
In general, the overhead of rCUDA is mainly due to data transfers between main memory and GPU memory. This was expected because once data is in the GPU memory, GPU computations require the same amount of time to be completed as in a native environment. In average, in our experiments, the overhead of running GPU-accelerated applications in a Xen VM with respect to a native domain is 2%, 2.8%, and 5.8% when using PCI passthrough, rCUDA over an InfiniBand fabric, and rCUDA over the Xen virtual network, respectively.

Once the main cause of the overhead has been studied, a deeper analysis is necessary to characterize the behavior of each application. In this regard, as shown in Figure 2.7, time required for data transfers varies depending on the copy direction (to or from GPU memory) and the memory type (pageable or pinned memory). In order to analyze the influence of these different transfer bandwidths on application execution time, Table 2.1 presents the total amount of data transferred in each direction, as well as the memory type. As we can observe, none of the applications uses pinned memory. Additionally, given that bandwidth attained for data copies also depends on transfer size, Figure 2.11 depicts how the total amount of transferred data shown in Table 2.1 is split into different message sizes in order to be actually transferred. Putting all this information together, Table 2.1 shows that CUDASW++ and GPU-BLAST mainly copy data from main memory to GPU memory, more than 90% of these transfers being done with messages greater than 32MB, as depicted in Figure 2.11(a). On the other hand, according to Table 2.1, the majority of the transfers in the CUDA-MEME application are from GPU memory to main memory, and almost 90% of the transferred data is copied in message sizes between 4 and 16MB, as shown in Figure 2.11(b). Finally, the LAMMPS application presents similar percentage of transfers in both copy directions, with 80% of data transferred in messages of size larger than 2MB.

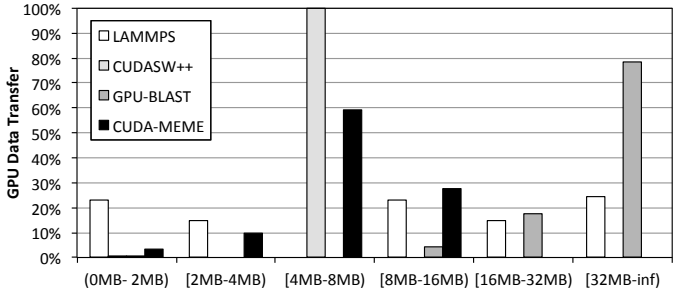
With the data gathered in this analysis, we can complete our study and conclude that

Applications	**HtoD pageable**		**DtoH Pageable**	
	GB	**%**	**GB**	**%**
LAMMPS	3	59	2	41
CUDASW++	0.195	98	0.004	2
GPU-BLAST	1.3	79	0.356	21
CUDA-MEME	0.048	0	100	100

TABLE 2.1: Data transfers in the applications under analysis



(a) Copies from host pageable memory to device memory.



(b) Copies form device memory to host pageable memory.

FIGURE 2.11: Histograms showing the percentage of transferred data according to message size.

when comparing the overhead of PCI passthrough and rCUDA (used from the inside of a VM), their behavior with respect to each other mainly depends on the type of data transfers they mostly perform (pageable host-to-device or pageable device-to-host), which present a very different performance as shown in Figures 2.7(c) and 2.7(d). In this regard, for applications in which copies from host to device have a bigger weight, PCI passthrough performs worse. On the contrary, for applications that mainly transfer data from device to host, then rCUDA performs worse. That is, there is a direct dependency of application performance on the bandwidth attained for each copy direction. This result points out the impact on application performance of the bandwidth attained by the underlying network connecting main memory and GPU memory.

Finally, notice that current cloud computing providers use the PCI passthrough mechanism to provide applications with CUDA acceleration. However, the average overheads shown in Figure 2.10 are computed with respect to executions with CUDA in a native domain. Therefore, in order to provide the right perspective, it is advisable to use as the baseline reference the performance of applications when using the PCI passthrough from the inside of a VM instead. In this regard, Figure 2.12 shows the average overhead experienced by applications when using the rCUDA middleware using as the baseline

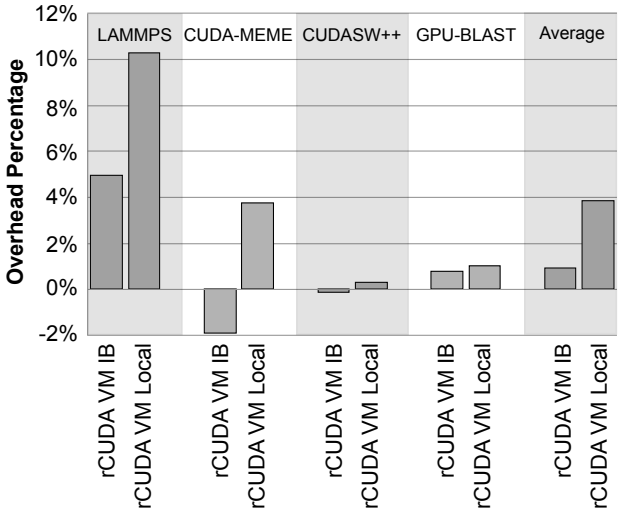


FIGURE 2.12: Average overhead experienced by applications with respect to executions with CUDA using the PCI passthrough from the inside of a VM.

reference their performance when executed using the PCI passthrough mechanism in order to access the GPUs. As can be seen the overhead is very low when an InfiniBand network is available. In the cases of CUDA-MEME and CUDASW++ the execution time is even lower than the obtained with the PCI passthrough mechanism. In the case that rCUDA is used through the virtual network, we can see that the overhead increases with respect to the previous scenario but this overhead remains low, less than 4% on average.

2.7.2 Applications Using Multiple GPUs

In the previous section we have presented an analysis of four different applications that offloaded their computations to one GPU. However, there are applications that can make use of several GPUs in order to further reduce their execution time. In this section we present performance results for applications using two GPUs.

Several system configurations can be used when several GPUs are leveraged in the context of Xen VMs and rCUDA. Figures 2.13 and 2.14 show four of these configurations. Figure 2.13 depicts the simplest one, where a Xen VM is assigned two GPUs by making use of the PCI passthrough mechanism and the GPUs are accessed by means of CUDA. This configuration is similar to that depicted in Figure 2.3 although two GPUs are used now. On the other hand, Figure 2.14 shows the configurations when rCUDA is used

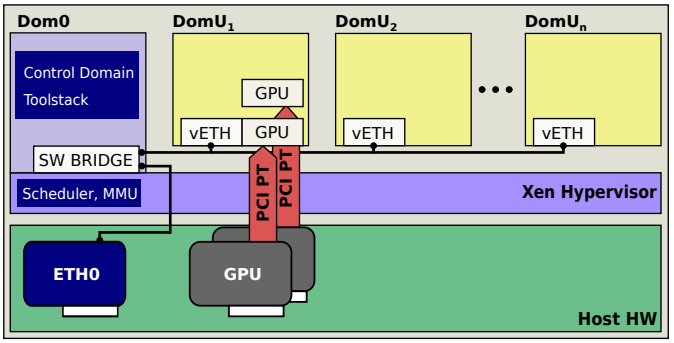
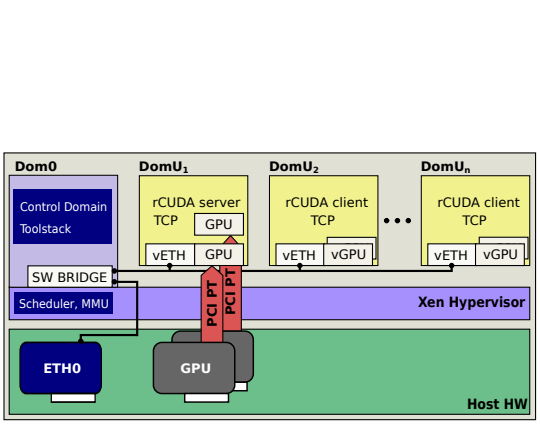


FIGURE 2.13: Configuration of a Xen-based system showing two GPUs assigned to one of the VMs. The GPU assignment is carried out by making use of the PCI passthrough mechanism. Therefore, both GPUs can only be used by the VM owning them.

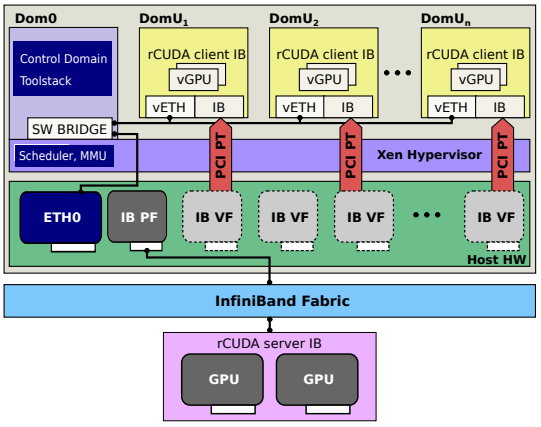
within Xen VMs in order to access two GPUs. Figure 2.14(a) shows the scenario where rCUDA is used to access the two GPUs located at the same host that executes the VMs. This configuration is similar to that presented in Figure 2.4(a) but two GPUs are leveraged now. Notice that the two GPUs are assigned, by making use of the PCI passthrough technique, to one of the VMs, where the rCUDA server is being executed. Furthermore, when the InfiniBand network is present in the cluster, two additional configurations are feasible: (1) both GPUs are located in the same remote node and (2) two remote nodes are used, each with one GPU. Figures 2.14(b) and 2.14(c) depict, respectively, these configurations, which are similar to the one shown in Figure 2.4(b) although two GPUs are used now. Finally, in the experiments carried out in this section, GPUs are also used in native domains. In the case of CUDA, the two GPUs are installed at the node running the application. In the case of rCUDA the two GPUs can be located in one remote node or in two remote nodes. All these scenarios will be considered in the performance tests carried out in this section.

Two applications will be used as test cases in this section: the CUDASW++ application already used in the previous section and the TRICO (TRIangle COunt) application [47]. TRICO is a CUDA implementation of a parallel algorithm for counting triangles (i.e. 3-cycles) in large graphs which additionally is able to take advantage of all the GPUs available in the node where it is being executed.

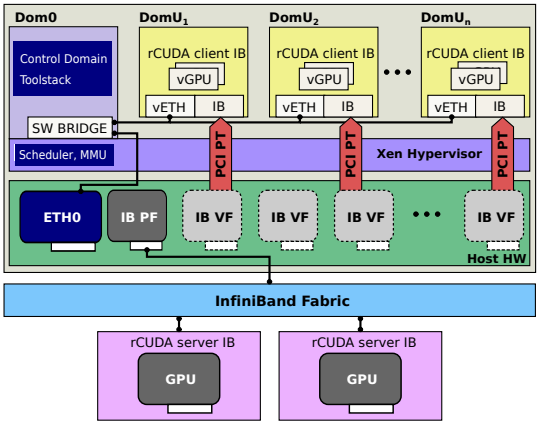
Figure 2.15(a) shows the performance results of the CUDASW++ application when executed using two GPUs. Label *"CUDA non-VM"* refers to the execution with CUDA with two local GPUs in a native domain whereas the case for the application being



(a) Testbed using the virtual network within Xen.



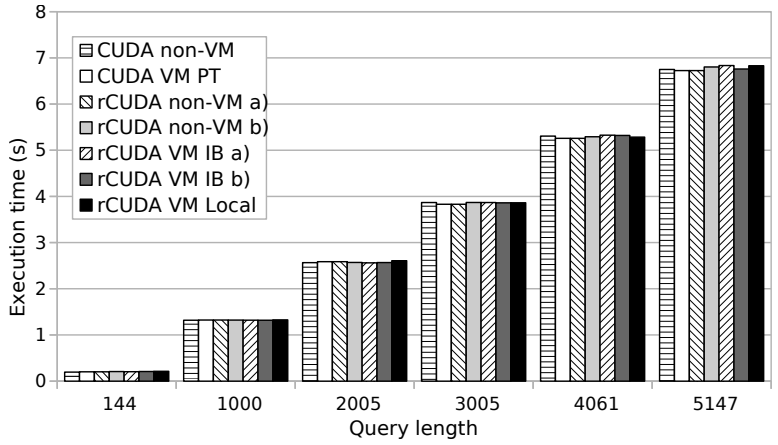
(b) Testbed using the InfiniBand fabric to access two remote GPUs located in the same remote node.



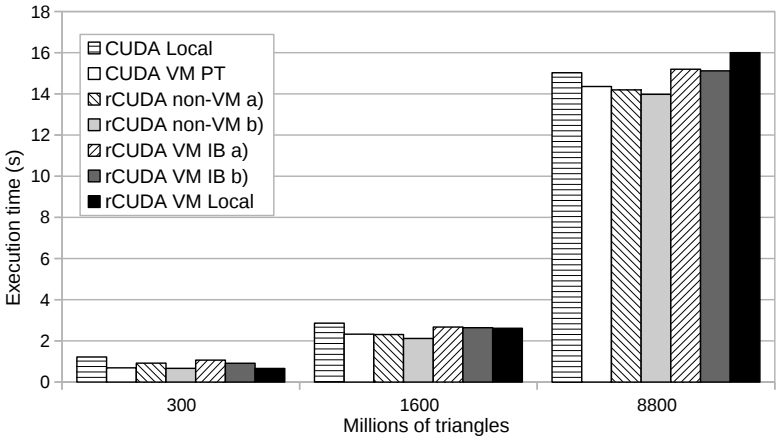
(c) Testbed using the InfiniBand fabric to access two remote GPUs located in two remote nodes.

FIGURE 2.14: Testbeds used with rCUDA. Two GPUs are provided to VMs. (a) In a single-node scenario, VMs use the virtual network (TCP/IP) to access the rCUDA server running in one of the VMs. (b/c) When an InfiniBand fabric is available in the cluster, VMs use such interconnect in order to access the remote GPUs, which can be located either in the same (b) or in different (c) remote nodes.

executed within a Xen VM and accessing the GPUs in the host by making use of PCI passthrough is referred to as *"CUDA VM PT"*. In the case of rCUDA, executions in a native domain (no VM involved) are referred to as *"rCUDA non-VM a)"* when both GPUs are located in the same remote node. When both GPUs are located in different remote nodes, the label *"rCUDA non-VM b)"* is used. In a similar way, when using rCUDA within a Xen VM, label *"rCUDA VM Local"* refers to the scenario depicted in Figure 2.14(a) where the virtual network provided by Xen is used to access the GPUs located at the same host executing the VM. Finally, labels *"rCUDA VM IB a)"* and *"rCUDA VM IB b)"* refer to the scenarios depicted in Figures 2.14(b) and 2.14(c),



(a) CUDASW++ application.



(b) TRICO application.

FIGURE 2.15: Performance of two applications when executed in different local and remote scenarios involving Xen VMs.

respectively, where the InfiniBand fabric is present in the cluster and therefore one or two remote GPU servers are used. It can be seen in Figure 2.15(a) that the performance of rCUDA when two GPUs are used is similar to that of CUDA in all the scenarios considered. On the other hand, Figure 2.15(b) presents the performance results for the TRICO application when two GPUs are used. In this case, a higher variability in the execution time of the application is observed, being the worst case the execution for 8800 millions of triangles with rCUDA when the virtual network provided by Xen is used with TCP/IP (scenario depicted in Figure 2.14(a)).

2.8 Conclusions

In this paper we have analyzed the use of the remote GPU virtualization mechanism in order to provide acceleration services to scientific applications running inside Xen VMs. We have considered two different scenarios: (1) in those clusters not leveraging an InfiniBand interconnect, a VM grants GPU access to the other VMs concurrently running in the same host, and (2) in those clusters were an InfiniBand fabric is already present, VMs access a remote GPU located in another node of the cluster.

First, we have used synthetic benchmarks to characterize the performance attained when using different underlying network fabrics. Afterwards, we have studied the impact on execution time of running scientific applications inside the virtualized domain.

The main conclusion from our exploration is that remote GPU virtualization solutions are a feasible option to provide CUDA acceleration services to Xen VMs. Our experiments showed that the overhead of executing accelerated applications within Xen VMs with respect to currently available approaches (i.e., PCI passthrough) greatly depends on the internals of each application, being negligible (0.92% on average) when the cluster already includes an InfiniBand interconnect and very low (3.84% on average) in the case of using the internal virtual network within Xen.

Nevertheless, overhead percentages are not the only result to keep from this exploration. Another important conclusion is that remote GPU virtualization solutions provide to data center managers the configuration flexibility that Xen currently lacks. In this manner, remote GPU virtualization frameworks not only provide the possibility to concurrently offer GPU acceleration services to several VM instances being executed in the same host, but they also provide the possibility of offering differentiated services to different data center users, given that cluster administrators keep complete control on how GPUs are shared among users. For example, it could be possible to create two groups of users for a given application: a smaller group including those users willing to pay more money in order to achieve higher application performance (i.e., not sharing GPUs) and a bigger group composed of those users preferring to wait some more time for their application to complete execution but at a lower economic cost (i.e., sharing GPUs among VMs).

As for future work, we plan to analyze the effect on application performance of sharing the available GPUs among several VMs. In this regard, it is necessary to develop a scheduler that coordinates the use of GPU memory among the several VMs sharing the GPUs. This scheduler is required in order to ensure that applications do not experience out-of-memory issues due to the fact that several of them are allocating GPU memory at the same time. Migrating GPU jobs [48] will be a useful technique in order to better coordinate the use of GPU memory resources among VMs. Finally, a new communication layer within rCUDA based on the use of shared memory will also be investigated. The purpose of this new shared-memory based communication layer is to avoid using the virtual network provided by the Xen hypervisor, thus attaining higher performance.

Acknowledgments

This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/077. Authors are also grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] Kernel-based Virtual Machine, KVM. `http://www.linux-kvm.org`, 2015. Accessed 19 October 2015.

[2] Xen Project. `http://www.xenproject.org/`, 2015. Accessed 19 October 2015.

[3] VMware virtualization. `http://www.vmware.com/`, 2015. Accessed 19 October 2015.

[4] Oracle VM VirtualBox. `http://www.virtualbox.org/`, 2015. Accessed 19 October 2015.

[5] A.A. Semnanian, J. Pham, B. Englert, and Xiaolong Wu. Virtualization Technology and its Impact on Computer Hardware Architecture. In *Proc. of the Information Technology: New Generations, ITNG*, pages 719–724, 2011.

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IBM Research Report*, 2014.

[7] Jie Zhang, Xiaoyi Lu, M. Arnold, and D.K. Panda. MVAPICH2 over OpenStack with SR-IOV: An Efficient Approach to Build HPC Clouds. In *Proc. of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid*, pages 71–80, 2015.

[8] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proc. of the International Symposium on Code Generation and Optimization, CGO*, 2014.

[9] Daniel P Playne and Kenneth A Hawick. Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA. In *Proc. of the Parallel and Distributed Processing Techniques and Applications, PDPTA*, pages 104–110, 2009.

[10] Ichitaro Yamazaki, Tingxing Dong, Raffaele Solcà, Stanimire Tomov, Jack Dongarra, and Thomas Schulthess. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, 26(16):2652–2666, 2014.

[11] Duraiswami Yuancheng Luo. Canny edge detection on NVIDIA CUDA. In *Proc. of the Computer Vision and Pattern Recognition Workshops, CVPR Workshops*, pages 1–8, 2008.

[12] Vladimir Surkov. Parallel option pricing with Fourier space time-stepping method on graphics processing units. *Parallel Computing*, 36(7):372–380, 2010.

[13] Pratul K. Agarwal, Scott Hampton, Jeffrey Poznanovic, Arvind Ramanthan, Sadaf R. Alam, and Paul S. Crozier. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 25(10):1356–1375, 2013.

[14] Guo-Heng Luo, Sheng-Kai Huang, Yue-Shan Chang, and Shyan-Ming Yuan. A parallel Bees Algorithm implementation on GPU. *Journal of Systems Architecture*, 60(3):271–279, 2014.

[15] NVIDIA GRID Technology. `http://www.nvidia.com/object/grid-technology.html`, 2015. Accessed 19 October 2015.

[16] J. Song and et al. KVMGT: a Full GPU Virtualization Solution. In *KVM Forum*, 2014.

[17] AMD Multiuser GPU, hardware-based virtualized solution. `http://www.amd.com/Documents/Multiuser-GPU-Datasheet.pdf`, 2015. Accessed 19 October 2015.

[18] V-GPU: GPU virtualization. `https://github.com/zillians/platform_manifest_vgpu`, 2015. Accessed 19 October 2015.

[19] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *Proc. of the SC Companion: High Performance Computing, Networking Storage and Analysis, SCC*, pages 1207–1214, 2012.

[20] Carlos Reaño, Federico Silla, and Jose Duato. Enhancing the rCUDA Remote GPU Virtualization Framework: From a Prototype to a Production Solution. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 695–698. IEEE Press, 2017.

[21] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, pages 4:1–4:7. ACM, 2015.

[22] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proc. of the IEEE Parallel and Distributed Processing Symposium, IPDPS*, pages 1–11, 2009.

[23] Tyng Yeu Liang and Yu Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Proc. of the IEEE Advanced Information Networking and Applications Workshops, WAINA*, pages 141–146, 2011.

[24] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proc. of the Euro-Par Parallel Processing, Euro-Par*, pages 379–391, 2010.

[25] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt*, pages 17–24, 2009.

[26] Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. In *Proc. of the International Workshop on Virtualization Technologies in Distributed Computing, VTDC*, pages 3–10, 2011.

[27] Shadowfax II - scalable implementation of GPGPU assemblies. `http://keeneland.gatech.edu/software/keeneland/kidron`, 2015. Accessed 19 October 2015.

[28] John Paul Walters, Andrew J. Younge, Dong-In Kang, Ke-Thia Yao, Mikyung Kang, Stephen P. Crago, and Geoffrey C. Fox. GPU-Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *Proc. of the IEEE International Conference on Cloud Computing, CLOUD*, 2014.

[29] Chao-Tung Yang, Hsien-Yi Wang, Wei-Shen Ou, Yu-Tso Liu, and Ching-Hsien Hsu. On implementation of GPU virtualization using PCI pass-through. In *Proc. of the IEEE Cloud Computing Technology and Science, CloudCom*, pages 711–716, 2012.

[30] Heeseung Jo, Jinkyu Jeong, Myoungho Lee, and Dong Hoon Choi. Exploiting GPUs in Virtual Machine for BioCloud. *BioMed research international*, 2013, 2013.

[31] NVIDIA. CUDA C Programming Guide 7.5. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2015. Accessed 19 October 2015.

[32] NVIDIA. CUDA Runtime API Reference Manual 7.5. `http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`, 2015. Accessed 19 October 2015.

[33] NVIDIA. *The NVIDIA GPU Computing SDK Version 5.5*, 2013.

[34] iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. `https://github.com/esnet/iperf`, 2015. Accessed 19 October 2015.

[35] C. Reaño and F Silla. Reducing the performance gap of remote GPU virtualization with InfiniBand Connect-IB. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 920–925, June 2016.

[36] Mellanox. Connect-IB Single and Dual QSFP+ Port PCI Express Gen3 x16 Adapter Card User Manual. `http://www.mellanox.com/related-docs/user_manuals/Connect-IB_Single_and_Dual_QSFP+_Port_PCI_Express_Gen3_%20x16_Adapter_Card_User_Manual.pdf`, 2014. Accessed 19 October 2015.

[37] Mellanox. ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual 1.7. `http://www.mellanox.com/related-docs/user_manuals/ConnectX-3_VPI_Single_and_Dual_QSFP_Port_Adapter_Card_User_Manual.pdf`, 2013. Accessed 19 October 2015.

[38] Ferran Pérez, Carlos Reaño, and Federico Silla. Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA. In *16th International Conference Distributed Applications and Interoperable Systems (DAIS)*, pages 82–95. Springer International Publishing, 2016.

[39] Mellanox. Mellanox OFED for Linux User Manual. `http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.3-1.0.1.pdf`, 2014. Accessed 19 October 2015.

[40] C. Reaño, R. Mayo, E.S. Quintana-Ortí, F. Silla, J. Duato, and A.J. Peña. Influence of InfiniBand FDR on the performance of remote GPU virtualization. In *Proc. of the IEEE International Conference on Cluster Computing, CLUSTER*, pages 1–8, 2013.

[41] Sandia National Laboratories. LAMMPS Molecular Dynamics Simulator. `http://lammps.sandia.gov/`, 2013. Accessed 19 October 2015.

[42] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L. Maskell. CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170–2177, 2010.

[43] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):1–10, 2013.

[44] Panagiotis D Vouzis and Nikolaos V Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[45] NVIDIA. NVIDIA Popular GPU-Accelerated Applications Catalog. `http://www.nvidia.com/content/gpu-applications/PDF/GPU-apps-catalog-mar2015.pdf`, 2015. Accessed 19 October 2015.

[46] Yongchao Liu. CUDA-MEME. `https://sites.google.com/site/yongchaosoftware/mcuda-meme`, 2014. Accessed 19 October 2015.

[47] Adam Polak. Counting triangles in large graphs on GPU. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 740–746, 2016.

[48] Javier Prades and Federico Silla. Turning GPUs into Floating Devices over The Cluster: The Beauty of GPU Migration. In *Proc. of the 6th Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA)*, 2017.

Chapter 3

Made-to-Measure GPUs on Virtual Machines with rCUDA

Javier Prades, Federico Silla. *ICPP'18* **Proceedings of the 47th International Conference on Parallel Processing Companion** - August 2018 - Article: 19 - Pages 1 - 8
https://doi.org/10.1145/3229710.3229741

Abstract

Virtual machines (VMs) are a mature technology widely used worldwide during the last decades. VMs allow to reduce acquisition costs of data centers as well as reduce the cost of operating such computing facilities, mainly regarding electricity costs. However, although VMs are a well-established technology, they do not efficiently address yet the usage of CUDA-compatible GPUs (Graphics Processing Units) for computation purposes, which are commonly used in order to reduce the execution time of applications. The main concern of the way VMs use GPUs is that these devices cannot be concurrently shared among VMs and, therefore, the flexibility provided by VMs is not extended to GPUs.

In this paper we propose to use the rCUDA remote GPU virtualization middleware in order to efficiently share GPUs among VMs. Our experiments show that sharing GPUs among VMs is beneficial in terms of overall throughput while increasing individual execution time of applications by a small percentage. Additionally, different levels of overhead can be decided in order to provide customers different qualities of service, which would cost a different fee. On the other hand, in addition to an increase in overall throughput, total energy consumption is decreased.

3.1 Introduction

Virtual machines (VMs) have been widely used during the last decades. In addition to provide an extraordinary flexibility to system administrators, who use them to detach services from bare metal in a transparent way to users, VMs are also intensively employed in cloud computing platforms such as Amazon Web Services, Microsoft Azure, etc. Furthermore, several companies provide virtualization solutions commonly used worldwide, such as VMware or Citrix, among other examples. Several virtualization frameworks are available, such as Xen [1], KVM [2], VMware [3], etc.

VMs allow to increase the utilization of the underlying hardware resources thus leading to an increase in the benefits obtained by service providers when renting those resources to customers. The most straightforward example is the comparison among renting a given server to a single customer and renting fractions of that very same server (in the form of different VMs) to several customers. Additionally, the different fractions of the server do not necessarily must have the same characteristics (amount of CPU cores and memory). Furthermore, it is possible to migrate VMs among real servers in order to make a better overall usage of the bare metal resources. Server consolidation actually allows the cost of data centers to be reduced by 45% [4]. Notice that in most data centers, bare metal resources are over provisioned in order to attain good performance when experiencing bursty workloads [5]. In this regard, data collected from more than 5000 production servers over a six-month time frame has shown that servers operate, most of the time, between 10% and 50% of their full capacity, thus wasting energy during low utilization periods [6]. Therefore, using VMs along with server consolidation is an excellent way to efficiently address the different workload levels of a given data center.

Although the use of VMs provides a lot of flexibility to data centers, VMs, however, do not efficiently address the use of GPUs (Graphics Processing Units). These devices have been widely used during the last decade as a way to reduce the execution time of applications belonging to domains as different as chemical physics [7], algebra [8], finance [9], computational fluid dynamics [10], biology [11], data analysis (Big Data) [12], image analysis [13], and artificial intelligence [14]. This reduction in the execution time is achieved because the most compute-intensive parts of these applications are offloaded

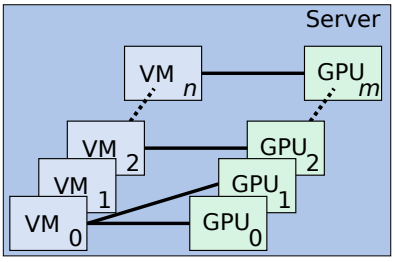


FIGURE 3.1: Assignment of GPUs to VMs when using the PCI passthrough technique, which causes that GPUs are assigned to VMs in an exclusive way.

to these devices, which behave as accelerators. Frameworks such as CUDA [15] assist programmers in using GPUs for general-purpose computing.

The reason why VMs do not properly address the usage of CUDA-compatible GPUs is because current technology does not allow to efficiently share a given GPU among several VMs for CUDA-acceleration purposes. Typically, GPUs are provided to VMs by making use of the PCI passthrough technique [16][17], which allows a GPU to be assigned to a VM in an exclusive way. Thus, the PCI passthrough technique does not allow to share a GPU among several VMs. This is shown in Figure 3.1, which depicts a server with n VMs and m GPUs. The GPUs in the server can only be assigned to VMs running in that server. Furthermore, the GPU-to-VM assignment is carried out in an exclusive manner and, therefore, a given GPU (or set of GPUs) will only be assigned to a single VM at a time. Later in this paper, in the performance evaluation section, a similar hardware configuration with 4 VMs and 4 GPUs will be used. Notice that, in Figure 3.1, in the case that the server hosts more VMs than GPUs, then some of the VMs would not have access to GPUs until other VMs release them. This lack of flexibility is opposed to the general idea of VMs, which is increasing the usage intensity of the underlying hardware. Notice that NVIDIA designed the GRID GPUs, which can be shared among several VMs for desktop virtualization but they do not provide CUDA acceleration services while being shared among several VMs. Thus, this type of GPUs do not properly address the aforementioned concern.

In order to provide GPU acceleration to VMs in an efficient and flexible way, the remote GPU virtualization mechanism [18] can be used. This technique allows a GPU to be transparently shared among several applications (or VMs) without requiring any modification in the source code of the application. This mechanism detaches GPUs from nodes, thus allowing applications (or VMs) to access virtualized GPUs regardless of the

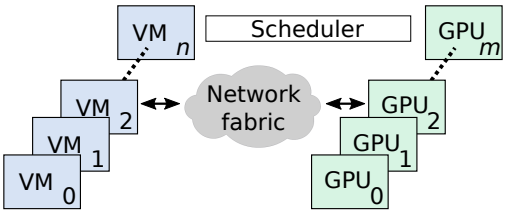


FIGURE 3.2: Assignment of GPUs to VMs when using the remote GPU virtualization mechanism, which allows GPUs to be concurrently shared among VMs.

exact computer where they are being executed. This is shown in Figure 3.2, where n VMs located in one or more bare metal servers access m GPUs (m can be smaller than n) located anywhere in the cluster. Obviously, a scheduler is required to properly orchestrate the assignment of GPUs to VMs without incurring in resource allocation errors. As can be seen, the detaching capabilities of the remote GPU virtualization mechanism is an efficient way to address the mentioned limitations of VMs regarding the use of GPUs as accelerators. The remote GPU virtualization mechanism also allows to migrate GPU jobs among nodes in the cluster [19], thus being a very efficient way to achieve server consolidation in the context of CUDA applications.

In this paper we explore the usage of the remote GPU virtualization mechanism in order to share a set of GPUs among several VMs. To that end, we leverage the rCUDA [18] middleware along with the KVM virtualization framework in order to execute several production applications inside the VMs.

The rest of the paper is organized as follows. In Section 3.2 a thorough motivation for sharing GPUs among VMs is presented. Later, Section 3.3 introduces the necessary background on the remote GPU virtualization mechanism showing also that the overhead introduced by such a technique is very small. Section 3.4 presents the performance evaluation of our proposal. Finally, Section 3.5 summarizes the main conclusions from this work and presents several research lines for future work.

3.2 Motivation

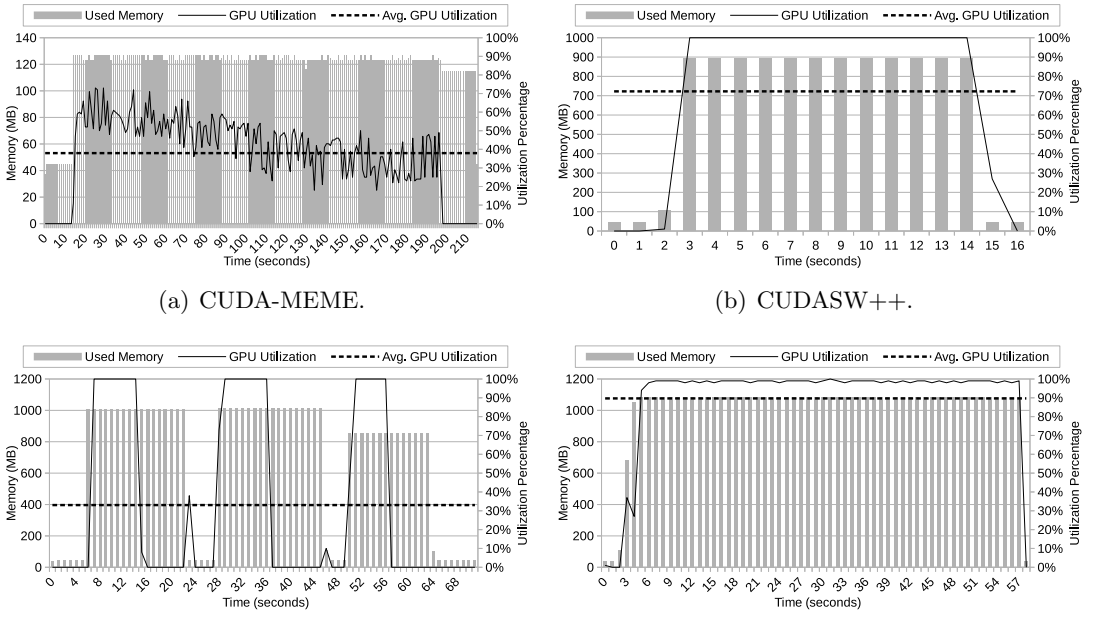
Our proposal in this paper is to share GPUs among several VMs for CUDA acceleration purposes. This possibility is not currently considered by data centers because GPUs must be assigned to VMs in an exclusive way by leveraging the PCI passthrough mechanism.

In the context of our proposal, the first question that may arise is whether it is useful to share a given GPU among several VMs. Two are the main limiting factors that may appear: (1) memory and (2) computing power. The first one refers to the fact that when a GPU is shared among several VMs, the sum of the memory allocated by each of them must not exceed the total amount of memory available in the GPU in order to avoid memory allocation errors that would otherwise cause applications to abort execution. On the other hand, the second limiting factor, computing power, refers to the fact that the cores in the GPU must be multiplexed among the several VMs sharing the GPU.

This multiplexing will cause that all the applications being concurrently served by the GPU may, obviously, experience an increase in their execution time. Notice, however, that applications do not usually make a continuous usage of the computing resources of the GPU because applications typically interleave periods of time using the CPU and periods of time using the GPU. Therefore, when an application is not using the GPU cores, other applications could use them, thus diminishing the aforementioned increment in execution time.

In order to analyze these two limiting factors we have studied the evolution of four different applications when executed with a NVIDIA Tesla K20 GPU [20] installed in a Supermicro server containing two 6-core Intel Xeon E5-2620 v2 processors and 32 GB of DDR3 SDRAM memory at 1600MHz. These applications, which will be used as test cases in this paper, are the following:

- CUDA-MEME [21] is a parallel CUDA implementation of the MEME algorithm, used for discovering motifs in a group of related DNA or protein sequences.
- CUDASW++ [22] is a bioinformatics software for Smith-Waterman protein database searches that takes advantage of the massively parallel CUDA architecture of NVIDIA Tesla GPUs to perform sequence searches.
- GPU-BLAST [23] has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST implementation using GPUs.
- LAMMPS [24] is a molecular dynamics simulator that can be used to model atoms or, more generically, as a parallel particle simulator.



(a) CUDA-MEME.

(b) CUDASW++.

(c) GPU-BLAST.

(d) LAMMPS.

FIGURE 3.3: Evolution of GPU utilization and memory occupancy during the execution of the four applications considered in this study.

Figure 3.3 shows the evolution of GPU utilization and memory occupancy during the execution of these four applications. In the case of the CUDA-MEME application, we have used for the execution DNA sequences with a length of 500. In a similar way, for the GPU-BLAST application we have leveraged a dataset with 2000 queries. In the case of the CUDASW++ application we used 5478 queries for the dataset. Finally, for the execution of the LAMMPS application we have used the Cu_u3.eam input file, which contains 1,536,000 atoms.

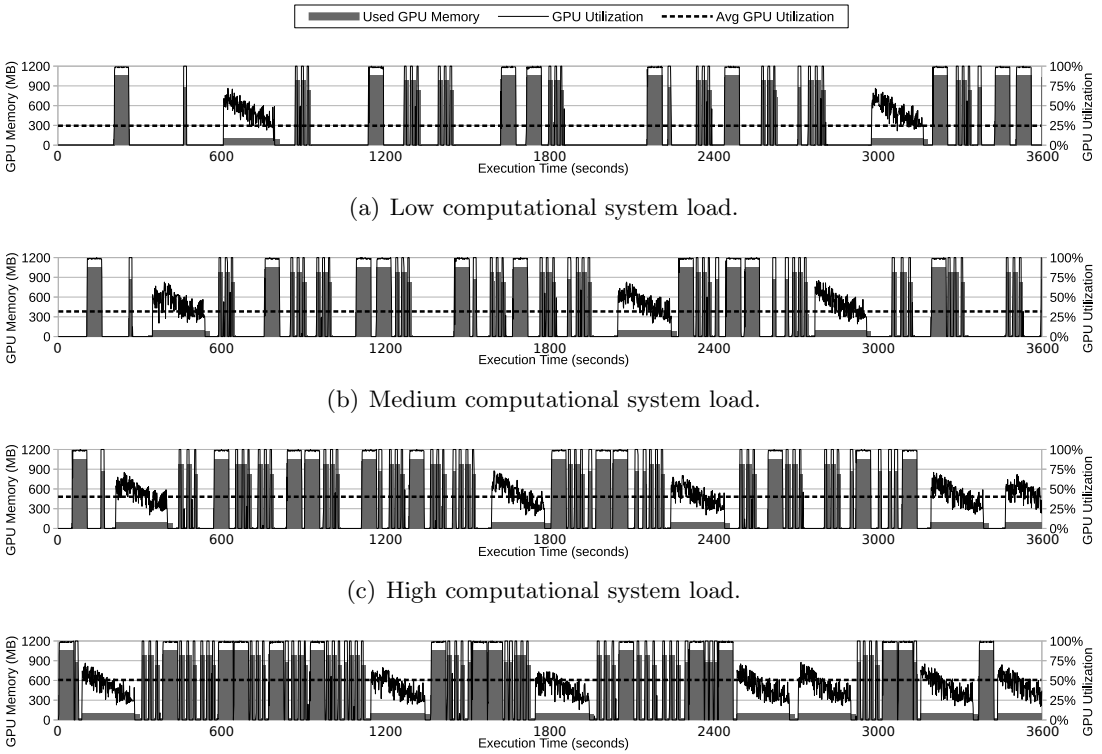
It can be seen in the figure that the four applications present a very different behavior, although in general memory occupancy is well far away from the approximately 5 GB of memory available in the K20 GPU. Additionally, GPU utilization is, in general, relatively low. In this regard, both the CUDA-MEME and GPU-BLAST applications present an average GPU utilization lower than 40%. In the case of the CUDASW++ application, average GPU utilization is increased up to 70%. Finally, the LAMMPS application presents the highest average GPU utilization, although not reaching 90%. It is noteworthy that all the four applications present intervals of time when the GPU is not used at all. As can be seen, none of the two limiting factors mentioned before are, in practice, a burden for concurrently sharing a GPU, at least for these four applications.

Notice that as more powerful GPUs are used, featuring both more cores and more memory, the mentioned limiting factors will progressively represent weaker constraints.

Figure 3.3 showed the evolution of GPU utilization and memory occupancy during the execution of an individual instance of each of the four applications considered in this study. Nevertheless, we can augment our motivation for concurrently sharing GPUs by executing a sequence of applications. Figure 3.4 shows the evolution of GPU utilization and memory occupancy during the execution of such a sequence of applications. Application order is randomly selected. Notice that applications are executed sequentially and therefore the GPU is not concurrently shared among them. In order to model several system loads, we have used different amounts of time among the completion of an application and the beginning of the next one. To that end, we have inserted 90 seconds among applications in order to model a low system load. In a similar way, we have inserted 45 seconds between applications in order to model a medium system load. In order to model high load, we have used 22 seconds between applications. Finally, a maximum system load has been modeled by not inserting any amount of time between applications. As can be seen four system loads are considered: from very low loads to continuous application arrival.

It can be seen in Figure 3.4 that average GPU utilization increases from 25% for the lowest loaded system up to 50% for the most loaded one. It is important to notice that even in the most loaded configuration, average GPU utilization does not exceed 50%. This means that, in general, this expensive resource is underutilized, causing a large delay in amortizing the initial economic investment carried out at purchase time. Furthermore, this underutilization also translates into a waste of energy, given that GPU power consumption is not linear with its utilization but it behaves more like a binary system where an idle GPU consumes few energy but a non-idle GPU presents a power consumption close to the maximum.

As a summary of all the information presented in this section and, according to the experience with the four applications considered in this paper, sharing a GPU among different applications is not necessarily a bad idea. Later in the paper we will show which are the effects of such sharing.



(a) Low computational system load.

(b) Medium computational system load.

(c) High computational system load.

(d) Maximum computational system load.

FIGURE 3.4: Evolution of GPU utilization and memory occupancy during the execution of a sequence of instances of the four applications considered in this paper for one hour time interval. Only one application is executed at a time. The order of applications is random. Four different intensities for the system load are considered.

3.3 Background on GPU Virtualization

Several remote GPU virtualization solutions exist for CUDA, although the rCUDA middleware [18] is the most modern one as well as the that provides the best performance [25]. Basically, these middleware proposals share a GPU by virtualizing it. In this way, these middleware solutions provide applications with virtual instances of the real device, which can therefore be concurrently shared. Usually, these GPU sharing solutions place the virtualization boundary at the API level (CUDA in the case of NVIDIA GPUs). In general, CUDA-based virtualization solutions aim to offer the same API as the NVIDIA CUDA Runtime API [26] does.

It can be seen in Figure 3.5 that the rCUDA middleware follows a distributed client-server approach. rCUDA works as follows: every time the application performs a call to a CUDA function, that call is received by the client side of the middleware which forwards it to the server side running in the node owning the GPU. There, the request

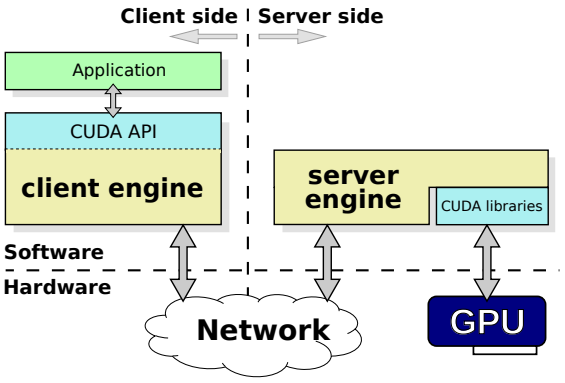


FIGURE 3.5: Architecture of the rCUDA middleware.

is interpreted and forwarded to the GPU. Upon completing the execution of the CUDA function in the real GPU, the results of such function are returned back from the server to the client side of the middleware. Finally, the client middleware forwards those results to the CUDA application.

It is important to notice that rCUDA provides GPU virtualization in a transparent way. That is, applications using rCUDA are not aware that their calls to CUDA functions are actually being serviced by a GPU located in another cluster node instead of by a GPU located in the local node.

rCUDA is binary compatible with CUDA. This means that the source code of CUDA applications do not have to be modified for using rCUDA. Moreover, contrary to other remote GPU virtualization solutions, rCUDA implements the entire CUDA API (except for graphics functions), also providing full compatibility for the other CUDA libraries such as cuSPARSE, cuDNN, cuSOLVER, etc. Regarding network support, rCUDA features different interconnects, such as TCP/IP, InfiniBand and RoCE.

As a final consideration for this background section, it is important to remark that although remote GPU virtualization has traditionally introduced a non-negligible overhead, given that applications do not access GPUs attached to the local PCI Express (PCIe) link but rather access devices that are installed in other nodes of the cluster (traversing a network fabric with a lower bandwidth), this performance overhead has significantly been reduced thanks to the recent advances in networking technologies as well as a careful design of the remote virtualization solution, as shown in [18]. Furthermore, in the context of the study presented in this paper, Figure 3.6 shows the execution

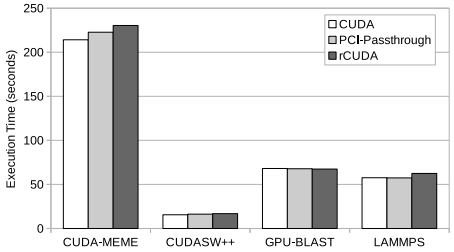
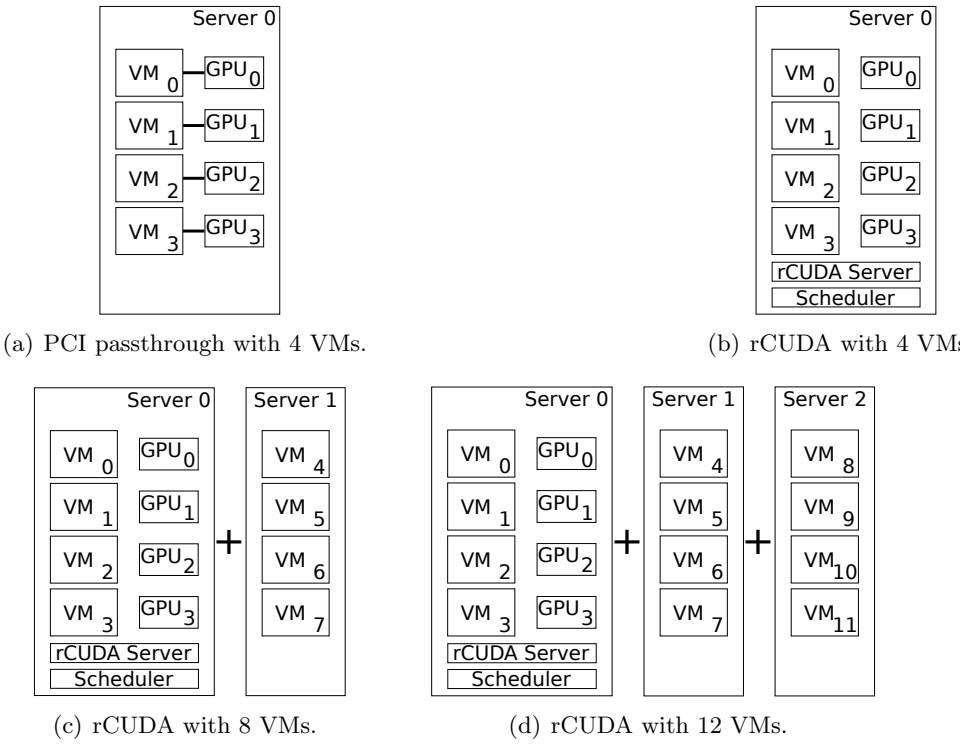


FIGURE 3.6: Execution time of the four applications under consideration in this study.

time of the four applications considered in this study. Three different scenarios are leveraged: (1) "*CUDA*", where applications have been executed in a native domain (without using VMs), (2) "*PCI-Passthrough*", where applications have been run inside a VM by accessing the GPU in the host thanks to the PCI passthrough mechanism, and (3) "*rCUDA*", where applications have been executed inside a VM by accessing a GPU located in another node of the cluster thanks to the rCUDA middleware. The FDR InfiniBand fabric was used in this latter case to communicate the VM and the remote GPU. It can be seen in Figure 3.6 that the overhead introduced by the rCUDA middleware is negligible for the CUDASW++, GPU-Blast and LAMMPS applications whereas for the CUDA-MEME application the overhead is about 10%. In this case it is important to remark that using the PCI passthrough mechanism also provides an important overhead.

3.4 Performance Evaluation

This section presents the perfomance evaluation of our proposal for using the rCUDA middleware in order to share GPUs among VMs. Figure 3.7 shows the four test beds that will be used in the experiments: Figure 3.7(a) shows the traditional system configuration, already discussed in Section 3.1 in Figure 3.1. In this case, we leverage a SYS7047GR-TRF Supermicro server, equipped with two 6-core Intel Xeon E5-2620 v2 processors, four Tesla K20m GPUs and 128 GB of DDR3 SDRAM memory at 1600MHz. This server also owns a Mellanox ConnectX-3 VPI single-port InfiniBand adapter (FDR InfiniBand). The rest of test beds depicted in Figure 3.7 make use of rCUDA in order to access (and share) the available GPUs. Figure 3.7(b) makes use of the same server as in Figure 3.7(a). In this case GPUs are accessed by making use of rCUDA in the local server. In the other two test beds (Figures 3.7(c) and 3.7(d)) two additional SYS1027GR-TRF Supermicro



(a) PCI passthrough with 4 VMs.

(b) rCUDA with 4 VMs.

(c) rCUDA with 8 VMs.

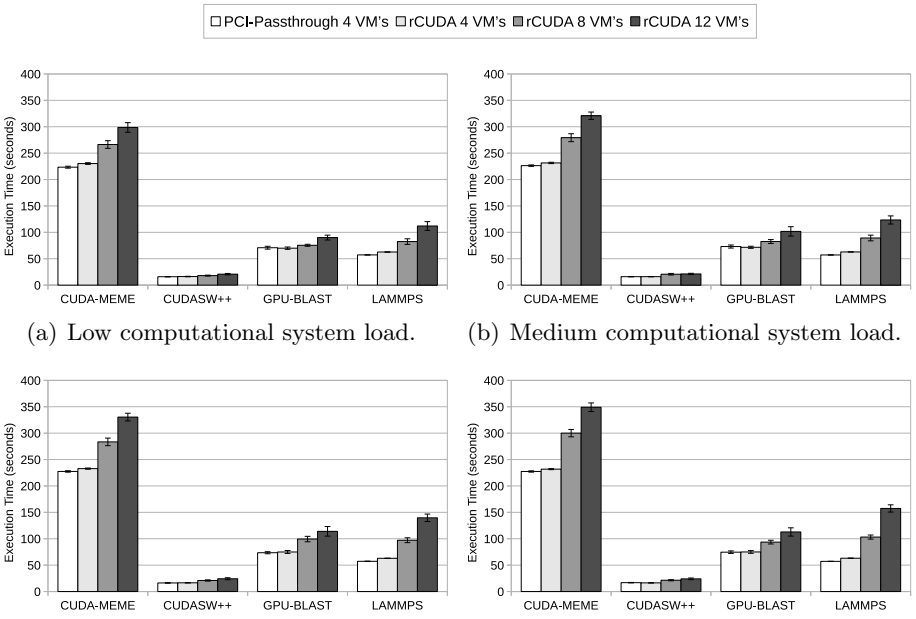
(d) rCUDA with 12 VMs.

FIGURE 3.7: Test beds used in the experiments in this section. Four VMs are used with PCI passthrough (with four GPUs) whereas up to 12 VMs are leveraged with rCUDA (with the same four GPUs).

servers, each of them equipped with two 6-core Intel Xeon E5-2620 v2 processors, 32 GB of DDR3 SDRAM memory at 1600MHz and one FDR InfiniBand adapter are used to host additional VMs. Notice that when a GPU is shared among several VMs, such as in Figures 3.7(c) and 3.7(d), each of the VMs can only make use of a fraction of the GPU memory. Notice also that the these fractions of the GPU memory do not necessarily have to be of the same size but it is possible to assign each VM a different amount of GPU memory, as far as the total sum does not exceed the total GPU memory. In our experiments we have equally distributed the memory available in the GPU among the VMs sharing that GPU.

Regarding the software configuration, CentOS 7.3.1611 was used in the three servers that were used to host the VMs. These servers also used KVM kernel module with QEMU version 1.5.3 as well as Mellanox OFED 4.1-1.0.2. On the other hand, CentOS 7.2.15.11 was used in the VMs along with CUDA 8.0. rCUDA version 18.03beta was used.

Figure 3.8 shows the averaged execution times of the four applications used as test cases



(a) Low computational system load.

(b) Medium computational system load.

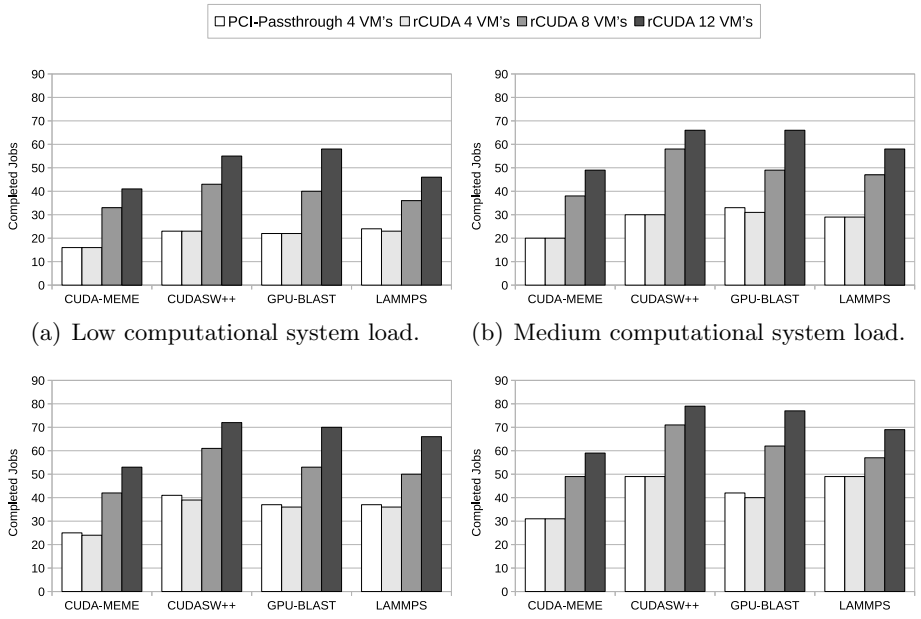
(c) High computational system load.

(d) Maximum computational system load.

FIGURE 3.8: Average execution time for each of the applications considered in this study when executed in the scenarios depicted in Figure 3.7. 95% confidence intervals are also shown. Four different intensities for the system load are considered.

when run in the scenarios depicted in Figure 3.7. As it was the case for Figure 3.4, four different intensities of the system load were used: low, medium, high and maximum. Results shown in Figure 3.8 were also gathered in the executions that were run for one hour. It can be seen in Figure 3.8 that when rCUDA is used without sharing GPUs, the overhead is negligible when compared to the PCI passthrough case (baseline cae in our study). This happens for all the system load intensities and all the applications considered. On the other hand, when GPUs are shared among VMs, we can see that for the "*rCUDA 8 VMs*" and "*rCUDA 12 VMs*" cases, where each GPU is shared among 2 and 3 VMs, respectively, the overhead is increased. This increment in the overhead is more noticeable as the system load increases. The maximum overhead is always experienced when GPUs are shared among 3 VMs.

Figure 3.9 shows the system throughput in terms of number of completed jobs for the one period used for the experiments. In the same way as for the overhead shown in Figure 3.8, similar throughput numbers are obtained in the baseline case (PCI passthrough) and in the rCUDA case when GPUs are not shared. It can also be seen that system throughput increases when GPUs are shared among VMs. Actually, the amount of completed jobs (system throughput) increases as the GPUs are shared among more VMs.



(a) Low computational system load. (b) Medium computational system load.

(c) High computational system load. (d) Maximum computational system load.

FIGURE 3.9: Total amount of jobs executed (system throughput) for each of the applications considered in this study when executed in the scenarios depicted in Figure 3.7. Four different intensities for the system load are considered.

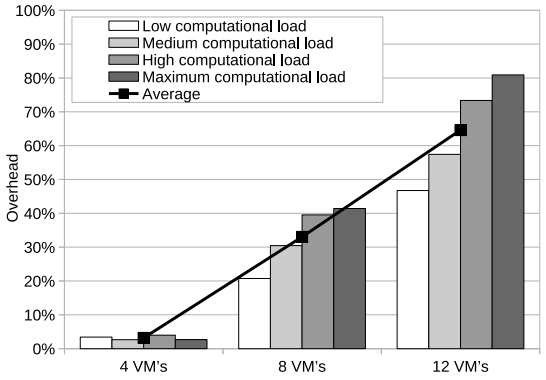


FIGURE 3.10: Average overhead depending on system load for the different test beds depicted in Figure 3.7. Baseline for the overhead calculation is the performance for the PCI passthrough scenario (Figure 3.7(a)).

Figures 3.10 and 3.11 summarize the results in Figures 3.8 and Figure 3.9. Figure 3.10 shows the average overhead, with respect to the PCI passthrough case, for each of the different system loads. For instance, in the case of 8 VMs, the average overhead is 33%. In the case of 12 VMs the average overhead is 65%. As explained before, these results demonstrate that sharing GPUs among VMs with rCUDA is beneficial for the data center because overall throughput, with respect to the PCI passthrough case, is increased, as shown in Figure 3.11. This figure shows that, despite overall throughput is not increased in a linear way with the number of VMs, sharing GPUs with rCUDA is

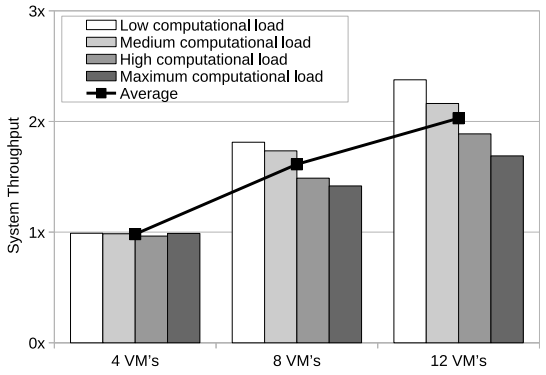


FIGURE 3.11: Average system throughput depending on system load for the different test beds depicted in Figure 3.7. Baseline for the throughput calculation is the performance for the PCI passthrough scenario (Figure 3.7(a)).

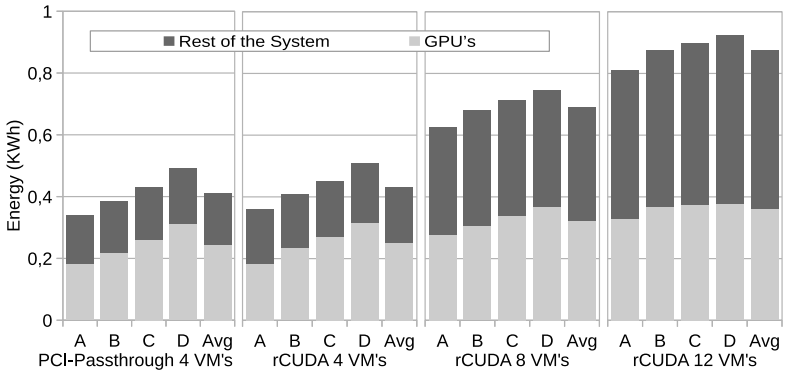


FIGURE 3.12: Energy consumption for each of the scenarios depicted in Figure 3.7. Labels "A", "B", "C" and "D" refer, respectively, to low, medium, high and maximum intensities of the system load.

worth when compared to the traditional GPU assignment mechanism based on the PCI passthrough technique.

The results in Figures 3.10 and 3.11 can also be seen from another point of view: with 4 GPUs, the PCI passthrough mechanism allows to provide service to 4 VMs with optimal performance. On the contrary, when rCUDA is used, an average overhead of 3% is experienced by those same 4 VMs. However, these same 4 GPUs can also provide service to 8 VMs with some overhead (32%) whereas 12 VMs can also be served if the data center is eager to provide service with higher overhead (65%). These different overhead levels allow for the creation of different service levels that could be charged differently to customers. For instance, those users willing to receive the best service would pay a higher fee for the 3% overhead. On the contrary, those users that prefer to pay a fee as small as possible would be serviced with the 65% overhead. In all the cases the throughput of the data center is increased.

In order to complete the study presented in this section, we have to consider energy consumption. Figure 3.12 presents the energy consumed by each of the test cases considered in this study. It can be seen that increasing the amount of VMs sharing a given GPU makes that energy consumption also increases. There are two reasons for this increment. On the one hand, when increasing the amount of VMs we are also increasing the amount of servers (in order to host the additional VMs). On the other hand, the energy consumed by the GPUs also increases because they are being used more time than before. Nevertheless, notice that the increment in the energy consumed by the GPUs is very small. If a deep analysis is carried out, we can see that energy consumed in the "*rCUDA 8 VMs*" case is 1.68 times larger than the energy consumed in the PCI passthrough case. However, in the "*rCUDA 8 VMs*" the amount of VMs serviced are twice the amount of VMs serviced in the PCI passthrough case, thus clearly obtaining a reduction in the energy-per-VM ratio. Similarly, the energy consumption in the case "*rCUDA 12 VMs*" is 2.1 times the energy consumed in the baseline case, although the amount of VMs serviced is 3 times larger than the amount of VMs serviced in the baseline case. Again, a clear reduction in the energy-per-VM ratio is achieved.

3.5 Conclusions

In this paper we have proposed to use the rCUDA middleware in order to provide efficient GPU access to applications running in VMs. The results of this study can be used by cloud computing providers, for instance.

Several are the conclusions from our work. First, rCUDA allows GPUs to be managed in a very efficient way. Actually, if compared to the current case based on the use of the PCI passthough technique, we may even say that rCUDA allows to manage GPUs whereas the current mechanism does not. Furthermore, rCUDA allows to change the GPU assigned to a VM without the need of rebooting the VM. Also, rCUDA allows to migrate GPUs, carry out a real scheduling process of the use of GPUs, etc. The second conclusion from our study is that the use of rCUDA presents a negligible overhead when GPUs are not shared. In a similar way, energy consumption is not increased. The third conclusion from our study is that it is possible with rCUDA to tailor the GPUs according to the real needs of customers. In this way, we can modulate the sharing degree of the

GPUs in order to obtain more powerful GPUs or less powerful GPUs, each of them with a different price.

Acknowledgments

This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/077. Authors are also grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] Xen Project. `http://www.xenproject.org/`, 2018. Accessed 3 May 2018.

[2] Kernel-based Virtual Machine, KVM. `http://www.linux-kvm.org`, 2018. Accessed 3 May 2018.

[3] VMware virtualization. `http://www.vmware.com/`, 2018. Accessed 3 May 2018.

[4] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2008.

[5] EMC Corporation. EMC VNX virtual provisioning applied technology. `https://www.emc.com/collateral/hardware/white-papers/h8222-vnx-virtual-provisioning-wp.pdf`, 2013. Accessed 3 May 2018.

[6] Rongdong Hu, Jingfei Jiang, Guangming Liu, and Lixin Wang. Efficient resources provisioning based on load forecasting in cloud. *The Scientific World Journal*, 2014.

[7] 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.

[8] Ichitaro Yamazaki et al. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *CCPE*, 2014.

[9] Vladimir Surkov. Parallel option pricing with Fourier space time-stepping method on graphics processing units. *PARCO*, 2010.

[10] Everett H. Phillips et al. Rapid aerodynamic performance prediction on a Cluster of graphics processing units. In *AIAA*, 2009.

[11] Pratul K. Agarwal et al. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *CCPE*, 2013.

[12] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14. ACM, 2014.

[13] Yuancheng Luo and R. Duraiswami. Canny edge detection on nvidia cuda. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, June 2008.

[14] Guo-Heng Luo et al. A parallel Bees Algorithm implementation on GPU. *JSA*, 2014.

[15] NVIDIA. CUDA C Programming Guide. Design Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2017. Accessed 3 May 2018.

[16] MS Vinaya, Nagavijayalakshmi Vydyanathan, and Mrugesh Gajjar. An evaluation of CUDA-enabled virtualization solutions. In *Proc. of the IEEE International Conference on Parallel Distributed and Grid Computing, PDGC*, pages 621–626, 2012.

[17] John Paul Walters, Andrew J. Younge, Dong-In Kang, Ke-Thia Yao, Mikyung Kang, Stephen P. Crago, and Geoffrey C. Fox. GPU-Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *Proc. of the IEEE International Conference on Cloud Computing, CLOUD*, 2014.

[18] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, 2015.

[19] J. Prades and F. Silla. Turning gpus into floating devices over the cluster: The beauty of gpu migration. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 129–136, Aug 2017.

[20] NVIDIA. TESLA K20 GPU ACCELERATOR Board Specification. `http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf`, 2013. Accessed 3 May 2018.

[21] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L. Maskell. CUDA-MEME: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units. *Pattern Recognition Letters*, 31(14):2170 – 2177, 2010.

[22] Yongchao Liu et al. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 2013.

[23] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

[24] W. Michael Brown, Axel Kohlmeyer, Steven J. Plimpton, and Arnold N. Tharrington. Implementing molecular dynamics on hybrid high performance computers: Particle-particle particle-mesh. *Computer Physics Communications*, 183(3):449 – 459, 2012.

[25] Carlos Reaño and F. Silla. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *2015 IEEE International Conference on Cluster Computing*, 2015.

[26] CUDA API Reference Manual 9.0. `https://docs.nvidia.com/cuda/`, 2016.

Chapter 4

Multi-tenant virtual GPUs for optimising performance of a financial risk application

Javier Prades, Blesson Varghese, Carlos Reaño, Federico Silla. **Journal of Parallel and Distributed Computing** - Volume: 108 - October. 2017 - Pages 28 - 44 https://doi.org/10.1016/j.jpdc.2016.06.002

Abstract

Graphics Processing Units (GPUs) are becoming popular accelerators in modern High-Performance Computing (HPC) clusters. Installing GPUs on each node of the cluster is not efficient resulting in high costs and power consumption as well as underutilisation of the accelerator. The research reported in this paper is motivated towards the use of few physical GPUs by providing cluster nodes access to remote GPUs on-demand for a financial risk application. We hypothesise that sharing GPUs between several nodes, referred to as multi-tenancy, reduces the execution time and energy consumed by an application. Two data transfer modes between the CPU and the GPUs, namely concurrent and sequential, are explored. The key result from the experiments is that multi-tenancy with few physical GPUs using sequential data transfers lowers the execution time and the energy consumed, thereby improving the overall performance of the application.

Keywords: GPU Virtualisation, Acceleration-as-a-Service, rCUDA, Multi-tenancy, Energy efficiency.

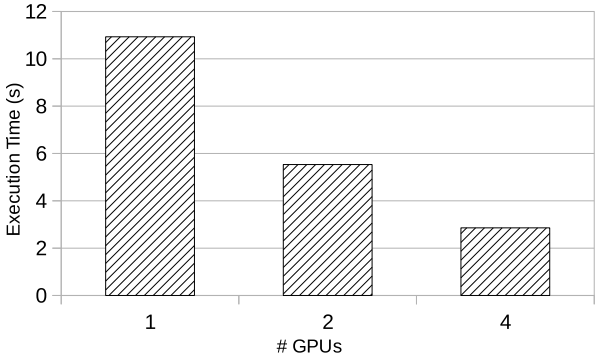


FIGURE 4.1: Execution time of the financial application on multiple local GPUs

4.1 Introduction

Hardware accelerators are achieving a prominent role in modern High-Performance Computing (HPC) clusters for making applications faster. This is evidenced by four out of top ten supercomputers listed on Top500 (http://top500.org) and the top ten supercomputers listed on Green500 (http://www.green500.org) in November 2015 have employed hardware accelerators, such as Graphics Processing Units (GPU). Incorporating GPUs in large clusters allows for heterogeneity, thus making it possible for an application to exploit the regular processor as well as the accelerator [1, 2].

Clusters can now be set up to employ a small number of GPUs by providing applications shared access to remote GPUs on-demand [3, 4]. Such a set up is feasible on a limited budget because not only are a few GPUs used to provide acceleration, but also the energy consumed is well justified since the GPUs are well utilised in the cluster [5, 6]. This is possible as a result of maturing GPU virtualisation technologies that facilitate virtual GPUs (vGPUs) in a cluster. An application can request Acceleration-as-a-Service[7] from one or many vGPUs. One vGPU can reside on a physical GPU (pGPU), referred to as *single tenancy*, but is limiting in that multiple applications cannot make use of the same pGPU since it is exclusively locked for a single application. When multiple vGPUs reside on the same pGPU, otherwise known as *multi-tenancy*, either the same application has access to a pool of vGPUs on the same pGPU or multiple applications can share the same pGPU. We hypothesise that using multi-tenancy can improve the performance of an application.

Numerous challenges arise when multiple GPUs are shared across a cluster for an application, of which three are considered in this paper. The challenges are addressed in this paper by exploring remote CUDA (rCUDA) [8], a GPU virtualisation framework, for improving the performance of a real-world case study employed in the financial industry. The application typically runs in a cluster environment, but can hugely benefit from GPU acceleration for deriving important risk metrics in real-time. The benefit of executing the application on multiple physical GPUs is shown in Figure 4.1. We hypothesise that using a large number of vGPUs can further optimise application performance. However, the following three challenges and research questions arise, which are addressed in this paper: (i) Data will need to be transferred from the CPU to the vGPUs for computations. However, data transfer will be restricted by bottlenecks due to limited bandwidth which affects the overall scalability of the application. Hence, "What data transfer approaches can mitigate the effect of data bottlenecks?" (ii) Multi-tenancy may degrade application performance since the underlying hardware resource is shared. This results in increased execution time and consequently higher energy consumption. Hence, "How can vGPUs be shared effectively to optimise application performance and energy consumed?" (iii) Using multi-tenancy an application can be deployed in multiple ways. For example, an application can be executed on 2 vGPUs residing on 1 pGPU or 8 vGPUs residing on 1 pGPU. These possibilities significantly increase with multiple pGPUs. Each deployment option consumes different amounts of energy and impacts the overall execution time. Hence, "Can performance and energy of an application be estimated in the multi-tenancy approach?"

To address the above challenges we propose two data transfer approaches, namely concurrent and sequential, for transferring data with the aim of mitigating the effect of data bottlenecks. In the context of the financial application, the sequential data transfer approach is expected to improve performance since data transfers from the CPU to the GPU and GPU computations can be overlapped for multiple pGPUs. The approach is further extended for overlapping the data movement and computation time for multiple vGPUs on the same pGPU resulting in a further improvement in performance of the application. The key result is that the financial application can be executed under two seconds for deriving risk metrics in an energy efficient manner on the same hardware compared to single tenancy thus confirming our initial hypothesis. Performance and energy consumed by the application are modelled to determine the combination of vGPUs

on a pGPU that can maximise performance and GPU utilisation and at the same time minimise the energy consumed.

The key contributions of this research are: (i) investigating the lack of scalability due to data transfer from CPU to the GPU in the context of the financial risk application, (ii) proposing two approaches to transfer data, namely concurrent and sequential, (iii) evaluating the above data transfer approaches in the context of single-tenancy for overlapping computations and data transfer of multiple pGPUs, (iv) developing an approach that exploits multi-tenancy for overlapping computations and data transfer of multiple virtual GPUs on the same physical GPU to optimise the performance of the application, (v) evaluating the performance of the application, considering execution time, GPU utilisation and energy consumed by the application, and (vi) developing a mathematical model to derive deployment options for the application by estimating performance and energy of different combinations of virtual GPUs mapped onto physical GPUs.

The remainder of this paper is organised as follows. Section 4.2 highlights related work in the area of HPC solutions for GPU virtualisation and financial risk applications. Section 4.3 briefly presents the rCUDA framework. Section 4.4 considers a financial risk application for evaluating the feasibility of multi-tenancy for improving performance. Section 4.5 presents the platform, experiments performed and the key results obtained. Section 4.6 concludes this paper.

4.2 Related work

High Performance Computing (HPC) solutions are exploited in the financial risk industry to accelerate the underlying computations of applications. This reduces overall execution times making such applications fit for real-time use. Solutions range from small scale clusters [9, 10] to large supercomputers [11, 12]. More recently, hardware accelerators with multi-core and many-core processors are employed. For example, financial risk applications are accelerated on Cell BE processors [13, 14], FPGAs [15, 16] and GPUs [17, 18].

HPC clusters offering heterogeneous solutions by using hardware accelerators, such as GPUs, along with processors on nodes are feasible [1, 2]. Clusters can be set up to

incorporate a GPU on each node. This is not an efficient solution for accelerating an application because of the relatively high cost of GPUs, high power consumption of nodes using GPUs and the under utilisation of GPUs (applications do not require acceleration of GPUs during their entire execution). However, a more efficient solution would be if nodes executing an application can access GPUs when required. This can be facilitated by GPU virtualisation. Currently there are no solutions available for the financial risk industry to harness the potential of GPU virtualisation. In this paper, we investigate the use of virtual GPUs for a financial risk application.

The mechanism of GPU virtualisation allows nodes of a cluster that do not own a physical GPU for accelerating computations of applications that run on it to remotely access GPUs. Acceleration is obtained as a service seamlessly to a requesting node without being aware of accessing remote GPUs. A single application (running on a Virtual Machine (VM) or on a node of a cluster without a hardware accelerator) benefits from the acceleration of a remotely located single GPU or multiple GPUs to reduce execution time. The rate of GPU utilisation can be increased since multiple applications can access the same GPU. This in turn reduces the number of GPUs that need to be installed in a cluster, and reduces the cost spent on energy consumption, cooling, physical space and maintenance, usually referred to as the Total Cost of Ownership (TCO). Furthermore, the source code of an application usually does not need any modification to reap the benefits of virtual GPUs.

GPU virtualisation is usually applied at the high-level Application Programming Interface (API) of GPUs because low level protocols used to interact with accelerators are proprietary and, additionally, not publicly available. Hence, APIs such as CUDA [19] or OpenCL [20] are used. In this paper we use CUDA (Compute Unified Device Architecture) for an application that is used in the financial risk industry.

There are several remote GPU virtualization frameworks supporting CUDA. GridCuda [21] supports CUDA 3.2, although it is not publicly available. vCUDA [22] supports the CUDA 3.2 and implements an unspecified subset of the CUDA runtime API. The communication protocol between the node that executes the application and the remote GPU has a considerable overhead, because of the costs incurred during encoding and decoding, which results in a noticeable drop of overall performance. GViM [23] is based on CUDA 1.1 and does not implement the entire runtime API. Furthermore,

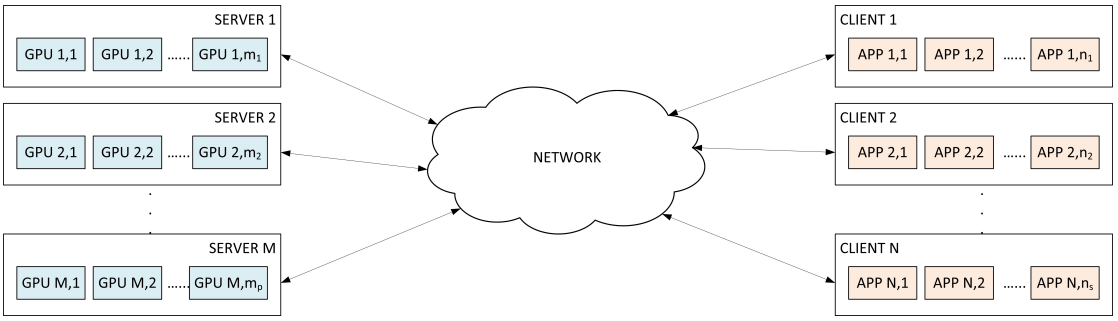


FIGURE 4.2: Distributed acceleration architecture facilitated by rCUDA

GViM is designed to be used on VMs so that applications executed on them can access GPUs located in the real host; GViM does not support the access of GPUs in remote nodes. gVirtuS [24] supports CUDA 2.3 an again implements only a small portion of the runtime API. For example, in the case of the memory management module, it implements only 17 out of the 37 available functions. Although it is intended mainly to be used by VMs for accessing real GPUs located in the same node, it facilitates TCP/IP communications between clients and servers, thus allowing the access to GPUs located in other nodes. DS-CUDA [25] supports CUDA 4.1 and includes specific communication support for InfiniBand Verbs, thus reducing the overhead of communications between the node executing the application and the node owning the GPU. However, DS-CUDA is limited in that it does not allow data transfers with pinned memory and supports maximum data transfer of 32 MB.

The rCUDA framework [8] is binary compatible with CUDA 6.5 and implements the entire CUDA Runtime and Driver APIs (with the exception of graphics functions). It provides support for the libraries included within CUDA, such as cuBLAS or cuFFT. In addition, a number of underlying interconnection technologies are supported by making use of a set of runtime-loadable, network-specific communication modules (currently TCP/IP and InfiniBand). Concurrent virtualization services are made available to remote clients simultaneously demanding GPU acceleration by managing an independent GPU context for each client. rCUDA performs better than other publicly available GPU virtualisation frameworks (considered in Section 4.3) and is therefore chosen for this research.

4.3 rCUDA

The rCUDA framework, otherwise referred to as remote CUDA, is used in the research presented in this paper. As shown in Figure 4.2, the rCUDA framework is a client-server architecture. Numerous *Clients* executing applications that can benefit from hardware acceleration can concurrently access *Servers* that have physical GPUs on them. The client makes use of the remote GPU to accelerate part of the software code of the application, referred to as kernel, running on it. The framework transparently handles the data management and the execution management; the transfer of data between the local memory of the client, the local memory of the server and the GPU memory, and the remote execution of the kernel.

Figure 4.3 shows the hardware and software stack of the client and the rCUDA server. The client nodes that execute the application (shown in Figure 4.2), make use of the rCUDA Client Library, which is a wrapper around the CUDA Runtime and Driver APIs. The library is responsible for (i) intercepting calls made by the application to a CUDA device, (ii) processing them for forwarding the calls to the remote rCUDA server, and (iii) retrieving the results of the calls from the rCUDA server. On the other hand, each GPU server has an rCUDA daemon running on it which receives CUDA requests and executes them on the physical GPU.

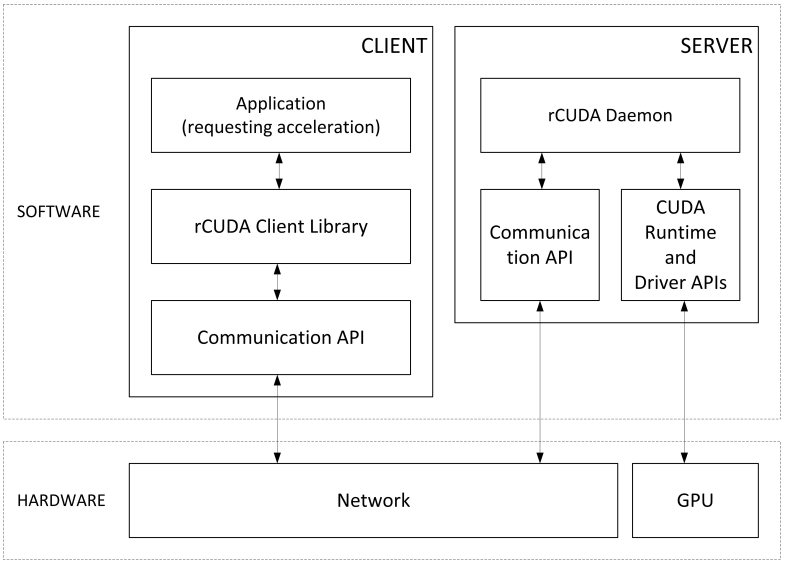


FIGURE 4.3: rCUDA client and server software/hardware stack

An efficient communication protocol is developed for seamless execution between rCUDA clients and servers. This protocol, using either regular TCP/IP sockets or the InfiniBand Verbs API when this high performance interconnect is available in the cluster, is designed to provide lightweight support to the remote CUDA operations provided by the external accelerator. The CUDA commands intercepted by the rCUDA client wrapper are encapsulated into messages in the form of one or more packets that travel across the network towards the rCUDA server. The format of the messages depends on the specific CUDA command transported. In general, the messages have low network overheads. Every CUDA command forwarded to the remote GPU server is followed by a response message, which acknowledges the success/failure of the operation requested on the remote server.

Figure 4.4 shows an example of the communication between the rCUDA client and the rCUDA daemon executing on the remote server. In this example, the following steps occur:

Step 1 - Initialise: The client establishes connection with the remote server automatically, and the request for acceleration services is intercepted and the GPU kernel along with related information such as statically allocated variables are sent to the server.

Step 2 - Allocate Memory: Based on the client request device memory is allocated on the GPU for data that will be required by the GPU kernel. The `cudaMalloc` requests are intercepted by the client and forwarded to the remote server.

Step 3 - Transfer Data to Device: All data required by the kernel is transferred from the host to the remote device.

Step 4 - Execute Kernel: The GPU kernel is executed remotely on the rCUDA server.

Step 5 - Transfer Data to Host: After the execution of the kernel on the remote server the data is transmitted back to the host.

Step 6 - Release Memory: The memory allocated on the remote device is released.

Step 7 - Quit: In this final step the client application stops communicating with the remote server. The rCUDA daemon executing on the server stops servicing the execution and releases the resources associated with the execution.

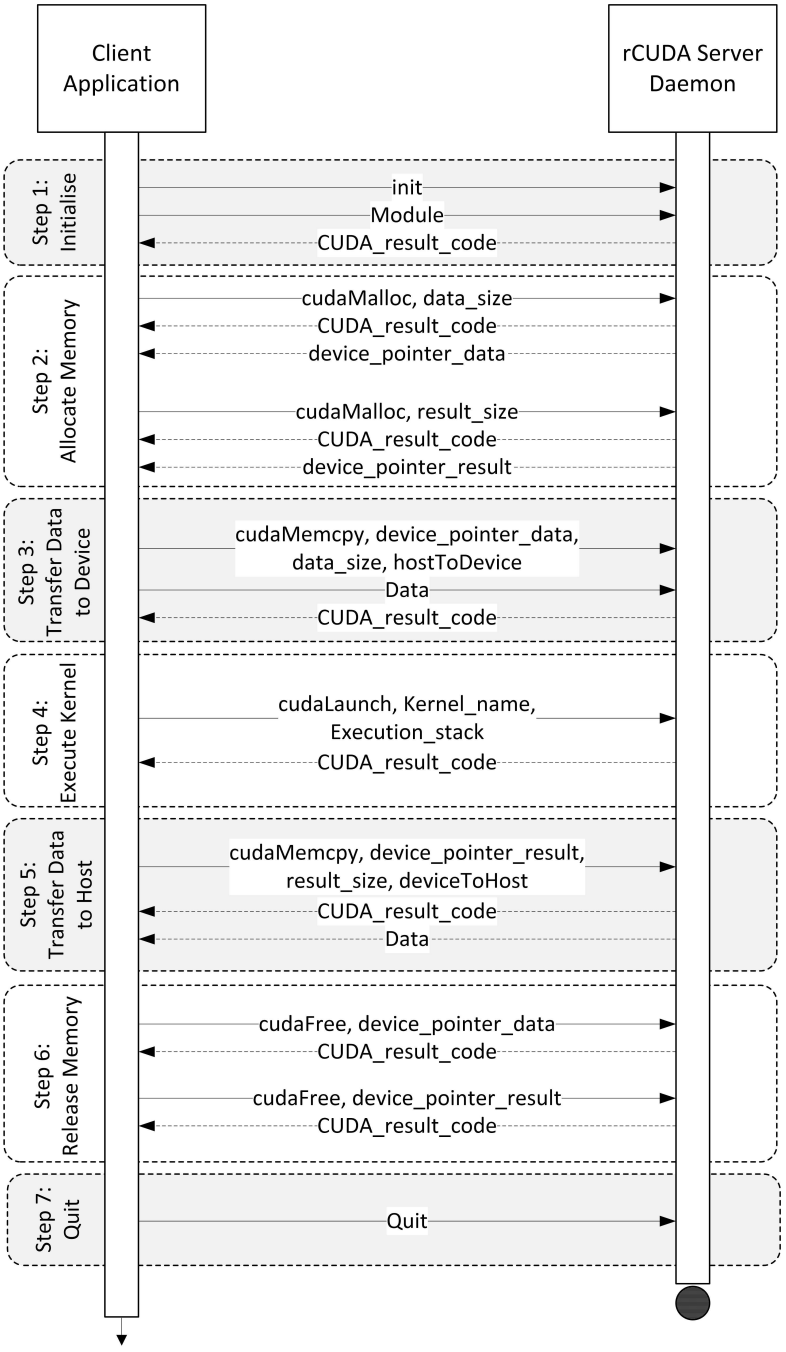
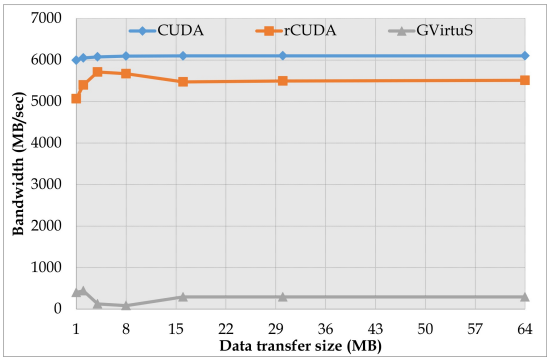
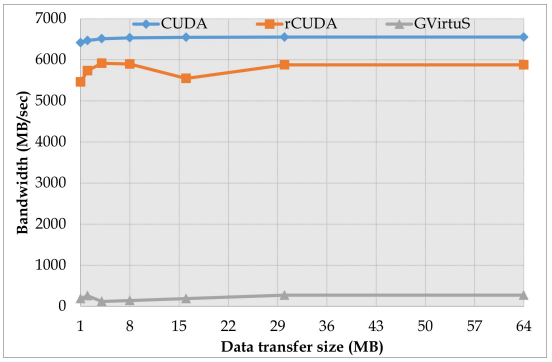


FIGURE 4.4: Communication sequence between a client and the rCUDA server daemon

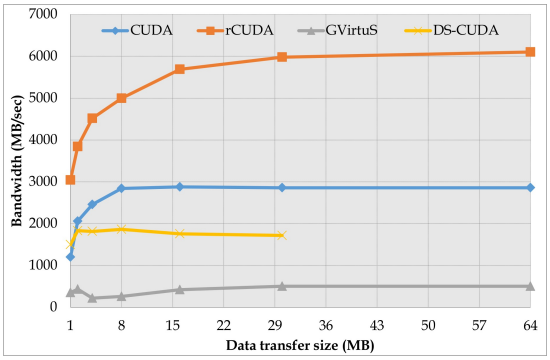
Figure 4.5 compares the performance of publicly available GPU virtualisation frameworks, namely DS-CUDA, gVirtuS and rCUDA by using the `bandwidthTest` benchmark from the NVIDIA CUDA Samples [26]. Our choice of selecting rCUDA for this research is based on its superior performance over other frameworks as shown in the figure. The performance of CUDA 6.5 is used as the baseline reference. Bandwidth is used as a



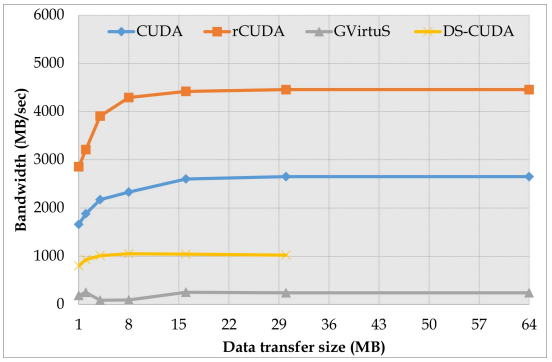
(a) Host pinned memory to device memory



(b) Device memory to host pinned memory



(c) Host pageable memory to device memory



(d) Device memory to host pageable memory

FIGURE 4.5: Comparison of bandwidth for pinned memory and pageable memory of rCUDA, DS-CUDA and gVirtuS using CUDA as a baseline reference (DS-CUDA does not support pinned memory)

measure for comparing performance since it is a limiting factor for data transfers between host (CPU) memory and device (GPU) memory (data size can be in the order of MB) and affects the performance of the virtualisation frameworks. Other metrics such as latency are less relevant in this context.

The test-bed employed for carrying out the bandwidth performance experiments is presented later in Section 4.5.1. Virtual Machine (VMs) were not employed to simplify the experiments. The bandwidth test was run on a native domain and the server side of the virtualisation framework used was executed in a remote node. The InfiniBand FDR network technology was used to connect both nodes. The rCUDA and DS-CUDA frameworks made use of the InfiniBand Verbs API and gVirtuS made use of TCP/IP over InfiniBand since it cannot take advantage of the InfiniBand Verbs API.

The three virtualisation frameworks support different versions of CUDA which had to be used for obtaining the bandwidth benchmarks. DS-CUDA is compatible with CUDA

4.1, gVirtuS supports CUDA 2.3 and rCUDA supports CUDA 6.5. In our experience, employing different CUDA versions has minimal impact on bandwidth performance and therefore no additional noise was introduced by using different versions.

The following observations are made from Figure 4.4. Firstly, CUDA achieves highest performance when pinned memory is used (refer Figure 4.5(a) and Figure 4.5(b)), achieving nearly a bandwidth of 6000 MB/s. The bandwidth is however reduced for copies using pageable memory (refer Figure 4.5(c) and Figure 4.5(d)).

Secondly, Figure 4.5 shows that rCUDA outperforms DS-CUDA and gVirtuS. For copies using pageable memory rCUDA even performs better than CUDA; this has been previously reported, which is due to the use of an efficient pipelined communication between rCUDA clients and servers based on the use of internal and pre-allocated pinned memory buffers [8]. rCUDA and DS-CUDA support InfiniBand Verbs API and therefore have access to large bandwidths which are available on this interconnect. However, DS-CUDA has relatively poor performance when compared to rCUDA. Therefore, it is assumed that both frameworks manage the InfiniBand interconnect differently. DS-CUDA neither supports memory copies larger than 32MB nor pinned memory. The performance of gVirtuS is significantly lower than the other frameworks. It may be immediately inferred that this is because TCP/IP is used and has a lower bandwidth in comparison to InfiniBand Verbs. However, using the `iperf` tool [27], TCP/IP over InfiniBand FDR provides approximately 1190 MB/s, which is a noticeably larger bandwidth than the one achieved by gVirtuS. Therefore, the poor performance of gVirtuS may be due to the inefficient handling of communication.

4.4 Financial risk application

A candidate application that can benefit from Acceleration-as-a-Service (AaaS) in HPC clusters is investigated in this section. We present such an application employed in the financial risk industry, referred to as *'Aggregate Risk Analysis'* [28] for validating the feasibility of our proposed multi-tenancy approach. The analysis of financial risk is underpinned by a simulation that is computationally intensive. Typically, this analysis is periodically performed on a routine basis on production clusters to derive important

risk metrics. Such a set up is sufficient when the analysis does not need to be performed outside routine.

Risk metrics will need to be obtained in real-time, such as in an online pricing scenario, in addition to routine executions. In such settings, a number of input parameters to the analysis will need to be varied to satisfy the customer. This generates a large number of requests to execute the analysis multiple times based on the complexity of the client's portfolio. It may not be feasible to furnish all these requests generated by single or multiple clients; it will be impossible to quickly obtain a large set of resources on an in-house cluster already provisioned for executing other routine jobs. Here, GPUs can play an important role in furnishing a large number of requests.

While GPUs can provide a feasible solution, employing a large number of GPUs to furnish bursts of requests will be expensive. As considered in Section 4.1 virtual GPUs are pragmatic and cost effective to minimise under utilisation. In this context, we leverage the acceleration offered by virtual GPUs in an HPC cluster to develop a faster application fit for use in real-time settings. The rCUDA framework suits such an application because minimal changes need to be brought about to the production cluster and the acceleration required for the analysis is obtained as a service from a remote host. The analysis has previously been investigated in the context of many-core architectures [29], but we believe virtual GPUs can be a better option.

Aggregate risk analysis is performed on a portfolio of risk held by an insurer or reinsurer and provides actuaries and decision makers with millions of alternate views of catastrophic events, such as earthquakes, that can occur and the order in which they can occur in a year. To obtain millions of alternate views, millions of trials are simulated with each trial comprising a set of possible future earthquake events and the probable loss for each trial is estimated.

4.4.1 Input and Output Data

Three data tables are required for the analysis, which are as follows:

i. *Year Event Table*, which is a database of pre-simulated occurrences of events from a catalogue of stochastic events that is denoted as YET. Each record in a YET called a

'trial', denoted as T_i, represents a possible sequence of event occurrences for any given year. The sequence of events is defined by an ordered set of tuples containing the ID of an event and the time-stamp of its occurrence in that trial $T_i = \{(E_{i,1}, t_{i,1}), \ldots, (E_{i,k}, t_{i,k})\}$.

The set is ordered by ascending time-stamp values. A typical YET may comprise thousands to millions of trials, and each trial may have approximately between 800 to 1500 'event time-stamp' pairs, based on a global event catalogue covering multiple perils. The representation of the YET is shown in Equation 4.1, where $i = 1, 2, \ldots$ and $k = 1, 2, \ldots, 1500$.

$$YET = \{T_i = \{(E_{i,1}, t_{i,1}), \ldots, (E_{i,k}, t_{i,k})\}\} \tag{4.1}$$

ii. *Event Loss Tables*, which is a representation of collections of specific events and their corresponding losses with respect to an exposure set denoted as ELT. Each record in an ELT is denoted as $EL_i = \{E_i, l_i\}$ and the financial terms associated with the ELT are represented as a tuple $\mathcal{I} = (\mathcal{I}_1, \mathcal{I}_2, \ldots)$.

A typical aggregate analysis may comprise 10,000 $ELTs$, each containing 10,000-30,000 event losses with exceptions even up to 2,000,000 event losses. The $ELTs$ can be represented as shown in Equation 4.2, where $i = 1, 2, \ldots, 30,000$.

$$ELT = \left\{ \begin{array}{lcl} EL_i & = & \{E_i, l_i\}, \\ \mathcal{I} & = & (\mathcal{I}_1, \mathcal{I}_2, \ldots) \end{array} \right\} \tag{4.2}$$

iii. *Portfolio*, which is denoted as PF and contains a group of Programs, P represented as $PF = \{P_1, P_2, \ldots, P_n\}$ with $n = 1, 2, \ldots, 10$.

Each Program in turn covers a set of Layers, denoted as L, cover a collection of $ELTs$ under a set of layer terms. A single layer L_i is composed of two attributes. Firstly, the set of $ELTs$ $\mathcal{E} = \{ELT_1, ELT_2, \ldots, ELT_j\}$, and secondly, the Layer Terms, denoted as $\mathcal{T} = (\mathcal{T}_1, \mathcal{T}_2, \ldots)$.

A typical Layer covers approximately 3 to 30 individual $ELTs$ and is represented as shown in Equation 4.3, where $j = 1, 2, \ldots, 30$.

$$L = \left\{ \begin{array}{lcl} \mathcal{E} & = & \{ELT_1, ELT_2, \ldots, ELT_j\}, \\ \mathcal{T} & = & (\mathcal{T}_1, \mathcal{T}_2, \ldots) \end{array} \right\} \tag{4.3}$$

The output of the analysis is a loss value associated with each trial of the YET. A reinsurer can derive important portfolio risk metrics such as the Probable Maximum Loss (PML) [30] and the Tail Value-at-Risk (TVaR) [31] which are used for both internal risk management and reporting to regulators and rating agencies. Furthermore, these metrics flow into a final stage of the risk analytics pipeline, namely Enterprise Risk Management, where liability, asset, and other forms of risks are combined and correlated to generate an enterprise wide view of risk.

4.4.2 Algorithm and GPU Implementation

Given the above three inputs, Aggregate Risk Analysis is shown in Algorithm 1. The data tables, YET, ELT and PF, are loaded into host (CPU) memory. The analysis is performed for each Layer and for each Trial in the YET and a Year Loss Table (YLT) is produced. In this paper, we assume a Portfolio comprising one Program and one Layer, and therefore the for loops of lines 1 and 2 iterate once. If there are N available

Algorithm 1: Aggregate Risk Analysis

Input : YET, ELT, PF
Output: YLT

```
1  for each Program, P, in PF do
2      for each Layer, L, in P do
3          Split YET to YET_i, where i = 1, 2, ..., N
4          for each i do
5              TransferDataToDevice (i, YET_i, ELT)
6              LaunchDeviceKernel (i)
7          end
8      end
9  end
10 Populate YLT from YLT_i, where i = 1, 2, ..., N
11 return
```

devices (GPUs), then the YET is split to N smaller $YETs$, represented as YET_i, where $i = 1, 2, \ldots, N$.

There are two functions that facilitate device execution. The first function `TransferDataToDevice` copies YET_i and the ELT to the device memory as shown in Algorithm 2.

Algorithm 2: TransferDataToDevice Function

Input : i

```
1 Select device i
2 Copy YET_i, ELT to device i
3 return
```

The second function `LaunchDeviceKernel` executes the function on the device as shown in Algorithm 3. Each event of a trial and its corresponding event loss in the set of $ELTs$ associated with the Layer is determined. A set of contractual financial terms ($\mathcal{I}$) are applied to each loss value of the Event-Loss pair extracted from an ELT to the benefit of the layer. The event loss for each event occurrence in the trial, combined across all $ELTs$ associated with the layer, are subject to further financial terms ($\mathcal{T}$) [28].

Two occurrence terms, namely (i) Occurrence Retention, $\mathcal{T}_{OccR}$, which is the retention or deductible of the insured for an individual occurrence loss, and (ii) Occurrence Limit, $\mathcal{T}_{OccL}$, which is the limit of coverage the insurer will pay for occurrence losses in excess of the retention are applied. Occurrence terms are applicable to individual event

Algorithm 3: LaunchDeviceKernel Function

Input : i
Output: YLT_i

```
1  Select device i
2  for each Trial, T, in YET_i do
3      for each Event, E, in T do
4          for each ELT covered by L do
5              Lookup E in the ELT and find corresponding loss, l_E
6              Apply Financial Terms to l_E
7              l_T ← l_T + l_E
8          end
9          Apply Financial Terms to l_T
10     end
11 end
12 return
```

occurrences independent of any other occurrences in the trial. The event losses net of occurrence terms are then accumulated into a single aggregate loss for the given trial. The occurrence terms are applied as $l_T = min(max(l_T - \mathcal{T}_{OccR}), \mathcal{T}_{OccL})$.

Two aggregate terms, namely (i) Aggregate Retention, $\mathcal{T}_{AggR}$, which is the retention or deductible of the insured for an annual cumulative loss, and (ii) Aggregate Limit, $\mathcal{T}_{AggL}$, which is the limit or coverage the insurer will pay for annual cumulative losses in excess of the aggregate retention are applied. Aggregate terms are applied to the trial's aggregate loss for a layer. The aggregate loss net of the aggregate terms is referred to as the trial loss or the year loss. The aggregate terms are applied as $l_T = min(max(l_T - \mathcal{T}_{AggR}), \mathcal{T}_{AggL})$.

A single thread is employed for the computations of each trial of the application. $ELTs$ corresponding to a Layer were implemented as direct access tables to facilitate fast lookup of losses corresponding to events. Each ELT is implemented as an independent table; therefore, in a read cycle, each thread independently looks up its events from the $ELTs$. All threads within a block access the same ELT. The device global memory stores all data required for the analysis. Chunking, which refers to processing a block of events of fixed size (or chunk size) for the efficient use of shared memory is employed to optimise the implementation; the computations related to the events in a trial and for applying financial terms benefit from chunking. The financial terms are stored in the streaming multi-processor's constant memory. In this case, chunking reduces the number of global memory update and global read operations.

In this paper, the implementation of fine-grain parallelism in `LaunchDeviceKernel` is not the focus. Instead, the optimisation of performance and efficiency of resource utilisation by managing the two functions, namely `TransferDataToDevice` and `LaunchDeviceKernel` on virtual GPUs is considered and reported in the next section.

4.5 Evaluation

In this section we optimise the performance of the financial risk application to reduce its execution time such that real-time response can be achieved. To this end we present (i) the hardware platform on which the experiments are performed and, (ii) the use of

TABLE 4.1: Scalability of the financial risk application when executed using CUDA

	No. of GPUs		
	1 GPU	2 GPUs	4 GPUs
Total execution time	10.928	5.53	2.857
Normalised execution time	1	0.506	0.261
Execution time with perfect scalability	10.928	5.464	2.732
Offset with respect to perfect scalability	0	0.066	0.125
% offset with respect to perfect scalability	0	1.2%	4.57%

the remote GPU virtualisation framework, and (iii) an approach for transferring data from a CPU to GPUs with the aim of reducing the execution time.

4.5.1 Platform

The experimental platform employed in this research comprises 1027GR-TRF Supermicro nodes. Each node contains two Intel Xeon E5-2620 v2 processors, each with six cores, operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1600 MHz. Each node has a Mellanox ConnectX-3 VPI single-port InfiniBand adapter (InfiniBand FDR) as well as a Mellanox ConnectX-2 VPI single-port adapter (InfiniBand QDR). The nodes are connected either by a Mellanox switch MTS3600 with QDR compatibility (a maximum rate of 40Gb/s) or by a Mellanox Switch SX6025, which is compatible with InfiniBand FDR (a maximum rate of 56Gb/s). One NVIDIA Tesla K20 GPU is available for acceleration on each node. Additionally, one SYS7047GR-TRF Supermicro server with identical processors was populated with 4 NVIDIA Tesla K20 GPUs and 128 GB of DDR3 SDRAM memory at 1600MHz, to serve as a local server for the purpose of comparison. The CentOS 6.4 operating system was used, and the Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools) was used at the servers along with CUDA 6.5.

4.5.2 Application Scalability

As presented in Section 4.1 the use of multiple GPUs reduces the execution time of the application by evenly distributing computations across the GPUs assigned to the application. However, a closer look at the performance as shown in Figure 4.1 highlights

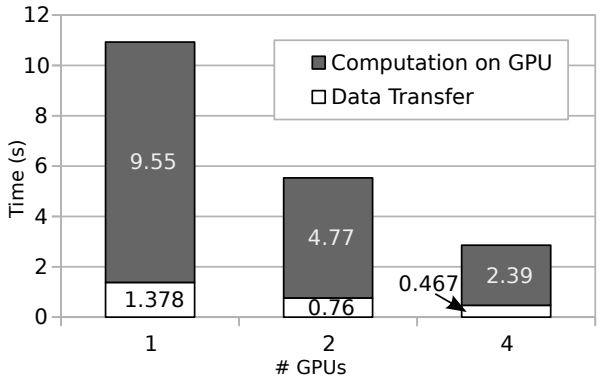
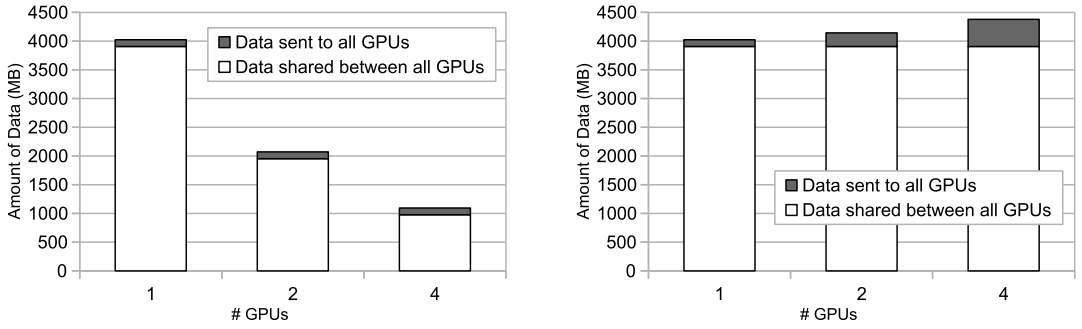


FIGURE 4.6: Computation and data transfer times for the financial risk application when executed on single and multiple GPUs with CUDA

that the scalability of the application as the number of GPUs increases is sub-linear. Table 4.1 is the result of executing the application on the Supermicro SYS7047GR-TRF server using CUDA with up to four GPUs. The normalised execution time indicates that perfect scalability is not achieved. For example, when two GPUs are used the normalised execution time should be 0.5 instead of 0.506 and similarly when four GPUs are employed 0.25 is expected as against 0.261. The offset of execution time with respect to perfect scalability as a reference increases with the number of GPUs involved in the computations.

To account for sub-linear scalability further investigations were carried out. The time taken for computations on the GPUs and the time taken for transferring data to the GPUs (1, 2, and 4 GPUs) were considered as shown in Figure 4.6. The GPU computations take most of the execution time of the application (87.39%, 86.25%, and 63.65% of the total application execution time when 1, 2, and 4 GPUs are used respectively). The GPU computations scale in a perfect manner as the number of GPUs available to the application is increased. However, the time taken for data transfer does not scale well and accounts for 12.6%, 13.74%, and 16.34% of total execution time when 1, 2, and 4 GPUs are used, respectively.

At first glance, it can be assumed that the increase in data transfer time may be due to the lower communication bandwidth of CUDA for transfers of small chunks of data (refer Figure 4.5(c) and Figure 4.5(d)). When pageable memory is transferred the attained bandwidth for data smaller than 10 MB is significantly reduced. Therefore, given that the size of input data transferred to each GPU is progressively reduced as the number of GPUs increases, then the input data may be smaller than 10 MB and thus the effective



(a) Data transferred to each GPU

(b) Total data transferred to all GPUs

FIGURE 4.7: Amount of data transferred during the execution of the financial risk application

bandwidth for moving data to the GPUs is reduced in practice. However, in the case of our application the initial data size is 4 GB and when this data is shared among four GPUs the data transferred to each GPU is larger than 10 MB. Hence, the data transfer to the GPUs is performed at full bandwidth.

A closer look at the application reveals that the YET data structure (4 GB) presented in Section 4.4 is uniformly split between the GPUs for computations. However, the $ELTs$ and PF data structures (120 MB and 4 MB) are not split between the GPUs, instead are transferred fully to each GPU. Consequently, the total data movement to GPUs increases which is shown in Figure 4.7. Excluding the $ELTs$, the data that is not split between the GPUs is less than 10 MB resulting in a lower bandwidth for transferring this data requiring an additional 2.6 milliseconds. However, this cannot fully account for sub-linear performance.

One important reason for the degradation of performance is data transfers to all GPUs are concurrently performed. Although each GPU is located in a different PCIe link, all data is extracted from main memory, which results in a bottleneck. This memory bottleneck is highlighted in Figure 4.8, which shows the bandwidth attained for each individual data copy when several data transfers are carried out concurrently to different destination GPUs by a single memory controller.

We summarise that for the financial risk application executing on multiple GPUs data transfers do not scale perfectly as the computations for two reasons. Firstly, there are input data structures that cannot be split between the GPUs and need to be copied onto

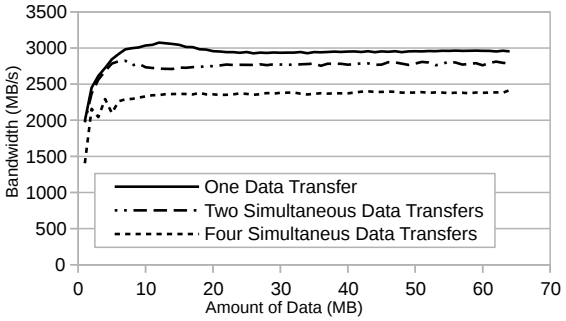


FIGURE 4.8: Attained bandwidth when concurrent data transfers to GPUs are performed. Source data is located in the same memory bank.

each GPU creating an overhead. Secondly, concurrent data transfers from the CPU main memory to GPUs result in a bottleneck at the memory controller.

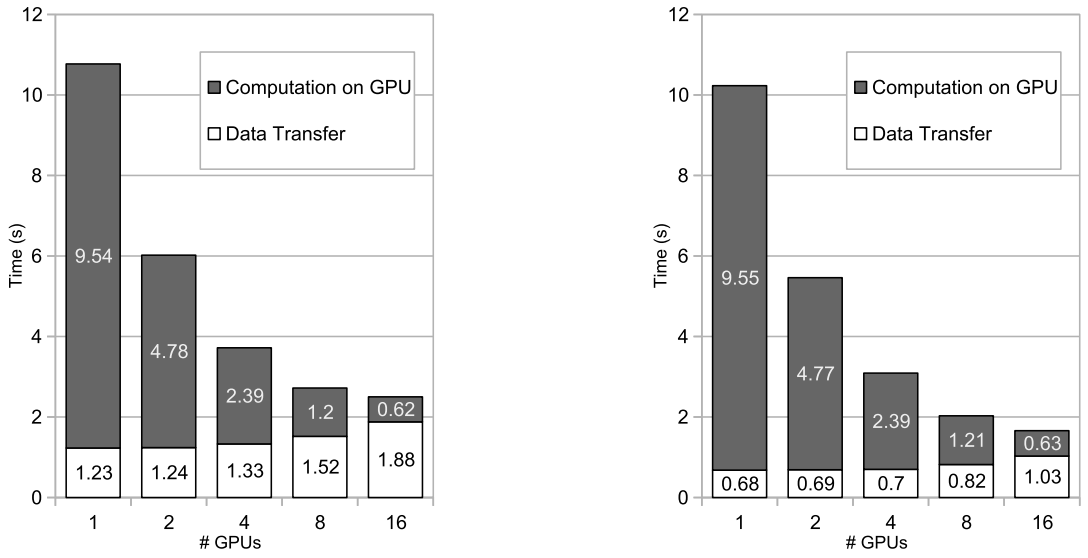
4.5.3 Reducing Execution Time Using rCUDA

Current servers are constrained in the number of GPUs that can be accommodated on them[1]. We believe remote GPU virtualisation (in this research rCUDA is employed) is an appropriate mechanism to make a large number of GPUs available to an application. Figure 4.9(a) and Figure 4.9(b) present the performance of the application using the QDR InfiniBand and the FDR InfiniBand networks respectively for up to 16 GPUs.

Figure 4.9 indicates that the computation times when using rCUDA on 1, 2, and 4 GPUs are the same as shown in Figure 4.6 using CUDA. This is expected given that the computation time on the GPU is independent of whether it is on the same node as the application or on a remote node. With increasing number of GPUs there is perfect scalability. When 16 GPUs are employed, the computation time is less than one second (0.62 seconds) making it possible to do an industry size simulation in real-time.

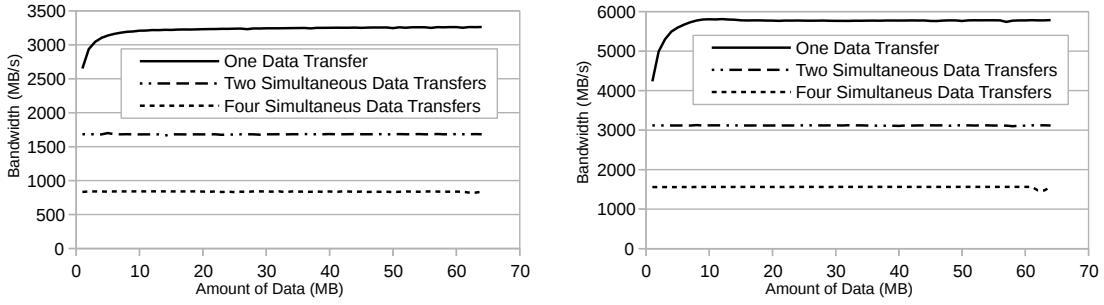
Two observations are made regarding data transfers. Firstly, when one remote GPU is used, the data transfer time using rCUDA is better than using CUDA (CUDA requires 1.378 seconds whereas rCUDA takes 1.23 seconds with QDR InfiniBand and 0.68 seconds with FDR InfiniBand). This lower transfer time as considered in Figure 4.5(c) is

[1]Manufacturers, such as Cirrascale and Supermicro, have integrated up to 8 GPU cards in a single server. However, these are exceptions and costly options. Moreover, there are performance bottlenecks since the GPUs are usually grouped as a set of four cards that share a single PCIe x16 link with a processor socket. This results in slower communication between main memory and the GPUs. Performance is further degraded when a GPU card comprises multiple devices.



(a) On QDR InfiniBand (b) On FDR InfiniBand

FIGURE 4.9: Scalability of the financial risk application when executed with rCUDA.

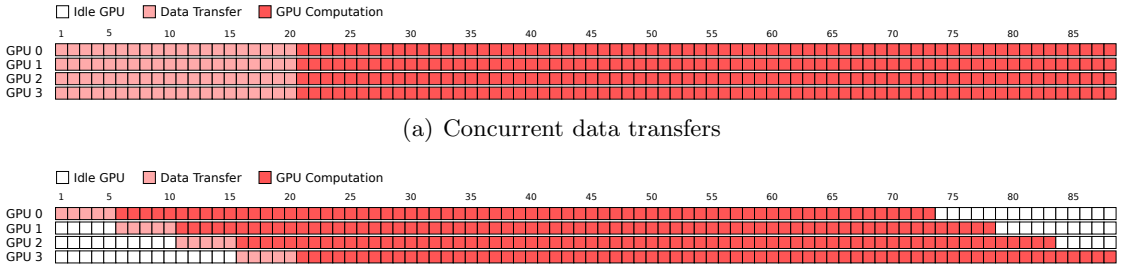


(a) On QDR InfiniBand (b) On FDR InfiniBand

FIGURE 4.10: Bandwidth attained for multiple data transfers concurrently to different remote GPUs using rCUDA.

because rCUDA obtains more bandwidth than CUDA by using pageable memory. The improvement of communication performance is seen in Figure 4.9(b) for 2 GPUs.

Secondly, data transfer using rCUDA follows a different trend to CUDA. For CUDA the data transfer times to each GPU reduced as the number of GPUs increased (refer Figure 4.6). On the contrary, rCUDA time increases when both QDR and FDR InfiniBand are used. This is not surprising since the reasons for sub-linear scalability of data transfer time considered in the previous section is applicable for rCUDA. In this case, the bandwidth bottleneck is the InfiniBand card in the cluster node executing the application, which is a single communication link for all the GPUs. This bottleneck is highlighted in Figure 4.10.



(a) Concurrent data transfers

(b) Sequential data transfers

FIGURE 4.11: Communication approaches for transferring data to GPUs.

Figure 4.10 shows the bandwidth achieved for individual data transfer to a different remote GPU when multiple transfers are executed concurrently. The bandwidth for each transfer is proportional to the number of data movement operations in progress. In addition to the previous observations that result in an increase of data transfer times, there are a large number of `cudaMalloc()` functions that are invoked prior to the data transfer (the memory allocation time is included in the data transfer time). In rCUDA, memory allocations for a large number of data structures on remote GPUs requires 2.7 milliseconds with FDR InfiniBand (compared to 1.7 milliseconds in CUDA on a local GPU) and 2.67 milliseconds with QDR InfiniBand (lower time due to low latency, despite reduced bandwidth [32]). Therefore, when a large number of GPUs are used by an application the time required for memory allocations can increase up to 43.2 milliseconds for 16 remote GPUs; this is 4.2% of the total data transfer time.

The use of rCUDA allows to leverage a large number of GPUs to speed up the application despite poor performance for data transfers. The total execution time is reduced from 2.86 seconds when using local GPUs on CUDA to 1.66 seconds when using remote GPUs on rCUDA. Reducing the total execution time enables the application to provide a solution in real-time.

4.5.4 Mitigating the Impact of Data Transfers in rCUDA

In this section, we consider two data transfer modes, namely concurrent and sequential, and further develop an approach based on multi-tenant GPUs in rCUDA.

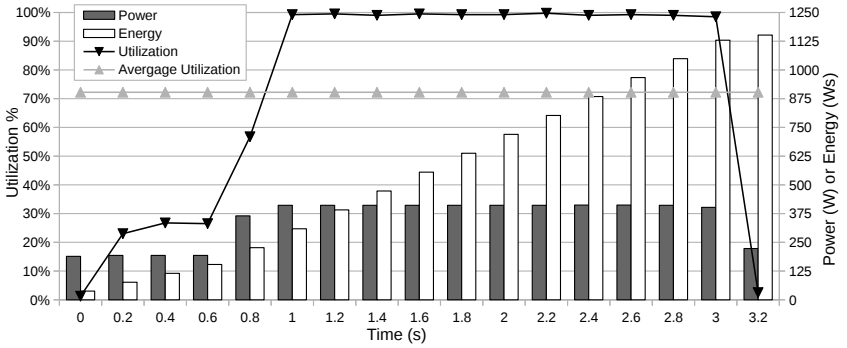
4.5.4.1 Concurrent vs Sequential Data Transfers

Figure 4.11(a) shows the life cycle of execution of a real application using rCUDA with four remote GPUs and FDR InfiniBand. Each cell represents execution time of 35 milliseconds. This corresponds to the four GPU execution shown in Figure 4.9(b). The same amount of data is moved to the four GPUs concurrently by interleaving across the network and the remote GPUs start computations at the same time approximately. However, from Figure 4.10 it was noted that the bandwidth achieved is inversely proportional to the number of multiple data transfers concurrently performed which results in degrading performance.

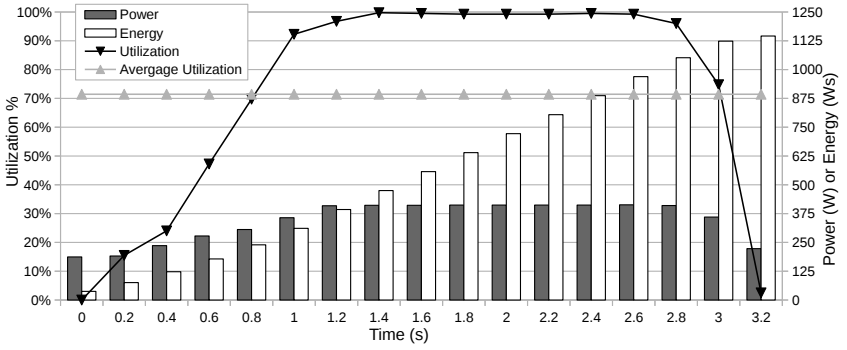
An alternate method is shown in Figure 4.11(b). Data to the first GPU is transferred without sharing the bandwidth for the remaining three data streams. Since there is no competition for bandwidth it only takes a quarter of the time required when data is concurrently transferred (shown in Figure 4.11(a)). Computations on the first GPU start while data is transferred to the second GPU. In this manner, data transfer is performed on fully available network bandwidth. This is referred to as the sequential data transfer method.

Data is transferred at full network bandwidth and there is an overlap with GPU computations in the sequential data transfer approach. However, it is noted that the execution time is not reduced since the fourth GPU begins its computations when it would in concurrent data transfers. Figure 4.12 shows the GPU utilisation, power and energy consumption of concurrent and sequential data transfers to GPUs. The average values of the four GPUs considered in Figure 4.11 are used. The Y-axis on the left indicates GPU utilisation and the Y-axis on the right shows power (in Watts) and energy (in Watts per second, denoted as Ws in the figure) consumed. The power and energy of GPUs are measured instead of the cluster since multiple GPU configurations (n GPUs per node) could be employed, which results in different energy measurements. There are no gains in the energy consumed and very little difference in GPU utilisation for both concurrent and sequential transfers.

Regardless, in this research sequential data transfer is foundational in developing an optimised approach for executing the application using remote GPUs which is based on multi-tenancy of virtual GPUs.

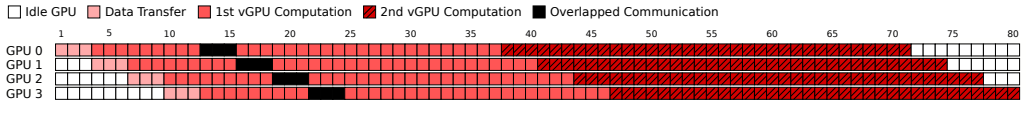


(a) Concurrent

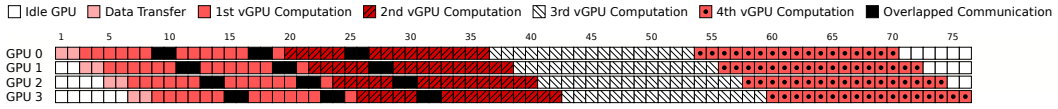


(b) Sequential

FIGURE 4.12: GPU utilisation, power and energy consumption of concurrent and sequential data transfers to GPUs considered in Figure 4.11



(a) 2 vGPUs per GPU



(b) 4 vGPUs per GPU

FIGURE 4.13: Sequential data copies with several vGPUs per GPU.

4.5.4.2 Multi-tenancy Approach

The key concept of the multi-tenancy approach is based on the fact that current GPUs perform kernel executions and DMA (Direct Memory Access) operations concurrently. If it were possible to move data to a GPU the same time it was executing a kernel, there could be gains in further improving the performance of the executing application.

This can be facilitated by a multi-tenancy approach in which a number of remote GPUs (or virtual GPUs referred to as vGPUs) reside on or are mapped onto the same physical

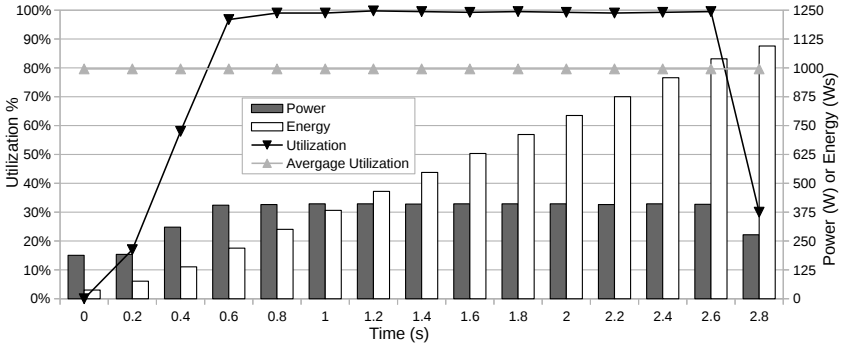
GPU (pGPU)[2]. Figure 4.13 shows the concept of multi-tenancy when 2 and 4 vGPUs are mapped to a pGPU.

When 2 vGPUs are mapped on to a pGPU as shown in Figure 4.13(a) 8 GPUs are available to the application (4 pGPUs are used). Input data will be split such that 8 GPUs will be used for computations. The initial data transfer is shown as *"Data Transfer"* followed by computations by the first vGPU labelled as *"1st vGPU Computation"*. After transferring data in the 12th time step, there are four more vGPUs that will require their input data. Data transferred to the remaining four vGPUs beginning at time step 13 are overlapped with the computations of the first four vGPUs. Since two vGPUs are mapped onto a single pGPU, computations of both vGPUs cannot progress in parallel as they belong to different GPU contexts. Therefore, the NVIDIA driver executes them sequentially (using as many GPU resources required by each kernel). So the second kernel must wait until the execution of the first kernel is completed.

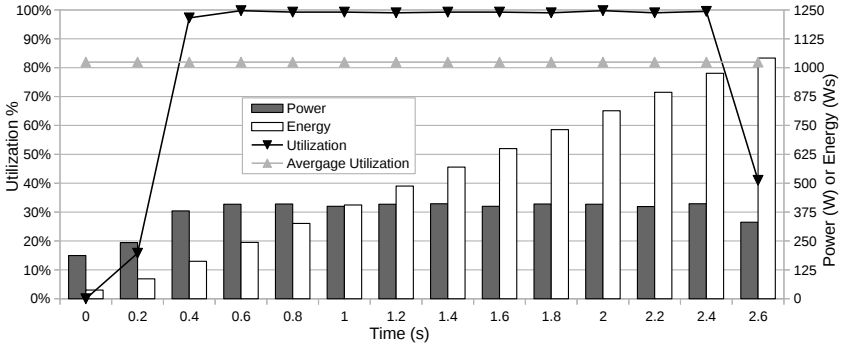
Two key observations are made from multi-tenant executions. Firstly, the total execution time has reduced in contrast to the execution life cycle presented in Figure 4.11(b) although the same hardware resources are used. The application completed execution in time step 80 using 2 vGPUs per pGPU compared to time step 88 when no multi-tenancy is employed. The time that each GPU computes is exactly the same. The time saved is because of the overlap between computations and data transfers of multiple vGPUs on the same pGPU. In Figure 4.11(b) data transfers overlapped with computations of other pGPUs but there were no overlaps on the same GPU.

Secondly, the data transfer time takes longer when more vGPUs are employed. In Figure 4.11(b), data is transferred completely to all GPUs at time step 20, whereas in Figure 4.13(a), the input data arrives at time step 24. The reasons for longer data transfer times have been considered in the previous section. Despite the larger data transfer time, the total execution time gains since there is an overlap between computation and data movement.

[2]Multi-tenancy is achieved on rCUDA by setting two environment variables prior to application execution, namely `RCUDA_DEVICE_COUNT` and `RCUDA_DEVICE_j`. The first variable indicates the number of GPUs accessible to the application. The second variable indicates the cluster node in which the j^{th} GPU is located. For example, "`export RCUDA_DEVICE_COUNT=2`" when 2 GPUs are assigned to the application and "`export RCUDA_DEVICE_0=192.168.0.1`" and "`export RCUDA_DEVICE_1=192.168.0.2`". The server of the `RCUDA_DEVICE_j` variables need to point to the same node. Hence, the application does not require to be modified to accommodate multi-tenancy using rCUDA.



(a) 2 vGPUs per pGPU



(b) 4 vGPUs per pGPU

FIGURE 4.14: GPU utilisation, power and energy consumption of the multi-tenancy approach considered in Figure 4.13.

Figure 4.13(b) shows the use of 16 vGPUs mapped on to 4 pGPUs. The execution time is further reduced due to the larger overlap between computation and data transfers when compared to 2 vGPUs residing on a single pGPU. Again the time for computing is the same on each physical GPU but the data copying time has increased. The overall execution time is further reduced to 76 time steps.

Multi-tenancy can be analysed from the perspective of energy required to complete the execution of the application. Figure 4.14 shows the energy consumed during the execution of the application along with the utilization of the physical GPU. The multi-tenancy energy consumption is lower than sequential communications without an overlap between data transfers and computations on the same GPU seen in Figure 4.12. The energy consumed is 1145 Watts per second without using multi-tenancy and 1094 and 1041 Watts per second when 2 and 4 vGPUs are tenants on a pGPU, respectively. It is observed that GPU utilisation increases in the multi-tenancy approach. The average GPU utilisation rises from 71.44% without multi-tenancy up to 79.65% for 2 vGPUs per pGPU and up to 81.93% when 4 vGPUs are mapped on to a pGPU.

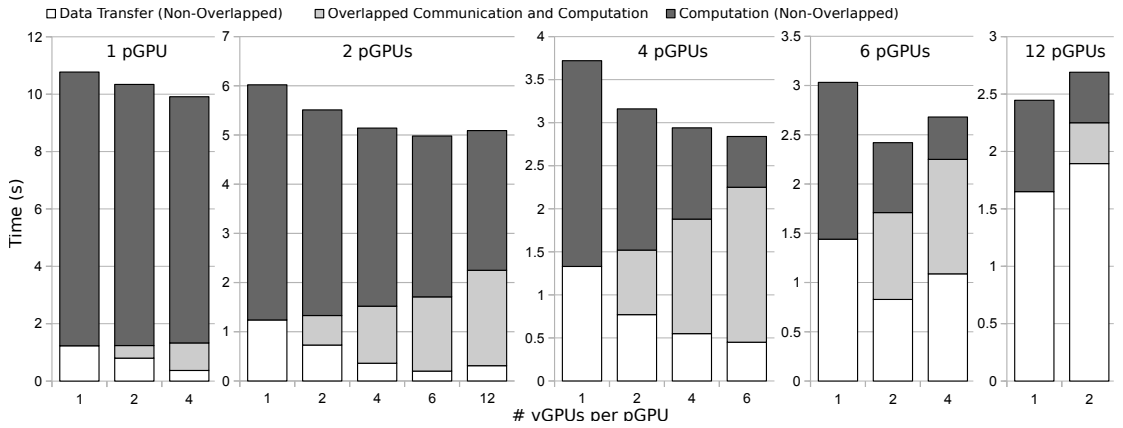


FIGURE 4.15: Application performance for different combinations of pGPUs and vGPUs using QDR InfiniBand

In short, multi-tenancy allows for data transfers to be overlapped with computations on the same GPUs thereby reducing total execution time of the financial risk application. Furthermore, the energy required to execute the application is reduced and the GPU utilisation is increased.

4.5.5 Performance Analysis Using Multi-tenancy

An analysis of the application performance as measured by execution time is presented in this section. The cluster nodes in our experimental set up have 12 cores (up to 24 threads with hyper-threading) and therefore we use a maximum of 24 vGPUs (to avoid any noise due to CPU overhead). Up to 12 pGPUs will be used to map the vGPUs.

Figure 4.15 and Figure 4.16 show the time taken for data transfer and computation for varying pGPUs when the rCUDA framework is used over QDR and FDR InfiniBand. The 'Overlapped data transfer and computation' label denotes that data transfers and computation are carried out concurrently on the same pGPU. The behaviour of the application is as expected. Multi-tenancy with sequential transfers allows for overlapping computations and data movement on the same pGPU, thus reducing the execution time. When QDR InfiniBand is used, time for data transfer without overlaps with communication is reduced up to 70%, 84%, 66%, and 42% when vGPUs are mapped to 1, 2, 4, and 6 pGPUs, respectively. In the case of FDR InfiniBand, the same time is 65%, 77%, 57%, and 56%. Consequently, the total power consumed is reduced but not indicated on the graph.

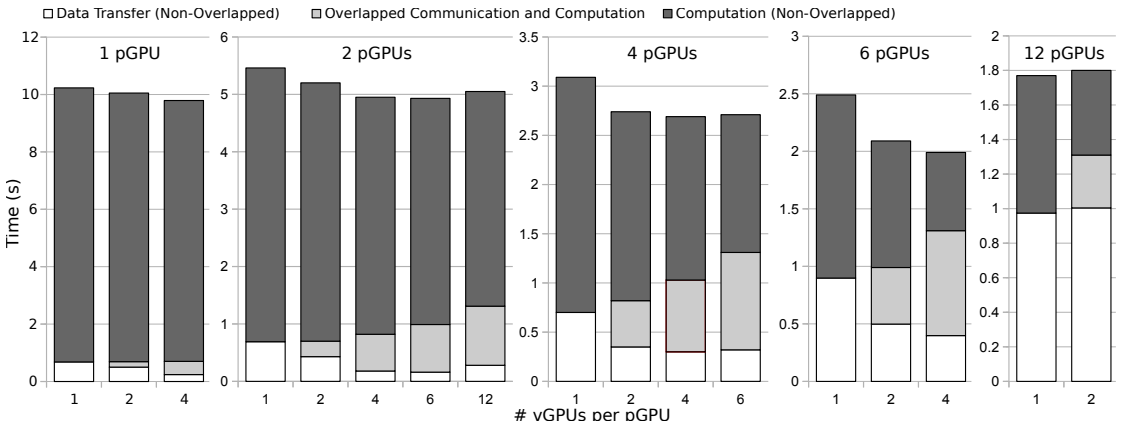


FIGURE 4.16: Application performance for different combinations of pGPUs and vGPUs using FDR InfiniBand

It is noted that when 12 pGPUs are used the data transfer times are not reduced further because (i) the execution time decreases with more pGPUs, and (ii) the data transfer time increases when more vGPUs are used allowing for little overlap between data transfers and computation on the same pGPU. This necessitates the need for determining the effective combination of pGPUs and vGPUs by estimating application perfomance both in terms of execution time and energy consumption.

4.5.6 Modelling Multi-tenancy for Performance and Energy Estimation

An important challenge is to automatically determine the best multi-tenancy configuration for a deployment that can maximise performance (minimising execution time), but at the same time minimise the energy consumed.

4.5.6.1 Performance Model

We firstly consider a basic model to account for execution time of the application when sequential data transfers are used with rCUDA, but without exploiting multi-tenancy. Subsequently, the model is optimised to take multi-tenancy into account. The model is then applied in the context of the hardware (NVIDIA Tesla K20 GPUs with QDR and FDR InfiniBand) we have employed in this research.

The total execution time depends on: (i) time for transferring data and (ii) time for computing on the GPUs as shown in Equation 4.4, which inherently depends on the

number of GPUs (pGPUs or vGPUs) available to the application.

$$TotalExecutionTime = T_{transfer}(\#GPUs) + T_{computation}(\#GPUs) \tag{4.4}$$

Since there is perfect scalability for the computation times on the GPU (Section 4.5.2 and Section 4.5.3), the time required for computations by a given number of GPUs can be obtained as shown in Equation 4.5.

$$T_{computation}(\#GPUs) = ComputationTime\_1pGPU \ / \ \#GPUs \tag{4.5}$$

The time to transfer the input data to all GPUs is shown in Equation 4.6. The time taken to allocate memory on each GPU using `cudaMalloc()` and the time for moving small and large data structures to the GPUs are taken into account. Different data sizes achieve varying network bandwidth (Figure 4.5(c)). To simplify the equation, the time to transfer data structures smaller than 100 bytes is denoted as $T_{small\_transfers}$[3]

$$\begin{aligned} T_{transfer}(\#GPUs) = \#GPUs \ * (&T_{cudaMalloc} + T_{small\_transfers} \\ &+ T_{transfer\_4MB} + T_{transfer\_120MB}) \\ &+ T_{transfer\_4GB} \end{aligned} \tag{4.6}$$

When multi-tenancy is taken into account there is an overlap between data transfers and computations on the same pGPU which reduces the total execution time. As shown in Figure 4.13(a), when 2 vGPUs are mapped onto a single pGPU, the time for data transfer is the time taken to move the first chunks of data to the pGPUs (until the completion of time step 12). The time for moving the remaining data chunks are not accounted for since it is overlapped by computation time. This is captured in Equation 4.7.

[3]Data structures smaller than 100 bytes achieve the same bandwidth and are therefore grouped together. The InfiniBand frame size is typically 2 KB, which will be sent to the GPU in all cases where data is smaller than 100 bytes.

TABLE 4.2: Time in seconds for GPU memory allocation and data transfer tasks of the financial risk application

Parameter	QDR	FDR
$ComputationTime\_1pGPU$	9.55	
$T_{cudaMalloc}$	0.00267	0.0027
$T_{small\_transfers}$	0.0048	0.0028
$T_{transfer\_4MB}$	0.00133	0.00079
$T_{transfer\_120MB}$	0.036	0.0205
$T_{transfer\_4GB}$	1.171	0.67

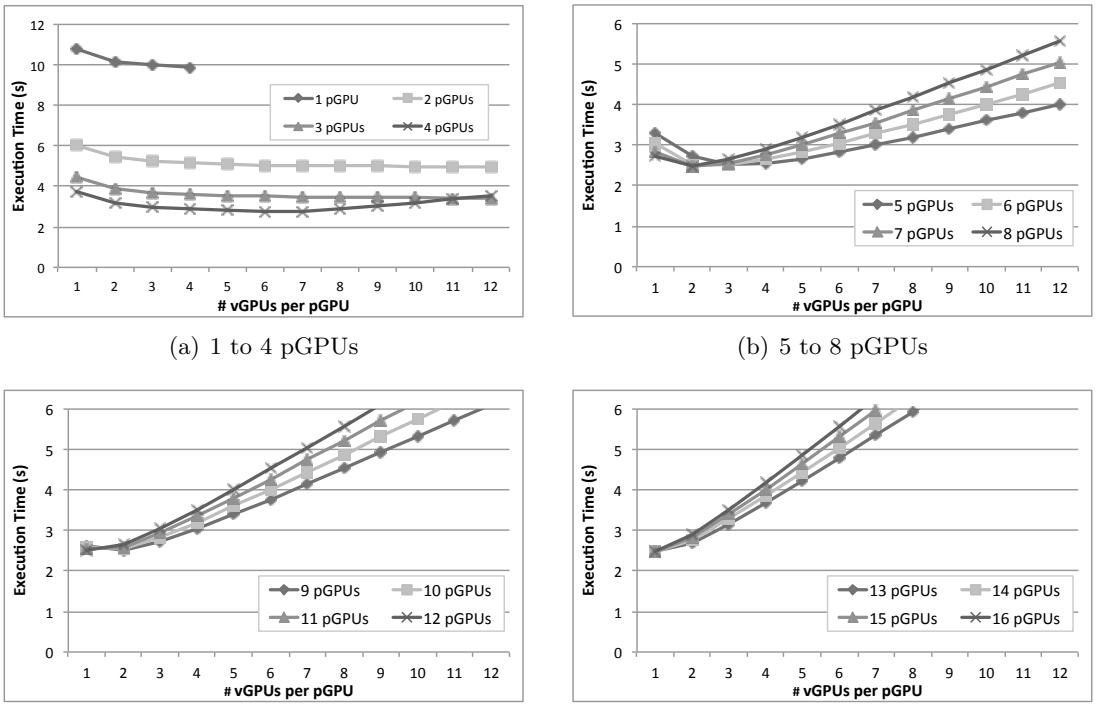
$$ExecTime\_Multitenancy_{fully\_overlapped} = T_{transfer}(\#vGPUs) \ / \ vGPUs\_per\_pGPU + vGPUs\_per\_pGPU * T_{computation}(\#vGPUs) \tag{4.7}$$

If a very large number of vGPUs are used, then all data transfer times may not be overlapped with computation times. This can happen when the computation on the vGPU is not long enough to overlap data transfers to the pGPU and the computations on it. In this case, the total execution time depends on the time required to copy data to all the vGPUs and is shown in Equation 4.8.

$$ExecTime\_Multitenancy_{not\_fully\_overlapped} = T_{transfer}(\#vGPUs) + T_{computation}(\#vGPUs) \tag{4.8}$$

As shown in Equation 4.9 the maximum value from Equation 4.7 and Equation 4.8 determines whether the application has significant overlaps between data transfer and computations.

$$ExecTime\_Multitenancy = MAX(ExecTime\_Multitenancy_{fully\_overlapped}, ExecTime\_Multitenancy_{not\_fully\_overlapped}) \tag{4.9}$$



(a) 1 to 4 pGPUs

(b) 5 to 8 pGPUs

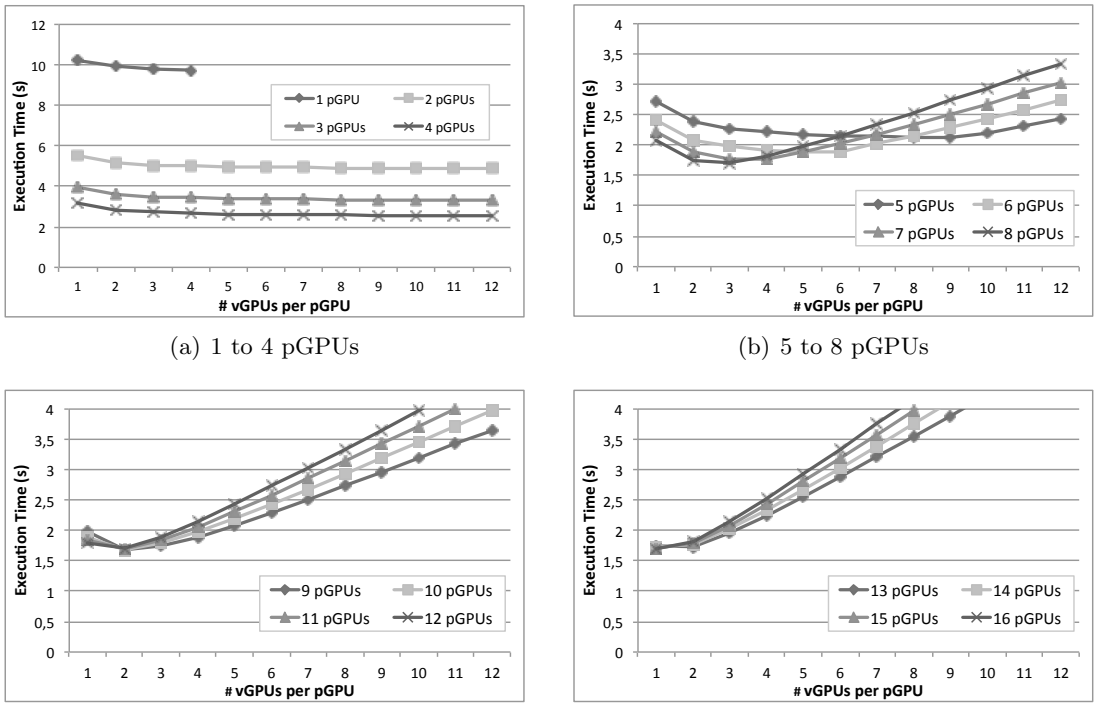
(c) 9 to 12 pGPUs

(d) 13 to 16 pGPUs

FIGURE 4.17: Results from performance model for QDR InfiniBand

Table 4.2 shows actual values of the model for the experimental platform used in this research.

Figure 4.17 and Figure 4.18 use these values in Equation 4.9 for 1 to 16 pGPUs and up to 12 vGPUs per pGPU. The combinations of pGPUs and vGPUs that require the lowest execution time can be explored in this space. The estimated execution times are grouped for 1 to 4 pGPUs, 5 to 8 pGPUs, 9 to 12 pGPUs, and 13 to 16 pGPUs. In Figure 4.17(a) and Figure 4.18(a), for one pGPU up to 4 vGPUs can be used. The total memory on the Tesla K20 devices is 4799 MB (from the `nvidia-smi` command), which is exhausted by more than 4 vGPUs (total memory size consumed by the application on 4 vGPUs is 4484 MB). It is inferred from the figures that a large number of vGPU has detrimental effect on performance due to the overheads in data movements. Using QDR InfiniBand the model predicts a saturation sooner than FDR InfiniBand because of the overhead of data transfers due to a lower bandwidth available on the QDR network. The optimal deployment configuration of the application is 7 pGPUs with 2 vGPUs per pGPU and 9 pGPUs with 2 vGPUs per pGPU using QDR InfiniBand and FDR InfiniBand respectively.



(a) 1 to 4 pGPUs

(b) 5 to 8 pGPUs

(c) 9 to 12 pGPUs

(d) 13 to 16 pGPUs

FIGURE 4.18: Results from performance model for FDR InfiniBand

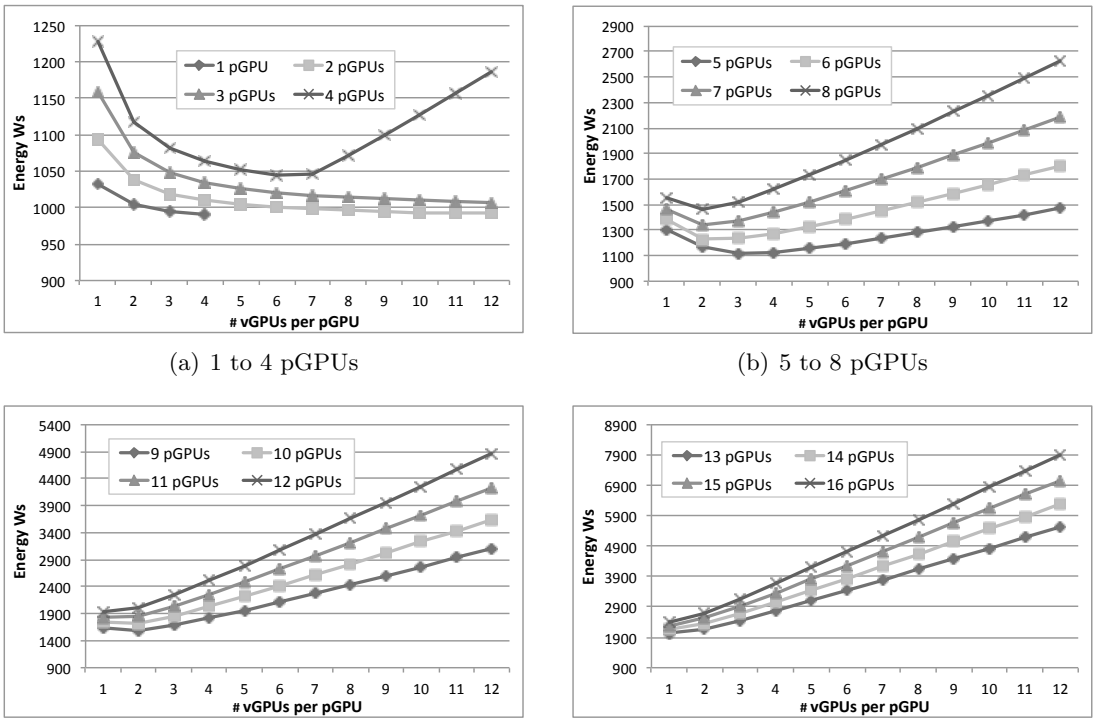
4.5.6.2 Energy Model

The amount of energy required to execute the application is modelled in this section. From Figure 4.13 it is inferred that a GPU can be in the following four different states: (1) idle, (2) receive data, but no computations, (3) receive data and compute simultaneously, and (4) compute, but no data to receive.

Power is measured by querying `nvidia-smi` every 200 milliseconds. The power required by the GPU in the first two states is the same. The NVIDIA Tesla K20 device requires 47 Watts while idling[4] and receiving data. The GPU requires 102 Watts in the last two states.

Using the above power readings for the four GPU states along with total execution time obtained from Equation 4.9 an energy model is developed as shown in Equation 4.10. The energy required by the GPU for computations (time spent on computations is obtained

[4]The idle state in Figure 4.13 is distinguished from the commonly known "idle" state. In Figure 4.13, the GPU has already been assigned to the application and therefore has been initialised by the GPU driver(this requires approximately 1.3 seconds in CUDA). After initialisation, the GPU does not perform any task, but actively waits for commands. In the commonly known "idle" state, the GPU is not assigned to an application and is not initialised by the driver. In this state, the Tesla K20 GPU requires 25 Watts.



(a) 1 to 4 pGPUs

(b) 5 to 8 pGPUs

(c) 9 to 12 pGPUs

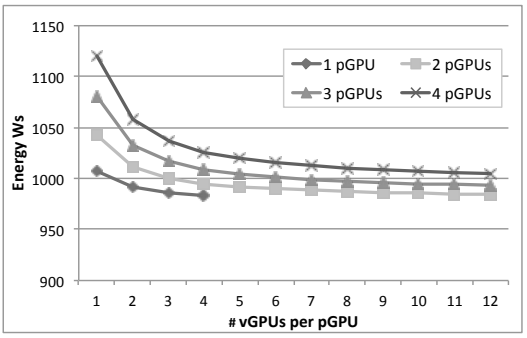
(d) 13 to 16 pGPUs

FIGURE 4.19: Results from energy model for QDR InfiniBand

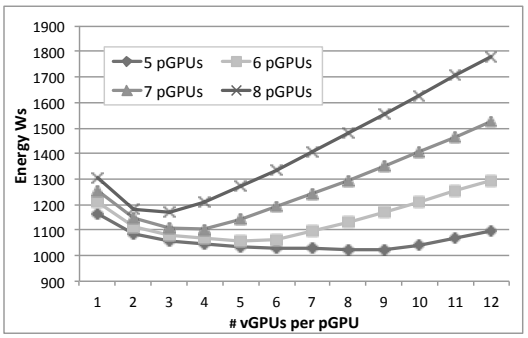
from Equation 4.5) is eliminated to obtain the energy spent in the first and second states. The computation time on the pGPUs is $vGPUs\_per\_pGPUs * T_{computation}(\#vGPUs)$.

$$TotalEnergy = \#pGPUs * (T_{computation}(\#pGPUs) * 102\ Watts + (ExecTime\_Multitenancy - T_{computation}(\#pGPUs)) * 47\ Watts) \quad (4.10)$$

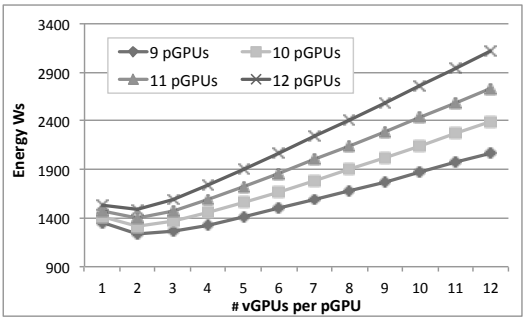
Figure 4.19 and Figure 4.20 present the results of the energy model from Equation 4.10. It is noted that an energy efficient deployment is obtained using 4 vGPUs on 1 pGPU for both QDR InfiniBand and FDR InfiniBand. This is as expected given that the least amount of hardware is employed. However, there is a trade off since the lowest execution times are not obtained in this configuration. In Figure 4.21 and Figure 4.22, an alternate space (*energy* $*$ *execution time*) is explored to find configurations that can maximise performance and minimise energy consumption.



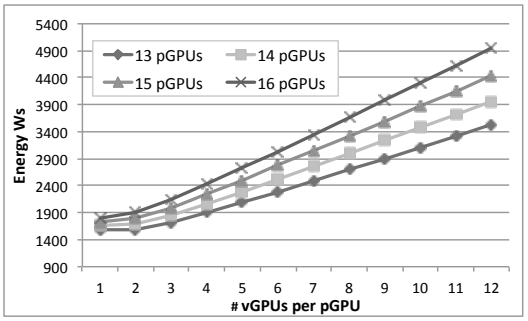
(a) 1 to 4 pGPUs



(b) 5 to 8 pGPUs

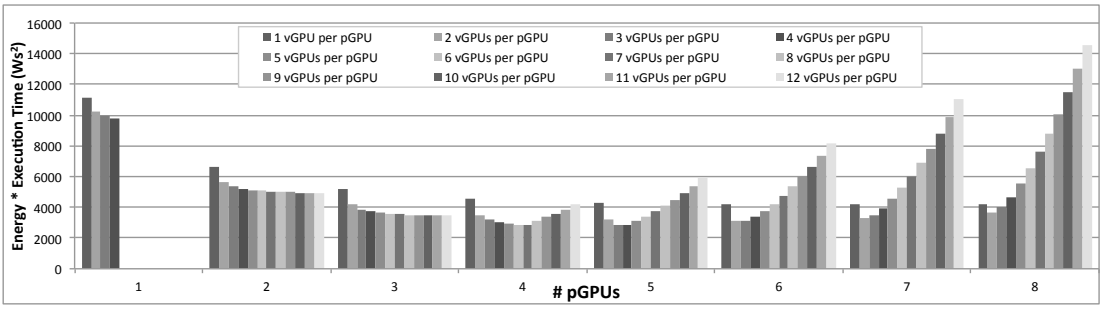


(c) 9 to 12 pGPUs

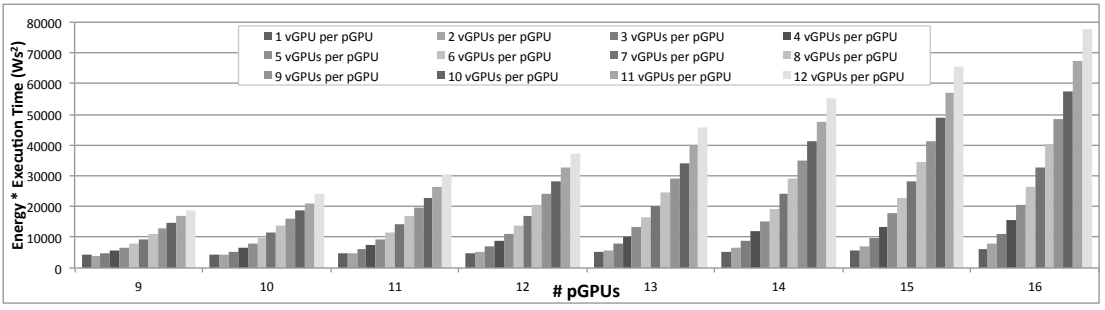


(d) 13 to 16 pGPUs

FIGURE 4.20: Results from energy model for FDR InfiniBand

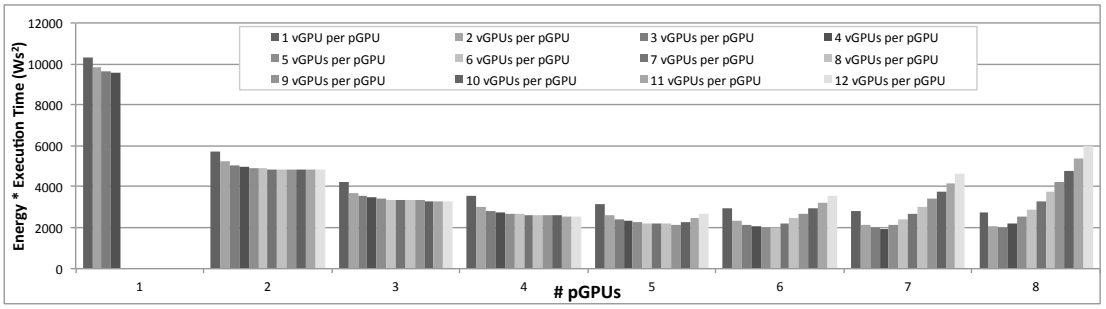


(a) 1 to 8 pGPUs

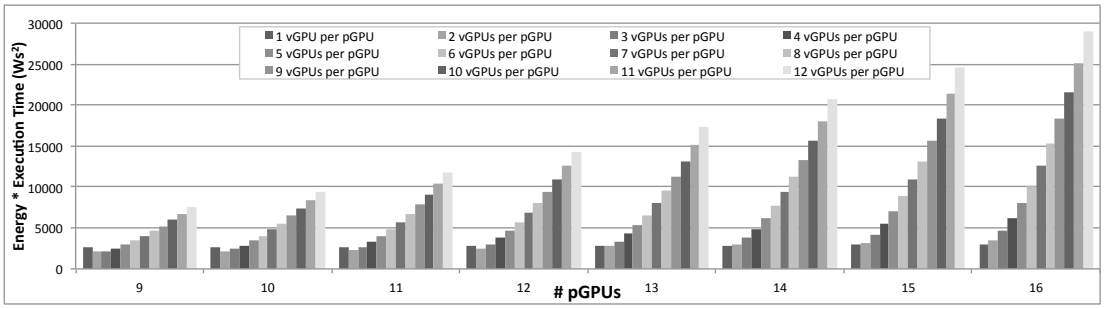


(b) 9 to 16 pGPUs

FIGURE 4.21: Combined space of energy and execution time using QDR InfiniBand



(a) 1 to 8 pGPUs



(b) 9 to 16 pGPUs

FIGURE 4.22: Combined space of energy and execution time using FDR InfiniBand

4.6 Conclusions

In this paper, we have demonstrated the benefits of virtual GPUs for an application. Single tenancy (using one virtual GPU on a single physical GPU) and multi-tenancy (using a number of virtual GPUs on a physical GPU) were explored in this context. Concurrent and sequential data transfer models were considered. We hypothesised that multi-tenancy can improve the performance of the application. To validate the hypothesis the application was executed using rCUDA (remote CUDA), a framework that virtualises GPUs in a High-Performance Computing (HPC) cluster and provides remote GPUs to nodes that require acceleration on demand. Experimental results indicate that multi-tenant virtual GPUs with sequential data transfers optimise the performance of the application with less hardware when compared to single tenancy.

References

[1] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium*

on Field Programmable Gate Arrays, pages 115–124, 2010.

[2] Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 91–100, 2012.

[3] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–108, 2012.

[4] Dipanjan Sengupta, Raghavendra Belapure, and Karsten Schwan. Multi-tenancy on GPGPU-based servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, pages 3–10, 2013.

[5] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and performance characterization of computational kernels on GPU. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, pages 221–228, 2010.

[6] S. Iserte, A. Castello, R. Mayo, E.S. Quintana-Orti, F. Silla, J. Duato, C. Reano, and J. Prades. Slurm support for remote GPU virtualization: Implementation and performance study. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 318–325, 2014.

[7] Blesson Varghese, Javier Prades, Carlos Reano, and Federico Silla. Acceleration-as-a-service: Exploiting virtualised GPUs for a financial application. In *Proceedings of the 11th IEEE International Conference on eScience*, pages 47–56, 2015.

[8] A. J. Pena, C. Reano, F. Silla, R. Mayo, E. S. Quintana-Orti, and J. Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*, 40:574–588, 12/2014 2014.

[9] A. Srinivasan. Parallel and distributed computing issues in pricing financial derivatives through quasi monte carlo. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, 2002.

[10] K. Huang and R.K. Thulasiram. Parallel algorithm for pricing american asian options with multi-dimensional assets. In *High Performance Computing Systems*

and Applications, 2005. HPCS 2005. 19th International Symposium on, pages 177–185, 2005.

[11] C. Bekas, A. Curioni, and I. Fedulova. Low cost high performance uncertainty quantification. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, 2009.

[12] D. Daly, Kyung Dong Ryu, and J.E. Moreira. Multi-variate finance kernels in the blue gene supercomputer. In *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pages 1–7, 2008.

[13] V. Agarwal, Lurng-Kuo Liu, and D.A. Bader. Financial modeling on the cell broadband engine. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008.

[14] Ciprian Docan, Manish Parashar, and Christopher Marty. Advanced risk analytics on the cell broadband engine. pages 1–8, 2009.

[15] A. Irturk, B. Benson, N. Laptev, and R. Kastner. FPGA acceleration of mean variance framework for optimal asset allocation. In *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pages 1–8, 2008.

[16] D.B. Thomas. Acceleration of financial monte-carlo simulations using FPGAs. In *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*, pages 1–6, 2010.

[17] L.A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. Pricing derivatives on graphics processing units using monte carlo simulation. *Concurrency and Computation: Practice and Experience*, 26(9):1679–1697, 2014.

[18] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. An efficient graphics processing unit-based parallel algorithm for pricing multi-asset american options. *Concurrency and Computation: Practice and Experience*, 24(8):849–866, 2012.

[19] *CUDA API Reference Manual 6.5*, 2014.

[20] *OpenCL 2.0 Specification*, 2013.

[21] Tyng Yeu Liang and Yu Wei Chang. Gridcuda: A grid-enabled CUDA programming toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, 2011.

[22] 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, 2009.

[23] 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*, pages 17–24, 2009.

[24] 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*, pages 379–391, 2010.

[25] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. DS-CUDA: A middleware to use many GPUs in the cloud environment. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1207–1214, 2012.

[26] NVIDIA. *The NVIDIA GPU Computing SDK Version 5.5*, 2013.

[27] iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. `https://github.com/esnet/iperf`, 2015.

[28] A.K. Bahl, O. Baltzer, A. Rau-Chaplin, and B. Varghese. Parallel simulations for analysing portfolios of catastrophic event risk. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1176–1184, 2012.

[29] Blesson Varghese. The hardware accelerator debate: A financial risk case study using many-core computing. *Computers & Electrical Engineering*, 46:157–175, 2015.

[30] G. Woo. Natural catastrophe probable maximum loss. *British Actuarial Journal*, 8:943–959, 2002.

[31] A. A. Gaivoronski and G. Pflug. Value-at-risk in portfolio optimization: Properties and computational approach. 7(2):1–31, 2005.

[32] C. Reano, R. Mayo, E.S. Quintana-Orti, F. Silla, J. Duato, and A.J. Pena. Influence of InfiniBand FDR on the performance of remote GPU virtualization. In *IEEE International Conference on Cluster Computing*, pages 1–8, 2013.

Chapter 5

Maximizing resource usage in Multi-fold Molecular Dynamics with rCUDA

Javier Prades, Baldomero Imbernón, Carlos Reaño, Jorge Peña-García, José Pedro Cerón-Carrasco, Federico Silla, Horacio Pérez Sánchez. **The International Journal of High Performance Computing Applications** - Volume: 34 - Issue: 1 - Jan. 1 2020 - Pages 5 - 19
https://doi.org/10.1177/1094342019857131

Abstract

The full-understanding of the dynamics of molecular systems at the atomic scale is of great relevance in the fields of chemistry, physics, materials science and drug discovery just to name a few. Molecular dynamics (MD) is a widely used computer tool for simulating the dynamical behavior of molecules. However, the computational horsepower required by MD simulations is too high to obtain conclusive results in real world scenarios. This is mainly motivated by two factors: (1) the long execution time required by each MD simulation (usually in the nanosecond and microsecond scale, and beyond) and (2) the large number of simulations required in drug discovery to study the interactions between a large library of compounds and a given protein target. To deal with the former, Graphics Processing Units (GPUs) have come up into the scene. The latter has been traditionally approached by launching large amounts of simulations in computing clusters that may contain several GPUs on each node. However, GPUs are targeted as a single node that only runs one MD instance at a time, which translates into low GPU occupancy ratios and therefore low throughput. In this work, we propose a strategy to increase the overall throughput of MD simulations by increasing the GPU occupancy through virtualized GPUs. We use the rCUDA middleware as a tool to decouple GPUs from CPUs, and thus enabling multi-tenancy of the virtual GPUs. As a working test in the drug discovery field, we studied the binding process of a novel flavonol to DNA with the GROMACS MD package. Our results show that the use of rCUDA provides with a 1.21x speed-up factor compared to the CUDA counterpart version while requiring a similar power budget.

Keywords: Molecular dynamics, GPU virtualization, rCUDA, GROMACS, GPU.

5.1 Introduction

Molecular dynamics (MD) has been consolidated as a popular tool in theoretical studies in molecular sciences. MD tools solve Newton's equations of motion for a given molecular system, which sample atomic motions usually in the nanoseconds to microseconds and milliseconds scale. These simulations are becoming more accurate along with the development of improved force fields, making it possible to accurately study processes such as protein folding [1]. A MD simulation starts with a molecular configuration and a physical model, which includes details about how atomic interactions are modeled. After the simulation is carried out, the user obtains insightful conclusions studying and analyzing the trajectory. The computational horsepower required by MD simulations is overwhelming as they assess millions of interactions of particles during many time steps [2]. Indeed, the accuracy or realism of the result is directly related to the amount of sampling.

There are many software packages for developing MD simulations such as GROMACS [3], AMBER [4] or NAMD [5] just to mention a few. Indeed, the development of all of these standardized tools has democratized the use of MD, even for those who are not specialists in simulator development. Of particular interest to us is GROMACS, which is an open-source MD tool extensively used in chemistry, mainly (although not limited to) for the simulation of biomolecules. GROMACS has as a primary goal to achieve the highest simulation efficiency by offering several parallelization approaches at different levels; vectorization, multithreading and CPU-GPU (i.e., Graphics Processing Unit). Some previous works have been carried out to improve the performance of a single GROMACS execution by using these parallel techniques [2, 3, 6–8].

However, the use of MD simulations for answering real scientific problems, such as the discovery of new drugs, typically involves a large number of independent simulations that are executed in a large computing cluster by using a resource manager, or job scheduler, such as Slurm [9]. These resource managers allow a collection of heterogeneous resources to be shared among the jobs that are executed in the cluster. However, these resource managers are not designed to fully leverage GPUs because they do not allow the shared access (i.e., multi-tenancy) to them from different processes [10].

In this paper we make use of a multi-tenant virtual GPU strategy for increasing the throughput of a batch of independent GROMACS simulations. To that end, we use the rCUDA middleware [11], which enables remote concurrent use of CUDA-compatible GPUs. This middleware decouples GPUs from CPUs thus enabling virtual CUDA-compatible devices on machines without local GPUs, still delivering an acceptable performance. Moreover, the physical GPUs are concurrently shared among several GPU processes and therefore the GPU occupancy can be improved by running several different GPU processes at the same time [10]. In addition to leverage virtual GPU multi-tenancy in order to increase overall throughput of a batch of independent MD simulations, we also leverage CPU-based MD simulations concurrently executed with the virtual GPU-based simulations in order to further increase overall throughput. In this regard, we show that by properly tuning the amount of resources used by each MD simulation, overall throughput of a batch of MD simulations can be noticeably increased with respect to the use of traditional CPU-based or GPU-based approaches. A complementary, and preliminary, study to the work presented in this paper was already presented in [12]. Contrary to that preliminary work, in this paper we provide a more mature analysis of the multi-tenant virtual GPU strategy when applied to a bunch of independent GROMACS simulations by following a different approach. Additionally, a new molecular system is studied in this paper. In this regard, we also provide an insight to the problem from a purely biological perspective.

The rest of the paper is structured as follows. Next section provides the required background about MD. It also briefly describes the rCUDA middleware. Afterwards, our strategy to improve the throughput of MD simulations in large heterogeneous clusters is thoroughly introduced. Next, the bio-informatics problem addressed in this paper is described, followed by the experimental results that show how system throughput can be increased by making use of virtual GPU multi-tenancy. Next, analysis of MD results from the biological side and its validation is commented. The last section summarizes the conclusions of this study and provides some directions for future work.

5.2 Background

This section provides the necessary background on MD simulations as well as on the rCUDA remote GPU virtualization middleware.

5.2.1 MD in Drug Discovery

We draw on our description of Virtual Screening (VS) methods for drug discovery, which was previously given in [13–15]. VS methods are computational techniques used in several scientific areas, such as catalysts and energy materials [16], and mainly drug discovery [17], where experimental techniques can benefit from computational simulation.

VS methods search within libraries of small molecules that can potentially bind to a drug target, typically a protein receptor or enzyme, with high affinity. In some cases, they actually "dock" small molecules into the structures of macromolecular targets. Moreover, they look for (i.e., score) the optimal binding sites by providing a ranking of chemical compounds according to the estimated affinity or *scoring* [18]. In general, VS methods optimize *scoring functions*, which are mathematical models used to predict the strength of the non-covalent interaction between two molecules after docking [19]. Indeed, these candidate molecules will continue the drug discovery process road-map that goes from in-vitro studies to animal investigations and, eventually, to human trials [20].

Although VS methods have been used for many years and have identified several compounds to be used as approved drugs, VS has not yet fulfilled all its expectations. Neither the VS methods nor the scoring functions used are sufficiently accurate to identify high-affinity ligands reliably. To deal with large numbers of potential candidates (many databases comprise hundreds of thousands of ligands), VS methods must be very fast and still they would require a large amount of computing time for each ligand.

One recent approach to increase accuracy of VS methods is to use several methods in the pipeline, starting from high-speed and low accuracy methods such as molecular similarity, then post-filtering result using mid accuracy techniques such as molecular docking, and ending up with more accurate and informative structure-based techniques such as MD. In this work we will focus our discussion in the execution of VS calculations with GROMACS.

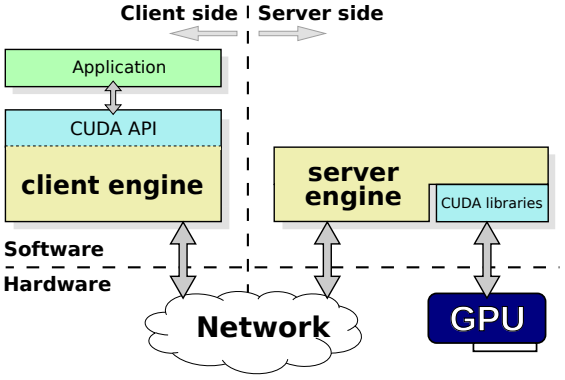


FIGURE 5.1: Architecture of the rCUDA middleware

5.2.2 rCUDA (remote CUDA)

Figure 5.1 depicts the architecture of the rCUDA middleware, which follows a client-server distributed approach. The client part of rCUDA is installed in the cluster node executing the application requesting GPU services, whereas the server side runs in the computer owning the actual GPU. The client side of the middleware offers the same application programming interface (API) as does the NVIDIA CUDA API. In this manner, the client receives a CUDA request from the accelerated application and appropriately processes and forwards it to the remote server. In the server node, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and provides the execution results to the server middleware. In turn, the server sends back the results to the client middleware, which forwards them to the initial application, which is not aware that its request has been served by a remote GPU instead of a local one.

rCUDA is binary compatible with CUDA 9.0 and implements the entire CUDA Runtime and Driver APIs (except for graphics functions). It also provides support for the libraries included within CUDA (cuDNN, cuBLAS, cuFFT, etc.). Additionally, it supports several underlying interconnection technologies by making use of a set of runtime-loadable, network-specific communication modules (currently TCP/IP, RoCE and InfiniBand). The InfiniBand and RoCE communication modules are based on the use of the RDMA feature present in these network fabrics. Independently of the exact network used, data exchange between rCUDA clients and servers is pipelined in order to attain high performance. Internal pipeline buffers within rCUDA use preallocated pinned memory, given the higher throughput of this type of memory, thus allowing that overall overhead of

using a remote GPU is negligible when InfiniBand is used [11]. When compared to other publicly available remote GPU virtualization frameworks, rCUDA provides the best performance [21].

5.3 System Configurations for Drug Discovery

The computing power required by MD simulators is tremendous. This need for large computing power comes from two different aspects. On the one hand, a single MD simulation requires a huge amount of computations to be completed. In this way, depending on the exact set of molecules to be considered, a single simulation may require several days to be carried out. Besides, in order to perform a complete analysis when searching for new drugs, it is common that MD simulations are executed in batches composed of tens or hundreds of different simulations, each of them working on a different set of ligands.

The computing power required by MD simulators can be achieved in several ways. The most traditional one is based on the use of a large collection of nodes, each of them composed of one or more processor sockets. In particular, hardware configurations where each node leverages two processors are very common because of the good performance/-cost ratio of these systems. In this scenario, a simulation may either be executed in the CPU cores of a single node or may span to several cluster nodes. Nevertheless, considering the cost of inter-node communications across the network fabric, it may be advisable to constrain a MD simulation to a single node if memory resources available in that node are enough for the problem size under execution. This decision may reduce the performance of individual simulations but would increase overall throughput, thus reducing total execution time of the batch of simulations.

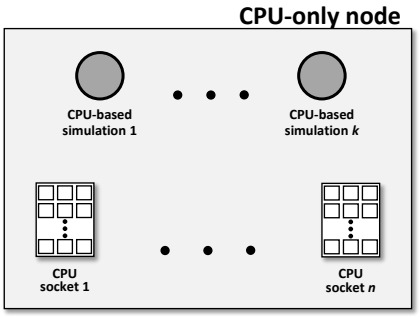
Another possibility to provide the tremendous computing power required by MD simulators is by using GPUs. These devices typically reduce total execution time by one or two orders of magnitude with respect to the use of CPUs. Unfortunately, using GPUs is not exempt from several concerns. For instance, GPUs are noticeable more expensive than CPUs. Also, a single MD simulation does not usually fully utilize the GPUs assigned to it. This non-100% utilization has several consequences: (1) some computing power is wasted at the same time that the bunch of simulations required for VS takes longer

and (2) GPUs waste some amount of energy while not being 100% utilized. In order to address this concern, we may think about concurrently running several MD simulations in the same GPU. However, it must be noticed that clusters usually leverage a job scheduler, such as Slurm, in order to dispatch jobs to nodes and these job schedulers are not able to provide the same GPU to more than one job. Therefore, when GPUs are used in the traditional way, their utilization cannot be easily increased unless the application is improved to generate a higher GPU utilization, which is not possible most of the times given that the very nature of the problem being addressed limits the modifications that can be applied to the application in order to achieve a higher GPU utilization.

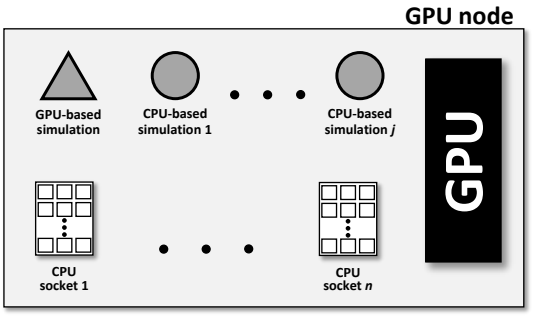
In order to increase GPU utilization and thus make a better usage of available resources, it is possible to virtualize these accelerators and make use of the multi-tenancy approach by leveraging the rCUDA middleware. In this way, a single GPU would be shared among several MD simulations thus making that GPU utilization gets closer to 100%. In this configuration it is possible to concurrently execute several MD simulations in nodes without GPUs while the GPUs located in a single server are shared among these simulations.

In this paper we analyze the three configurations mentioned above (CPU, GPU, and virtualized GPU with rCUDA) in order to find out which one of them best fits the tremendous computational needs of MD simulators. In this regard, although simulation performance is important, given that these simulations are often batched in tens or hundreds of instances, we put the focus of this study on overall throughput instead of individual simulation performance. To conduct this study, we consider the configurations depicted in Figure 5.2 as the basic case studies for each of the three scenarios presented above. Figure 5.2(a) displays the basic case study for the CPU-only configuration. In this case we assume that MD simulations do not spread beyond a single node, as discussed above and therefore the basic case study is composed of a single node comprising a given amount of CPU cores. In this node, one or more concurrent simulations can be executed. The exact amount of concurrent simulations depends on several factors and must be investigated.

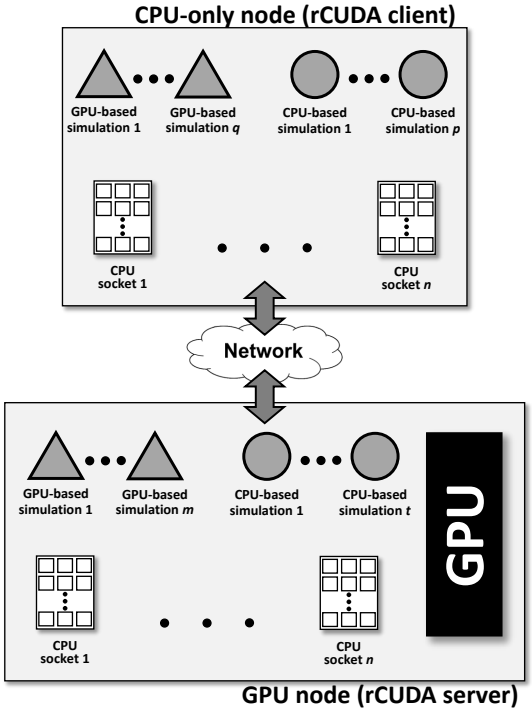
Figure 5.2(b) depicts the basic case study when GPUs are present in the cluster and are used in the traditional way (with job schedulers such as Slurm). As in the previous case, a single node is considered in order to avoid the overhead of inter-node communications



(a) CPU-based configuration. k MD simulations are concurrently executed in the n processor sockets available in the node.



(b) GPU-based configuration. One MD simulation is executed in the GPU whereas j additional simulations are executed in the CPU cores not used by the GPU-based simulation.



(c) rCUDA-based configuration. q+m GPU-based MD simulations share the available GPU whereas p+t simulations are executed in the CPU cores not used by the GPU-based simulations.

FIGURE 5.2: Hardware configurations for each of the baseline case studies considered in this paper.

when the simulation spreads over several cluster nodes. Notice, however, that the GPU-based MD simulation may not require all the CPU cores available in the node. In this case it would be possible to execute one or more CPU-based simulations leveraging the cores not used by the GPU-based instance. This would increase overall throughput.

Finally, Figure 5.2(c) shows the basic case study when rCUDA is used to virtualize

GPUs thus enabling multi-tenancy. It can be seen in Figure 5.2(c) that this basic case study is composed of two nodes: one of the nodes has the GPU and executes the rCUDA server whereas the other node does not include GPUs and therefore executes the MD simulations using the remote GPU in the other node. Given that this scenario allows to concurrently run several MD simulations on the same GPU, the exact amount of simulations must be investigated. The exact number of simulations sharing the GPU will depend on the GPU characteristics. Additionally, this analysis should also include which is the amount of CPU cores provided to each of the simulations that reports the best performance. Moreover, the node running the rCUDA server could also be used to execute additional MD simulations in the GPU, also using rCUDA. Furthermore, it must be noticed that this analysis may conclude a configuration for the MD simulations where several CPU cores (either in the client node or in the server node) are not used. These cores might be used to run CPU-only simulations. In this regard, the obvious goal is to increase as much as possible the overall throughput when tens or hundreds of MD simulations must be executed. To that end, in next sections we will compare the throughput achieved by each of the configurations presented in Figure 5.2, obviously considering that the basic case study for rCUDA includes more resources than the other two basic case studies (it includes two nodes instead of one node).

However, before analyzing the performance and throughput of each of these configurations, we need to understand the bio-informatics problem that is addressed in this paper. This is done in next section.

5.4 Flavonoids as a Working Example

As discussed above, MD is now implemented in drug design work-flows with a focus on improving the accuracy of docking predictions on protein-lingad systems, where the former is the targeted molecule associated to a health disorder. However, there is an increasing effort in the search of molecules able to bind DNA [22]. Indeed, they might be used to either protect living cells from exogenous reactive species (i.e. reactive oxygen species able to initiate side biological degradation phenomena) or to specifically hall cell division machinery (i.e. anticancer molecules reacting with cancer cells). Herein, we decided to use a recent model system designed by [23], who conducted a joint molecular

docking and experimental study to propose a new molecule able to bind to the minor groove of DNA. According to these authors, a fisetin derivative labeled as DEPHBC [2-(3,4-diethoxyphenyl)-3-hydroxy-4H-benzo[h]chromen-4-one], strongly binds to the minor groove of DNA and in addition provides stabilization to the DNA helix architecture. That latter feature foresees a very promising application for developing enhanced drugs, and therefore make the system an ideal working example to test our computational strategy. For the records, the chemical structure and atomic charges of the isolated DEPHBC ligand were fully optimized at the B3LYP/6-31+G(d) level of theory using the Gaussian16 suite of quantum mechanical codes [24], while the DNA model used in our study was directly provided by Halder et al [23]. The resulting model system is subsequently parametrized by using the well-known AMBER99SB force field [25] and a TIP3P water model [26]. Although, other force field may be used, the major goal of this contribution is to show rather than conducting a large assessment/benchmark of MD parameters, lies beyond the scope of our contribution. However, such approach has been successfully used to mimic DNA-related system [27]. In addition, it should be underlined that the use of rCUDA can be successfully used to produce long MD trajectories and therefore help to further force field benchmarking studies.

In short, the goal of the paper is consequently to fill the gap between the earlier reported docking results and the experimental evidences by accounting for dynamical effects with the GROMACS analysis.

5.5 System Performance and Throughput

This section presents the experimental evaluation of this study, based on Intel CPUs and NVIDIA GPUs. First of all, we briefly introduce the hardware and software environment where the experiments are carried out. Afterwards, the performance and throughput of GROMACS is analyzed using CPUs, using real GPUs and using virtual GPUs. In a later subsection we present the overall throughput of the three system configurations discussed in previous sections.

5.5.1 Test bed: Hardware and Software Environment

Experiments have been carried out in a cluster based on two x86-based SYS1028GR-TR Supermicro nodes. Each of the nodes contains two 10-core Intel Xeon E5-2630 v4 processors, and has a Mellanox ConnectX-4 VPI single-port InfiniBand adapter (EDR InfiniBand). The nodes are connected by a Mellanox switch with EDR compatibility (a maximum rate of 100Gb/sec). One of the nodes is equipped with one Tesla P100 GPU owning 16 GB of RAM memory. This node will be used to execute GROMACS using CUDA in the traditional way. This node will also be used to execute the rCUDA server. On the other hand, the other node will be used to execute GROMACS using CPU cores. This node will also be used as the rCUDA client, that is, it will execute GROMACS while remotely using the GPU in the other node.

The CentOS 7.3 operating system and the Mellanox OFED 4.4-2.0.7 were used along with the NVIDIA driver 390.59 and CUDA 8.0. The rCUDA version used is 18.12beta, which is a development version containing all the functionality required to execute applications from any domain using remote GPUs although performance is not fully optimized yet. Regarding GROMACS, version 2016-1 has been used.

5.5.2 Performance Characterization

Although in this paper we put the focus on overall system throughput, in this section we begin the study by characterizing the performance of the GROMACS MD simulator in the three scenarios discussed in previous sections. Figure 5.3 depicts the performance attained by the MD simulator. CPU-only executions of GROMACS are considered as

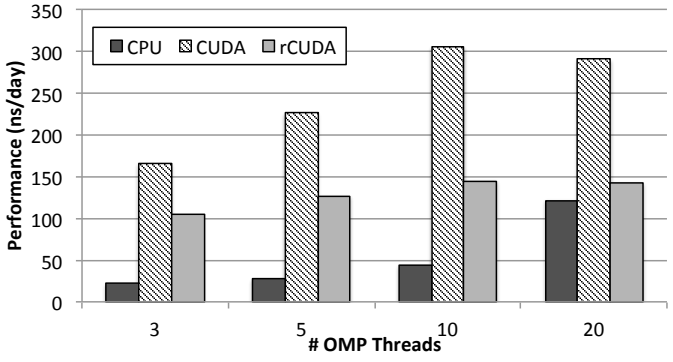


FIGURE 5.3: Performance of the MD simulations when 3, 5, 10 and 20 threads are leveraged. The three basic case studies are considered.

well as executions using a single (real) GPU and executions using a remote virtual GPU across the EDR InfiniBand network. For each of the scenarios, GROMACS has been configured to use 3, 5, 10 or 20 OMP threads (simply threads from now on). Notice that it was not possible to configure GROMACS to use either 1 or 2 threads because of the nature of the simulations being carried out (GROMACS forced to 3 the minimum amount of threads to be used in the simulations).

Figure 5.3 shows that the best performance for the CPU-only simulations is achieved when all the cores in the node are devoted to the simulation. Actually, performance when 20 cores are used is much larger than twice the performance when GROMACS is configured to use 10 cores. Interestingly, performance when 3, 5 and 10 cores are used is proportional to the number of cores. Performance when 20 cores are used does not follow this trend.

When the local GPU is leveraged in the traditional way using CUDA (non-virtualized GPU), it can be seen that performance of GROMACS greatly depends on the exact number of threads used during the simulation. This result is very interesting because it shows that performance not only depends on the use of the accelerator but it also depends on how that accelerator is used. In the particular case of the molecules considered in this study, the best performance is achieved when GROMACS is configured to use 10 threads. In this regard, performance when 20 threads are used is slightly lower than that attained for 10 threads. This result is very important because it shows that a GPU-based facility where all the simulations are configured to use GPUs may easily waste resources: (a) in case the simulations are configured to use all the CPU cores, performance is not maximized, (b) in case simulations are properly configured to maximize performance, some cores at every cluster node will remain idle.

Figure 5.3 also displays the performance when GROMACS leverages a remote GPU (no GPU sharing in this case yet). It can be seen that the performance of a single simulation when rCUDA is used is noticeably lower than the performance when the local GPU is used with CUDA. This lower performance is due to the fact that a development version of rCUDA has been used in this study. This version of rCUDA, which is a major step forward with respect to previous rCUDA versions, contains all the functionality required to execute CUDA applications although its performance has not been optimized yet. This performance was optimized in previous versions of rCUDA [11] although the

functionality of those versions was limited and did not allow to execute some applications. It is expected that next releases of the rCUDA middleware will perform significantly better than the one used in this paper, thus making the overhead of using remote GPUs negligible, as shown in [11].

In addition to analyze the performance of GROMACS in each of the configurations depicted in Figure 5.2, taking a look at energy can provide a complementary perspective to the analysis. Figure 5.4 displays the energy required to perform the simulations in each of the hardware configurations considered. The metric used to show energy is relative to the simulated time: the nanosecond. System energy has been measured by polling once every second the power distribution unit (PDU) present in the cluster. Used unit is APC AP8653 PDU, which provides individual energy measurements for each of the servers connected to it. Therefore, energy measurements shown in Figure 5.4 refer to the entire node executing the MD simulator.

Figure 5.4 shows that energy required in the CPU-only configuration decreases as the amount of threads involved in the execution of GROMACS increases. This result was expected given that, although the energy consumed by the node depends on the amount of active cores, the increment in energy for larger amounts of active cores is absorbed by the energy required by the rest of components of the node. Therefore, the reduction in execution time shown in Figure 5.3 for larger amounts of threads compensates the energy consumed by the additional cores used to run those threads. As a consequence, the faster the simulation is completed, the lower energy is required.

In the case of using the GPU in the traditional way with CUDA (scenario depicted in

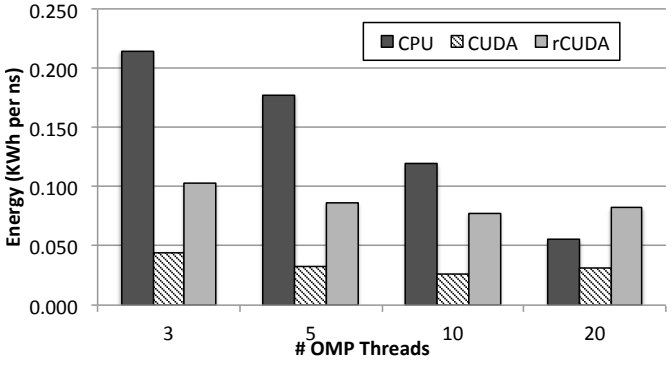


FIGURE 5.4: Energy per simulated ns required by the MD simulations when 3, 5, 10 and 20 threads are leveraged. The three basic case studies are considered.

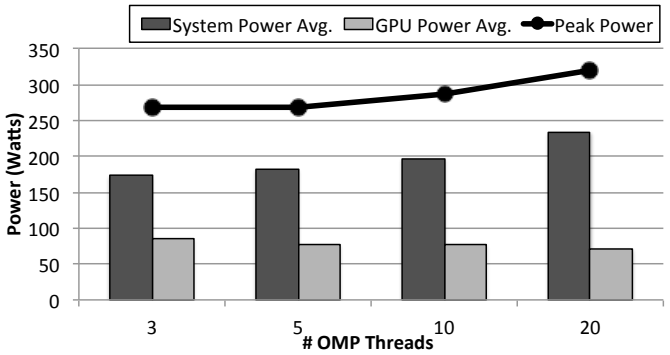


FIGURE 5.5: Average power required by the GPU and by the rest of the system in the CUDA scenario. Peak power required by the entire node also shown. Simulator configurations using either 3, 5, 10 or 20 threads are considered.

Figure 5.2(b)), Figure 5.4 shows that energy per ns is also proportional to execution time. The reason is the same as for the CPU-only scenario: although using a larger amount of threads requires more CPU cores to be active, and thus more power consumption (see Figure 5.5), and additionally also causes a larger GPU utilization, the benefits in performance compensate for that increased power demand at the same time that the additional required energy is partially hidden by the power consumption of the rest of the components of the node. Furthermore, notice in Figure 5.4 that the energy required when 20 threads are leveraged by GROMACS is slightly larger than the one used when 10 threads are used. This higher energy consumption is aligned with the lower performance (larger execution time) shown in Figure 5.3 for the 20-thread CUDA case study.

Finally, regarding the rCUDA configuration, it can be seen in Figure 5.4 that this scenario requires a much larger amount of energy than the CUDA configuration. Two are the reasons for this larger energy demand. On the one hand, in this case we have considered the energy required by the two nodes involved in this scenario: the one executing the CPU part of GROMACS (client side) and the one executing the GPU part of the simulation (rCUDA server). On the other hand, as it was shown in Figure 5.3, performance in the rCUDA configuration is lower than in the CUDA case. This translates into a longer execution time thus causing that energy consumption is higher. Nevertheless, remember that in this work we aim at analyzing the benefits of using a multi-tenant virtual GPU strategy for increasing the throughput of independent GROMACS simulations. Therefore, although the results presented in Figure 5.4 regarding energy consumption for rCUDA are not promising, we should wait until the GPU is shared among several GROMACS instances before making conclusions.

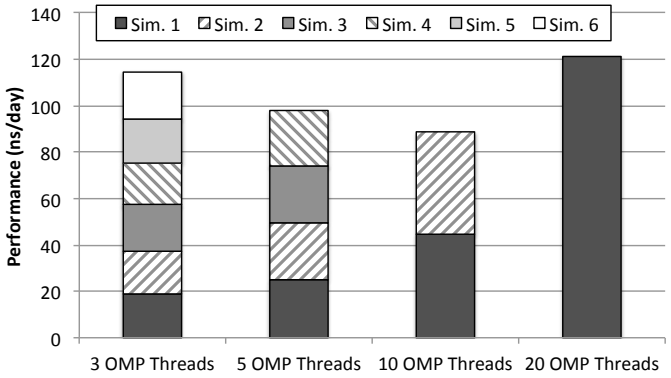


FIGURE 5.6: Throughput of the CPU-only MD simulations when several instances are concurrently executed in the same node. Simulator configurations using either 3, 5, 10 or 20 threads are considered.

5.5.3 Throughput for Each Case Study

In the previous section, the performance of a single instance of GROMACS when executed without any concurrency with other instances has been shown. However, given that we are interested in overall system throughput when tens or hundreds of MD simulations are executed (typical VS workflow used in drug discovery), further experiments must be conducted in order to find out the performance of GROMACS simulations when they are concurrently executed with other MD simulations for each of the scenarios discussed above. In this section we present those throughput results. Notice that in this section we do not mix yet different flavors of the GROMACS simulations. That is, in this section we consider that all instances of GROMACS use either the CPU, the GPU with CUDA or the GPU with rCUDA. In next section we will present throughput results when different GROMACS flavors are combined.

Figure 5.6 shows the overall throughput in the CPU-only scenario. Results in Figure 5.6 have been gathered by executing up to 6 concurrent GROMACS instances in the same node (remember that we have discarded the case study where a simulation spans over several cluster nodes). In order to run up to 6 GROMACS instances in the same node, the simulator has been configured to use 3, 5, 10 or 20 threads. Notice that GROMACS configurations with different amounts of threads have not been mixed. That is, when 3 threads are considered, all the instances of GROMACS make use of such an amount of threads. The same holds for 5 and 10 thread configurations of GROMACS.

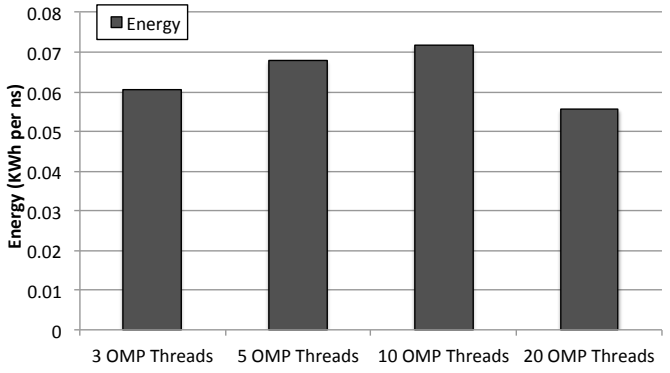


FIGURE 5.7: Energy per simulated ns required by GROMACS when several CPU-only instances are concurrently executed in the same node. Simulator configurations using either 3, 5, 10 or 20 threads are considered.

It can be seen in Figure 5.6 that the best throughput is achieved when a single GROMACS instance is executed using all the available cores in the node (20 threads). This configuration achieves slightly better throughput than the second best option, which is interestingly composed of six 3-thread instances of GROMACS. It is shown in Figure 5.6 that aggregated throughput when six 3-thread instances of GROMACS are concurrently executed in a node is clearly larger than configurations with 5 or 10 threads, despite wasting two of the cores of the node (6 instances of 3-thread simulations require 18 cores instead of 20 cores). Notice that executions in Figure 5.6 have been launched by making use of the `numactl` command, which attaches processes to cores for all the execution of the application, so that data stored in the core caches do have to be migrated during application execution. In this manner, and given that resource managers not always make use of this feature, throughput of CPU-only executions in a real deployment might be slightly lower than that shown in Figure 5.6.

Figure 5.7 presents the energy required by the node concurrently executing the several instances shown in Figure 5.6. It can be seen that the best simulator configuration, attending to energy consumption, is using 20 threads (flooding the entire node with a single simulation). This result is consistent with the energy results previously shown in Figure 5.4 and point out that the additional energy required because of the activation of more cores in the node has a lower impact on the energy/performance ratio than the impact generated by the associated reduction in execution time.

In the case of the CUDA scenario shown in Figure 5.2(b), and given that we are not considering yet mixing different flavors of GROMACS executions, only the case for

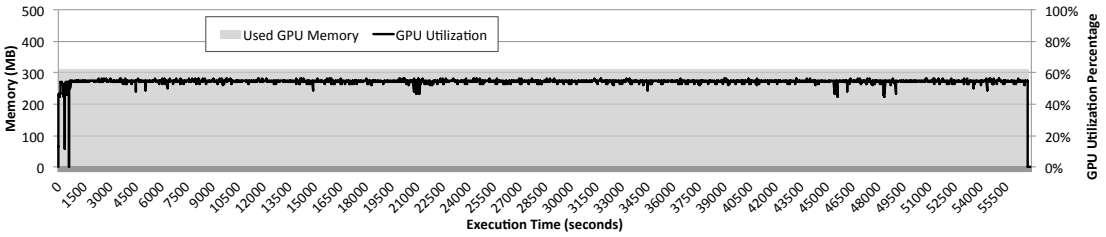


FIGURE 5.8: GPU memory and GPU utilization along the execution time of the GROMACS simulator configured to use 10 threads with the molecules under study. Simulation was configured to last 200 ns of simulated time.

one instance of the GPU-based simulator can be analyzed (mixing different flavors of GROMACS will be analyzed in next section). In this case, as shown in Figure 5.3, maximum performance is attained when GROMACS is configured to use 10 threads. For this particular execution, Figure 5.8 displays the GPU memory usage and GPU utilization along execution time (GROMACS was configured to simulate 200 ns of the movements of the molecules). Data for GPU memory usage and GPU utilization have been gathered by polling the GPU in the node once every second. A homemade program based on the NVML NVIDIA library is used to that end.

It can be seen in Figure 5.8 that the GPU memory footprint of the MD simulation is about 300 MB. This memory footprint is quite small if compared to the memory available in the P100 GPU (16 GB). Furthermore, it can be seen in the figure that GPU utilization remains almost constant despite the large amount of small kernels executed. In this regard, the utilization of the GPU is never larger than 60%. This result is very important because it points out that GPU resources are clearly underutilized. Actually, it is expected that this under utilization is exacerbated in newer and more powerful GPU generations where the gap between the performance of the CPUs and the performance of the GPUs increases. The rationale for this statement is the following: for a simulation as the one depicted in Figure 5.8, the CPU part of the application will take approximately the same time to be executed given that newer processors will not noticeably improve performance per core but they are expected to be more power efficient, according to the trend followed during the last decades. However, the time required for executing the kernels in the GPU will be reduced in newer GPUs presenting a larger amount of cores which, additionally, are more efficient. In this way, given that MD simulations alternate CPU and GPU periods for their entire execution time, it is expected that the GPU

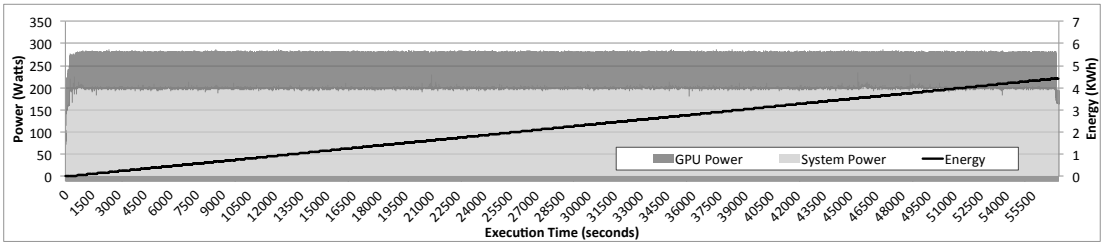


FIGURE 5.9: Instant power and accumulated energy along the execution time of the GROMACS simulator configured to use 10 threads with the molecules under study. Simulation was configured to last 200 ns of simulated time. Instant power is split into GPU power and system power.

periods become shorter due to a reduced execution time whereas execution time of CPU periods remain almost constant. As a consequence, GPU utilization will be reduced.

Figure 5.9 shows the instant power and accumulated energy along the execution time of the simulation shown in Figure 5.8. Instant power is split into GPU power and system power. System power data was gathered by polling once every second the PDU present the cluster, as mentioned before. In order to split power data provided by the PDU into system power and GPU power, the homemade program based on the NVML library was used to collect, every second, the power required by the GPU. Thus, system power presented in Figure 5.9 is the difference between the power measurement provided by the PDU and the power numbers provided by the homemade NVML-based program. It can be seen in Figure 5.9 that power required by the system is around 200 Watts whereas power required by the P100 GPU is around 75 Watts. Furthermore, it can be seen that power required by both the system and the GPU remain almost constant for all the execution time of the simulation. This result was expected from the GPU utilization numbers shown in Figure 5.8, which also remain almost constant for the entire execution of GROMACS. On the other hand, given that consumed energy is proportional to instant power and execution time, it can be seen in Figure 5.9 how total energy requirements for the execution of this simulation increases with execution time. This increment is linear because instant power remains constant during execution time.

Overall throughput in the rCUDA scenario is shown in Figure 5.10. Remember that this case study, contrary to the other two case studies, leverages two nodes instead of only one node. In this way, we can use both nodes to execute instances of GROMACS that will share the GPU located in one of the nodes thanks to rCUDA. Figure 5.10 depicts performance results when the GROMACS instances are configured to make use of 20, 10,

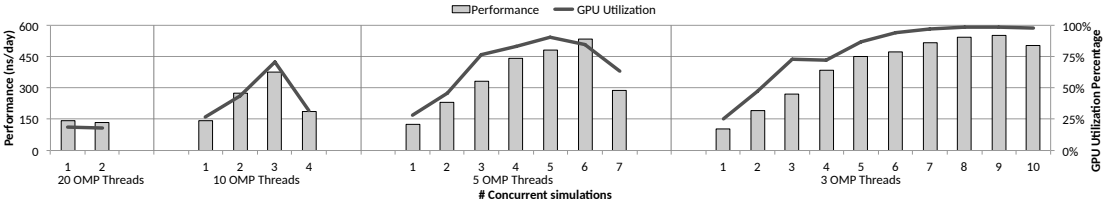


FIGURE 5.10: Throughput and GPU utilization when several instance of GROMACS share the GPU in the rCUDA server by leveraging the rCUDA middleware. Simulator configurations using either 20, 10, 5 or 3 threads are considered.

5 and 3 threads. In the first scenario, one GROMACS instance is executed in the node without GPU whereas the other instance is executed in the node running the rCUDA server (GROMACS instances flood first the client node and then continue filling the server node). It can seen in Figure 5.10 that aggregated performance when 20 threads are used is not increased when a second instance is executed in the node with the GPU. This is due to two different reasons. The first one is that the rCUDA server requires some CPU cores in the GPU server to be run and thus it competes with GROMACS in that node. More precisely, the rCUDA server requires a core per each application process it serves. In this way, given that it is serving 2 instances of GROMACS, it requires 2 cores in the GPU node, in addition to the 20 cores already used by the MD simulator. This oversubscription causes the reduction in performance shown in Figure 5.10. The second, although less important, reason for not increasing performance when a second 20-thread GROMACS instance is executed in the GPU server is that computations in all the 20 threads of a given GROMACS instance must be completed before the simulation can proceed with the next time step. Therefore, given that the rCUDA server process and the GROMACS instance in execution in that node are bothering each other, some threads get delayed thus causing that the entire application executes slower. That is, given the large granularity of the simulations, waiting time becomes the bottleneck. This can also be observed in the low GPU utilization reported in this configuration. In a similar way, although aggregated performance is noticeably increased when 10 threads are used by each GROMACS instance, when all the 40 available cores in the system are used by GROMACS, performance drops. The reason for this is drop in performance is the same as in the previous case. Notice that in this case the rCUDA server make use of 4 cores and therefore oversubscription is larger than in the previous case.

GROMACS performance with configurations using 5 and 3 threads per instance is noticeably better, as shown in Figure 5.10. It can be seen that in the case of 3-thread

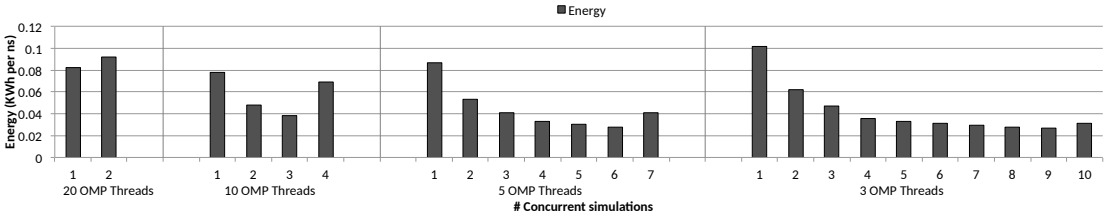


FIGURE 5.11: Energy per simulated ns required by GROMACS when several simulator instances share the GPU in the rCUDA server by leveraging the rCUDA middleware. Simulator configurations using either 20, 10, 5 or 3 threads are considered.

simulations (the smallest granularity considered in this study) GPU utilization is almost 100% beyond 6 instances. Moreover, aggregated performance is almost 600 ns/day when 8 concurrent GROMACS instances share the GPU. Notice that this result is achieved with a version of rCUDA whose performance is not optimized yet. Throughput results with an optimized version of rCUDA are expected to be improved. Actually, with an improved version of rCUDA, it is expected that 100% GPU utilization is achieved with a smaller amount of instances (currently 8 instances) thus allowing to use a larger amount of cores for additional CPU-only simulations. This would further increase overall throughput.

Figure 5.11 shows the energy point of view of the results shown in Figure 5.10. As in previous figures, the energy required for simulating a nanosecond is displayed. The energy required by both nodes (client and server sides) is considered in the figure. It can be seen that the more instances of the simulator are concurrently run, the better energy results are obtained. This rule is broken when the server system gets congested either at the GPU or at the CPU. In this regard, Figure 5.11 shows an increment in the energy trend for the second instance when 20 threads are used, for the fourth instance when GROMACS uses 10 threads, for the seventh instance in the case of using 5 threads per simulation and, finally, in the tenth instance when GROMACS is run using 3 threads per instance. Furthermore, as in the previous figures, energy is proportional to execution time (or performance). Additionally, if the energy per ns required when running 8 3-thread GROMACS instances with rCUDA (Figure 5.11) is compared with the energy per ns required when running one 10-thread simulator instance with CUDA (Figure 5.9), it can be seen that energy is similar in both cases.

Finally, remember that overall throughput in the CUDA case was 300 ns/day per node (results in Figure 5.3 for 10-thread simulations). Notice, however, that the hardware

used to achieve the performance in the CUDA and in the rCUDA cases is not the same. Although in both cases only one GPU is leveraged, in the rCUDA case a second node has been used. Therefore, more CPU cores were available in the rCUDA scenario. In order to perform a fair comparison, in next section we analyze the throughput attained by each of the system configurations presented in Figure 5.2 when using a similar amount of hardware resources.

5.5.4 Overall System Throughput

In the previous sections we have first analyzed the performance of GROMACS in each of the scenarios depicted in Figure 5.2 when a single instance is run without sharing resources with other instances. Later, we have studied how performance was improved when several instances were concurrently run in each of the scenarios (except the GPU-based one, which does not allow several instances of GROMACS to share the GPU). However, it is also possible to mix the different flavors of GROMACS (CPU-based and GPU-based executions) in order to increase overall throughput of the system. In this section we perform such an analysis.

CPU and GPU-based flavors of GROMACS can be mixed in the CUDA (Figure 5.2(b)) and rCUDA (Figure 5.2(c)) scenarios (the CPU-based scenario only allows to run CPU instances of GROMACS). In these two scenarios, the CPU cores not devoted to GPU-based simulations can be used to execute additional instances of GROMACS using only the CPU cores. By making this kind of mixtures, it is expected to increase overall throughput. Using the performance data gathered in previous sections, we can make projections about overall system throughput in terms of aggregated ns/day. In order to make such projections, we will use the performance data shown in Table 5.1. This table shows that a single CPU-based simulation using 20 threads achieves a performance of 121.33 ns/day (this is the value already shown in Figure 5.3). Similarly, when only 3 threads are used, performance is lowered to 22.59 ns/day. If the GPU is used by a GROMACS simulation configured with 10 threads, attained performance is 305.46 ns/day. On the other hand, in the rCUDA scenario, when 8 concurrent 3-thread simulations share the GPU (using two different cluster nodes to run the CPU processes) aggregated performance is 542.53 ns/day. Furthermore, when rCUDA is used in a single node (all the GROMACS simulations being run in the node owning the GPU and running the

TABLE 5.1: Performance achieved by several GROMACS configurations

Configuration	Label	Performance (ns/day)
CPU 20 threads	A	121.33
CPU 3 threads	B	22.59
CUDA 10 threads	C	305.46
rCUDA 2 nodes: eight 3-thread instances	D	542.53
rCUDA 1 node: five 3-thread instances	E	452.64

rCUDA server), 5 concurrent 3-thread simulations report an aggregated performance of 452.64 ns/day. Notice that in this latter configuration, all the 20 cores in the node are used because the GROMACS instances are using 15 cores whereas rCUDA makes use of 5 additional cores to serve those 5 GROMACS instances.

With the data presented in Table 5.1 we can make two different projections in order to mix several GROMACS flavors. First, we can assume a cluster composed of n nodes where half of the nodes own a GPU and the other half do not own a GPU. In this cluster configuration, the non-GPU nodes would be used to execute CPU-based instances of GROMACS. For these CPU-based simulations, the best configuration is using 20 threads per instance as shown in Figure 5.3. In addition to the GROMACS executions run in the non-GPU nodes, each GPU node would execute one 10-thread simulation. This would leave 10 unused cores in the node, which could be used to run three 3-thread instances of GROMACS. This cluster configuration would therefore report an overall throughput equal to 494.56 ns/day per each couple of nodes (one node with GPU and one node without GPU). With this very same hardware resources (one non-GPU node and one GPU node), in the rCUDA case, it would be possible to execute eight 3-thread GPU instances of GROMACS with rCUDA and two 3-thread CPU instances in the spare cores. This would provide a throughput equal to 587.71 ns/day. This translates into a 1.19x speed-up when virtual GPUs are used. Notice that the same hardware resources are leveraged in both cases.

The second projection that can be made with the numbers in Table 5.1 is assuming a cluster where every node owns one GPU. In this scenario, we could use a single node to execute the rCUDA-based GROMACS simulations. In this case, that node would

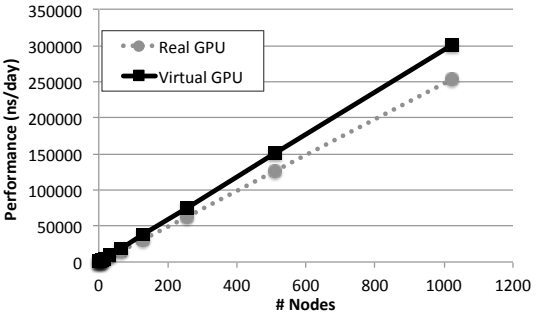


FIGURE 5.12: Aggregated throughput projection for a hybrid cluster composed of n nodes where half of the nodes own a GPU whereas the other half of the nodes do not leverage any accelerator.

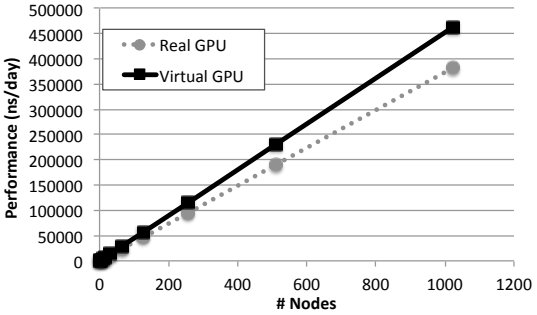


FIGURE 5.13: Aggregated throughput projection for a homogeneous cluster composed of n nodes where all the nodes own one GPU.

be able to run five 3-thread GROMACS instances, thus using all the 20 cores in the node and reporting a throughput equal to 452.64 ns/day. In the case of the CUDA executions, three 3-thread CPU-based instances of GROMACS could be run in addition to the 10-thread GPU-based one. That would report a total throughput equal to 373.23 ns/day. As can be seen, in this cluster configuration, speed-up of using multi-tenancy with virtual GPUs would be 1.21x. Again, same hardware resources would be used in both cases.

The assumptions above can be formalized using the following equations:

- For the hybrid cluster composed of nodes with one GPU and nodes without GPU:

  Throughput real GPU = (A + 3B + C) * n/2

  Throughput virtual GPU = (2B + D) * n/2

- For the homogeneous cluster where all the nodes own one GPU:

  Throughput real GPU = (3B + C) * n

  Throughput virtual GPU = E * n

The equations above allow us to make a throughput estimation depending on the number of nodes in the cluster, which is referred to as n in the equations. Furthermore, labels "A", "B", "C", "D" and "E" refer to the values shown in Table 5.1. Figures 5.12 and 5.13 present such estimations for the hybrid and homogeneous clusters, respectively. It can be seen in both figures that applying the multi-tenant virtual GPU strategy effectively increases the throughput of independent GROMACS simulations while using the same hardware as in the real GPU scenario.

5.5.5 Analysis of obtained MD results in terms of biological validation

Regarding the biological significance and correctness of obtained results from previously mentioned MD results, As one can see in Figure 5.14 the computed Root Mean Square Deviation (RMSD) for the DNA structure along the MD trajectory ranges from around 0.3 to 2.2 nm. Although these might seem high RMSD values for such structural model, this is the logical consequence of performing long MD simulations without imposing any structural restriction to the studied DNA fragment. Of course, we may add end-to-end distance constraints to DNA in order to reduce these RMSD values. Besides, from Figures 5.15 and 5.16 we can observe that ligand DEPHBC remains stable regarding non-covalent interactions with DNA system, as Halder et al. reported experimentally [23], which confirms the validity of our proposal. However, our main goal was to show how rCUDA helps to increase the sampling of a bio-model free of any geometrical restriction to add a full dynamic protocol. It is also worth stressing that in spite of such measurable variation in the RMSD, the characteristic double helix architecture is maintained, so that reliable macroscopic conclusions can be extracted from the GROMACS output.

5.6 Conclusions and Future Work

MD are computational tools that simulate the dynamical behavior of atoms and molecules. This simulation process is of paramount importance for several fields such as drug discovery, material simulation, etc. However, the number of simulations and the computational horsepower required by them, limits the success of MD techniques in real scenarios and the only solution is to scale to heterogeneous supercomputers comprised of CPUs

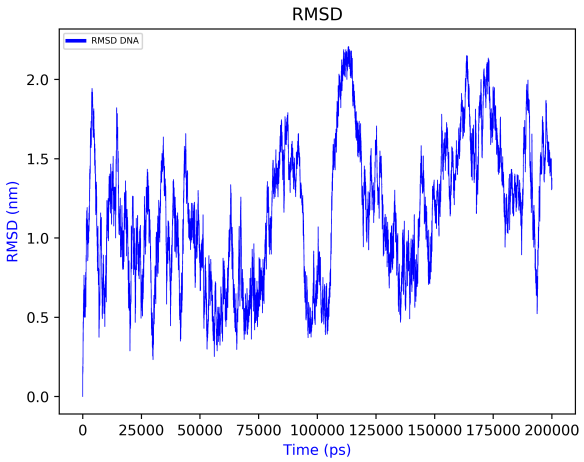


FIGURE 5.14: RMSD over time for the DNA structure.

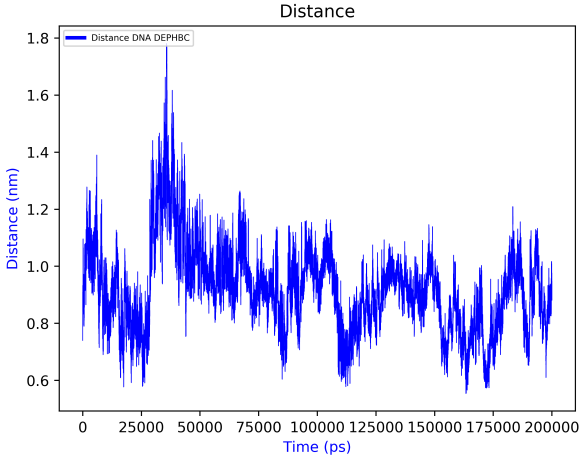


FIGURE 5.15: Average DNA(center of mass) to DEPHBC distance over time.

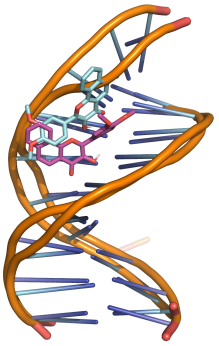


FIGURE 5.16: Superposition of first and last frame of the DNA-DEPHBC MD simulation.

and GPUs. This paper shows that making use of a multi-tenant virtual GPU strategy is an effective way to enhance the overall throughput of GROMACS MD simulations. To that end, we have used the virtualized GPUs provided by the rCUDA middleware. rCUDA enables remote concurrent usage of CUDA-compatible GPUs and thus physical GPUs can be concurrently shared among several applications. This fact increases GPU occupancy by running several GROMACS instances at the same time. Furthermore, space cores in the system are devoted to run CPU-based GROMACS instances. Our results show that the use of rCUDA allows a speed-up over 1.21x while using the very same hardware resources. In addition, we apply our proposal to a system of biological relevance (DNA-DEPHBC) and validate it against previously obtained experimental results.

Future work includes widening this analysis with other data sets in order to verify the stability of the results. Other execution configurations should also be explored. For instance, instead of using single node simulations, spanning the execution of GROMACS to several cluster nodes should also be taken into account. Additionally, other GPU generations (such as the NVIDIA V100 GPU featuring more cores and more memory) should also be addressed in order to assess the feasibility of our proposal in recent cluster deployments. Finally, instead of making throughput projections, the actual performance of mixed GROMACS configurations should be investigated. Furthermore, according to our experience with the GROMACS simulations conducted in this study, energy consumption is probably lower when the multi-tenant virtual GPU strategy is leveraged. Thus, the exact energy requirements of such mixed configurations should also be analyzed. In this regard, if out intuition is confirmed, not only throughput would be increased but also total energy required to complete the MD simulations would be reduced.

Acknowledgments

This work jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grants (20524/PDC/18, 20813/PI/18 and 20988/PI/18) and by the Spanish MEC and European Commission FEDER under grants TIN2015-66972-C5-3-R, TIN2016-78799-P and CTQ2017-87974-R (AEI/FEDER, UE).

We also thank NVIDIA for hardware donation under GPU Educational Center 2014-2016 and Research Center 2015-2016. The authors thankfully acknowledge the computer resources at CTE-POWER and the technical support provided by Barcelona Supercomputing Center - Centro Nacional de Supercomputaciòn (RES-BCV-2018-3-0008). Furthermore, researchers from Universitat Politècnica de València are supported by the Generalitat Valenciana under Grant PROMETEO/2017/077. Authors are also grateful for the generous support provided by Mellanox Technologies Inc. Prof. Pradipta Purkayastha, from Department of Chemical Sciences, Indian Institute of Science Education and Research (IISER) Kolkata, is acknowledged for kindly providing the initial ligand and DNA structures.

References

[1] David E Shaw et al. Atomic-level characterization of the structural dynamics of proteins. *Science*, 330(6002):341–346, 2010.

[2] Mark James Abraham et al. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[3] Berk Hess et al. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3), 2008.

[4] David A Case et al. The amber biomolecular simulation programs. *Journal of computational chemistry*, 26(16):1668–1688, 2005.

[5] James C Phillips et al. Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[6] Sander Pronk, , et al. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[7] Michael Patra et al. Long-range interactions and parallel scalability in molecular simulations. *Computer physics communications*, 176(1):14–22, 2007.

[8] AH Poghosyan et al. Parallel peculiarities and performance of gromacs package on hpc platforms. *Int. J. of Scientific and Eng. Research*, 4(12):1755–1761, 2013.

[9] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003.

[10] S. Iserte et al. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. In *CCGrid*, May 2016.

[11] Carlos Reaño et al. Local and remote gpus perform similar with EDR 100G InfiniBand. Middleware '15, pages 4:1–4:7. ACM, 2015.

[12] Javier Prades et al. Increasing molecular dynamics simulations throughput by virtualizing remote gpus with rcuda. ICPP '18, 2018.

[13] Irene Sánchez-Linares, Horacio Pérez-Sánchez, José M Cecilia, and José M García. High-throughput parallel blind virtual screening using BINDSURF. *BMC Bioinformatics*, 13(Suppl 14):S13, 2012.

[14] Baldomero Imbernón et al. METADOCK: A Parallel Metaheuristic schema for Virtual Screening methods. *The International Journal of High Performance Computing Applications*, March 2017.

[15] Banegas-Luna et al. Advances in distributed computing with modern drug discovery. *Expert opinion on drug discovery*, (just-accepted), 2018.

[16] Alejandro A Franco. Multiscale modelling and numerical simulation of rechargeable lithium ion batteries: concepts, methods and challenges. *RSC Advances*, 3(32): 13027–13058, 2013.

[17] Douglas B Kitchen, Hélène Decornez, John R Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature Reviews Drug Discovery*, 3(11), 2004.

[18] Nathalie Lagarde and othersu. Benchmarking data sets for the evaluation of virtual ligand screening methods: review and perspectives. *J. of Chemical Inf. and Modeling*, 55(7), 2015.

[19] Ajay N Jain. Scoring functions for protein-ligand docking. *Current Protein and Peptide Science*, 7(5):407–420, 2006.

[20] Peter Csermely et al. Structure and dynamics of molecular networks: a novel paradigm of drug discovery: a comprehensive review. *Pharmacology & Therapeutics*, 138(3), 2013.

[21] Carlos Reaño and Federico Silla. A performance comparison of cuda remote gpu virtualization frameworks. In *2015 IEEE International Conference on Cluster Computing*, Sept 2015.

[22] M. Noroozi et al. Effects of flavonoids and vitamin c on oxidative dna damage to human lymphocytes. *American Journal of Clinical Nutrition*, 67, 1998.

[23] Dipanjan Halder and Pradipta Purkayastha. A flavonol that acts as a potential dna minor groove binder as also an efficient g-quadruplex loop binder. *Journal of Molecular Liquids*, 265, 2018. ISSN 0167-7322.

[24] M. J. Frisch et al. Gaussian 16 Revision A.03, 2016.

[25] V. Hornak et al. Comparison of simple potential functions for simulating liquid water. *Proteins*, 65, 2006.

[26] William L. Jorgensen et al. Comparison of simple potential functions for simulating liquid water. *J. of Chem. Physics*, 79(2), 1983.

[27] Hassan Pezeshgi Modarres et al. Understanding and engineering thermostability in dna ligase from thermococcus sp. 1519. *Biochemistry*, 54(19):3076–3085, 2015.

Chapter 6

Turning GPUs into Floating Devices over The Cluster: The Beauty of GPU Migration

Javier Prades, Federico Silla. **Proceedings of the 46th International Conference on Parallel Processing Workshops** - August 2017 – Pages 129 - 136 [1]
https://doi.org/10.1109/ICPPW.2017.30

Abstract

Virtualization techniques have shown to report benefits to data centers and other computing facilities. In this regard, not only virtual machines allow reducing the size of the computing infrastructure while increasing overall resource utilization but also virtualizing individual components of computers may provide significant benefits. This is the case, for example, for the remote GPU virtualization technique, implemented in several frameworks during the recent years.

The large degree of flexibility provided by the remote GPU virtualization technique can, however, be further increased by applying the migration mechanism to it, so that the GPU part of applications can be live migrated to another GPU elsewhere in the cluster during execution time in a transparent way.

In this paper we present a discussion about how the migration mechanism has been applied to different GPU virtualization frameworks. We also provide a big picture about the possibilities that migrating the GPU part of applications can provide to data centers and other computing facilities. We finally present the first results of an ongoing work consisting on applying the migration mechanism to the rCUDA remote GPU virtualization framework.

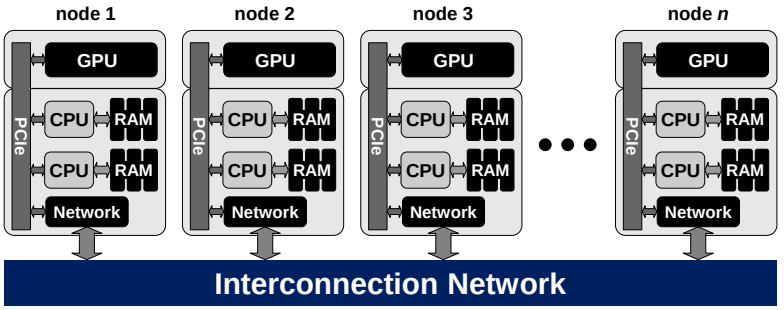
6.1 Introduction

Virtualization has become a very important mechanism to increase the efficiency of data centers and other computing facilities. Virtualization allows acquisition costs to be better tailored to the real computing needs. Moreover, virtualization allows data centers to noticeably reduce their energy footprint by consolidating servers, therefore switching off the hardware resources that are not being used at a given point in time. The concept of virtualization can be applied at different levels, as exposed below.

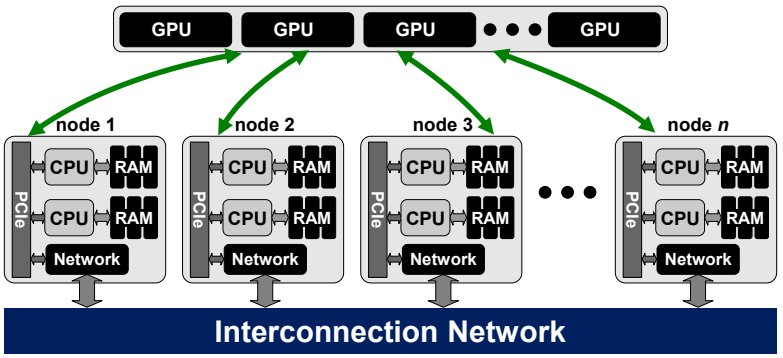
Firstly, the virtualization mechanism can be applied at the computer level, leading to the well known and widely used virtual machine frameworks. Examples of this technology are solutions such as VMware [1], Xen [2], KVM [3], or VirtualBox [4], which have became so popular because several instances of these frameworks (several virtual machines) can be concurrently executed in a real computer, sharing its resources and hence increasing overall utilization. As a consequence of the widespread use of virtual machines, processor manufacturers like Intel or AMD incorporate an increasing virtualization support into their products [5].

Notice that, in the context of the previous frameworks, virtualization can also be applied at the device level in order to provide support to virtual machines. For instance, network adapters for technologies and manufacturers as different as Mellanox' InfiniBand or Intel's Ethernet include virtualization features [6][7] which basically allow the network adapter to be replicated, at the logic level, so that different replicas of the network card are assigned to different virtual machines. In a similar way, graphics processing units (GPUs) have recently also included some virtualization support. This is the case, for instance, of the GRID K1 GPU by NVIDIA [8], which can be shared among up to 64 virtual machines, although it is only intended for desktop virtualization.

In addition to provide support to virtual machines, virtualization of individual devices of a computer may be also intended to provide an increased degree of flexibility at the cluster level. For example, networked disks enable sharing a file system across a cluster. In a similar way, the recent remote GPU virtualization technique, implemented in frameworks like rCUDA [9], GVirtuS [10], or DS-CUDA [11], among others, allows a set of GPUs to be concurrently shared among several cluster nodes. Figure 6.1 depicts this idea. In Figure 6.1(a) a cluster composed of n nodes is shown, each node containing



(a) Example of a GPU-accelerated cluster.

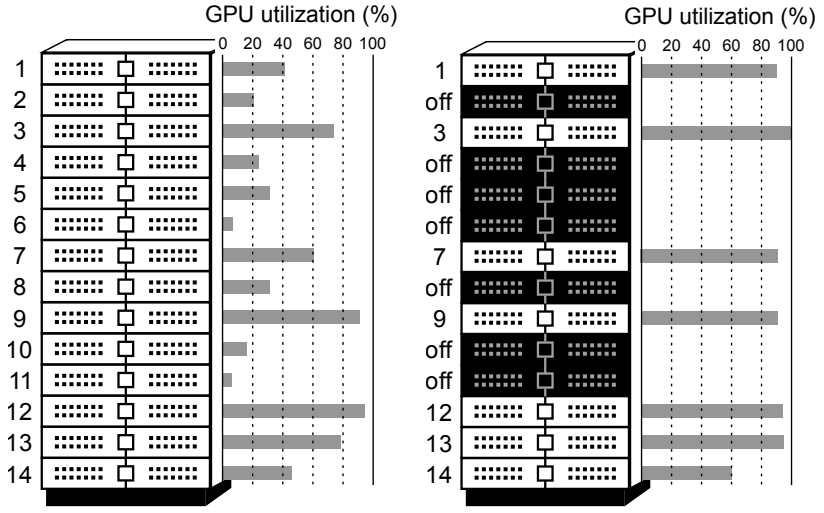


(b) Logical configuration of a cluster when the remote GPU virtualization technique is used.

FIGURE 6.1: Comparison, from a logical point of view, of two cluster configurations: (a) remote GPU virtualization is not leveraged; (b) remote GPU virtualization is used.

two Xeon processors and one NVIDIA Tesla GPU. Figure 6.1(b) shows the new cluster envision after applying the remote GPU virtualization mechanism. In the new cluster configuration, GPUs are logically detached from nodes and a pool of GPUs is created. GPUs in this pool can be accessed from any node in the cluster. Furthermore, a given GPU may concurrently serve more than one application. This sharing of GPUs not only increases overall GPU utilization but also reduces the total energy required to operate a computing facility [12], thus loosening the big energy and power consumption concerns of future data centers. Additionally, remote GPU virtualization also allows easier system upgrades, given that a cluster without GPUs can execute GPU-accelerated applications just by attaching one or more GPU servers to the cluster.

Remote GPU virtualization provides a lot of flexibility to the way that GPUs are actually used in a cluster because this mechanism allows to separately schedule, for a given application, the use of CPUs and the use of GPUs. That is, the application can be assigned CPU cores in some nodes of the cluster while using GPUs belonging to a different set of nodes. Moreover, GPUs can be concurrently shared among different applications.



(a) GPU utilization before consolidating GPU servers.

(b) Cluster state after consolidating GPU servers.

FIGURE 6.2: Usage of GPU migration in a cluster in order to consolidate GPU jobs and reduce energy. In (a) all the nodes in the cluster are switched on whereas in (b) seven nodes have been switched off thanks to GPU server consolidation after having migrated GPU jobs.

This large degree of flexibility can, however, be further increased by allowing the GPUs assigned to a given application to move around in the cluster while the application is in execution. This movement means that the application is initially provided one or more GPUs in one or more nodes of the cluster but, during application execution, the GPU part of the application is transparently migrated to other GPU (or GPUs) elsewhere in the cluster. This migration of the GPU part of an application can provide many different benefits to data centers and other computing facilities.

Probably, the most immediate benefit of migrating the GPU part of an application is to support server consolidation. In this regard, notice that resource utilization in data centers evolves over time, depending on the exact workload applied at every moment. Therefore, at some point in time, the utilization of the GPUs in the cluster may be similar to that depicted in Figure 6.2(a). This figure shows a small cluster composed of 14 nodes, each of them including one GPU. Next to each node, the utilization of its GPU is displayed. It can be seen that some nodes present a high GPU utilization. For instance, nodes 9 and 12 are using their GPUs at 90% approximately. On the contrary, some other nodes present a very low GPU utilization, such as nodes 6 and 11, whose GPUs are almost not used. In this scenario it would be useful to gather the GPU jobs being executed in those nodes into other nodes. That is, it would be useful to consolidate

the GPU jobs into a smaller number of servers, so that those nodes that become free can be switched off, thus reducing the energy consumption of the data center. Figure 6.2(b) depicts this consolidation of GPU jobs, where jobs generating a lower GPU utilization, such as the ones in nodes 2, 4, 5, 6, 8, 10, and 11 have been migrated to other nodes. After job migration, the nodes sourcing the movement of jobs have been switched off, thus consuming a negligible amount of energy.

Another benefit of GPU migration is checkpointing. It is well known that errors happen with a non-negligible frequency in large computing facilities. Additionally, GPUs are not exempt from suffering errors. For instance, in [13] it was shown that an important fraction of tested GPUs exhibited a detectable, pattern-sensitive rate of soft errors. In this context, the widely used checkpointing technique should be applied to long-running jobs, so that the computations (and invested energy) carried out until the moment of the failure are not lost. Checkpointing GPU applications requires, however, a special management of the GPU state. Therefore, if a mechanism for live migrating the GPU part of an application is developed, then that very same mechanism can be used to store the GPU state in disk instead of moving it to another node of the cluster.

In addition to having motivated the need for GPU migration, in this paper we also present a review about how the migration mechanism has been implemented within different GPU virtualization frameworks. We also present the first results of an ongoing work consisting on applying the migration mechanism to the rCUDA remote GPU virtualization framework. To that end, the rest of the paper is organized as follows. Section 6.2 presents a revision of different GPU virtualization solutions. Next, Section 6.3 provides a review on GPU migration implementations, including the one we are currently working on in the context of the rCUDA middleware. In Section 6.4 we present the first results of applying the migration mechanism to the rCUDA framework. Finally, Section 6.5 concludes this work.

6.2 About Remote GPU Virtualization

Several software-based GPU sharing mechanisms have been developed in the context of CUDA [14] during the recent years, such as, for example, DS-CUDA [11], rCUDA [9], vCUDA [15], GridCuda [16], GVirtuS [10] or GViM [17]. Basically, these middleware

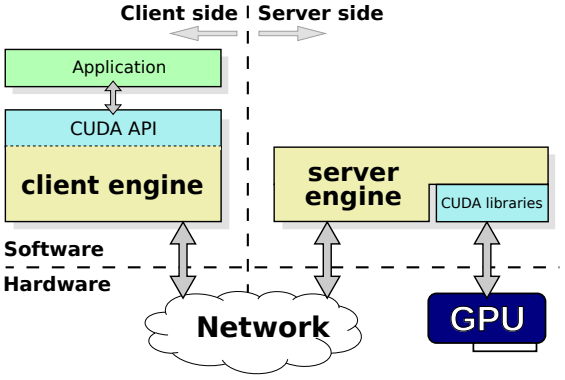


FIGURE 6.3: General organization of remote GPU virtualization frameworks.

proposals share a GPU by virtualizing it. Usually, these GPU sharing solutions place the virtualization boundary at the API level. In general, CUDA-based virtualization frameworks aim to offer the same API as the NVIDIA CUDA Runtime API [18] does.

Figure 6.3 depicts the architecture usually deployed by these GPU virtualization solutions, which follow a distributed client-server approach. The client part of the middleware is installed in the cluster node executing the application requesting GPU services, whereas the server side runs in the node owning the actual GPU. Communication between client and server may be based on shared-memory mechanisms or on the use of a network fabric, depending on the exact features of the GPU virtualization middleware and the underlying system configuration. The architecture depicted in Figure 6.3 is used in the following way: the client middleware receives a CUDA request from the accelerated application and appropriately processes and forwards it to the server middleware. In the server side, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client middleware, which forwards them to the accelerated application. Notice that GPU virtualization solutions provide GPU services in a transparent way and, therefore, applications are not aware that their requests are actually serviced by a virtual GPU instead of by a local one.

Different GPU virtualization solutions feature different characteristics. For instance, the vCUDA technology, intended for Xen virtual machines, only supports the old CUDA version 3.2 and implements an unspecified subset of the CUDA Runtime API. Moreover, its communication protocol presents a considerable overhead, because of the cost of the

encoding and decoding stages, which causes a noticeable drop in overall performance. GViM, also targeting Xen virtual machines, is based on the obsolete CUDA version 1.1 and, in principle, does not implement the entire CUDA Runtime API. GVirtuS is based on the old CUDA version 6.5 and implements only a small portion of its API. Despite being designed for virtual machines, it also provides TCP/IP communications for remote GPU virtualization, thus allowing applications in a non-virtualized environment to access GPUs located in other nodes. In a similar way, GridCuda also offers access to remote GPUs in a cluster, but supports the old CUDA version 2.3. Moreover, there is currently no publicly available version of GridCuda that can be used for testing. Finally, DS-CUDA integrates version 4.1 of CUDA and includes specific communication support for InfiniBand. However, DS-CUDA presents several strong limitations, such as not allowing data transfers with pinned memory.

Regarding rCUDA (remote CUDA), this middlewware supports version 8.0 of CUDA, the latest available one at the time of writing this paper, being binary compatible with it, which means that CUDA programs do not need to be modified for using rCUDA. Furthermore, it implements the entire CUDA API (except for graphics functions) and also provides support for the libraries included within CUDA, such as cuFFT, cuBLAS, cuSPARSE, cuDNN, cuSOLVER, etc. rCUDA provides specific support for different interconnects. This is achieved by making use of a set of runtime-loadable, network-specific communication modules, which have been specifically implemented and tuned in order to obtain as much performance as possible from the underlying interconnect. Currently, three modules are available: one intended for TCP/IP compatible networks, another one specifically designed for InfiniBand, which makes use of the RDMA feature of this network, and a third one intended for RoCE networks, which also leverages RDMA features. Compared to other publicly available remote GPU virtualization frameworks, the rCUDA middleware provides the best performance [19].

6.3 Implementing GPU Migration

Migrating GPUs has been addressed in the past in several works, mostly intended for checkpointing purposes, although migration could be seen as a secondary goal as well. For instance, in [20] a prototype implementation of a checkpointing framework for CUDA

applications, named CheCUDA, is presented. As a prototype, it only supports a small fraction of the functions within the obsolete Driver API of CUDA 2.2. CheCUDA is based on the use of the BLCR framework [21], which allows checkpointing the CPU-part of an application. In order to know which are the memory areas of the GPU to be included in the checkpoint, CheCUDA provides a limited set of wrappers to some of the basic cuMemAlloc functions in the Driver API. These wrappers record all the necessary information about the reserved memory areas (starting address, length, etc). To that end, CheCUDA needs the application source code to be modified in order to include the `CheCUDA.h` header file. Given that CPU and GPU must be synchronized for checkpointing, once the checkpoint signal has been triggered, the CheCUDA framework waits for the next `cuCtxSynchronize()` function before performing the checkpointing.

Another proposal is described in [22], where a non-mature hybrid checkpointing technology intended to support checking a running GPU kernel at any time during its execution is presented. The proposal is transparent to the programmer given that no source code modification is required to perform the checkpoint, although it is based on the debug interface of CUDA, therefore forcing kernels to run in synchronization mode, thus causing a large execution overhead. Additionally, the debug interface of CUDA requires detailed debug information which can only be found in debug versions of applications, which is not usually the case for commodity software.

A similar proposal is presented in [23], although in this case the proposal is intended for OpenCL [24] instead of CUDA. OpenCL-based GPU virtualization frameworks aim to provide the same API as OpenCL [24] does. One of these solutions is VOCL [25], although other frameworks are available, such as SnuCL [26], VCL [27], or dOpenCL [28]. The mechanism presented in [23] is intended to support the VOCL framework. This mechanism is not designed for checkpointing purposes but it was devised for migrating GPU jobs. Anyway, as the authors in [23] mention, it could also be used for checkpointing applications. Similarly to the previous proposals, this framework also requires that the kernels in the GPU are completed before migration begins. Additionally, as in previous proposals, it is also based on intercepting the memory allocation calls in order to store the required information to perform the GPU migration. Moreover, a couple of functions are provided in order to trigger migration from the executing application. When migration is triggered, the framework looks for a suitable destination GPU.

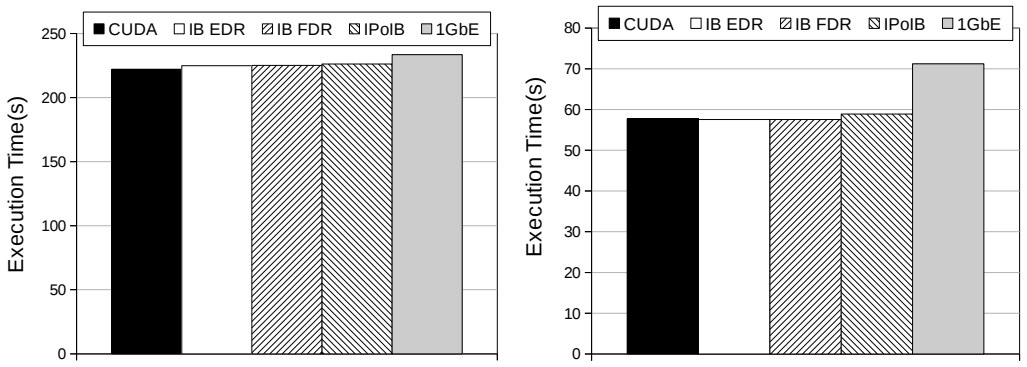
One more proposal for checkpointing is described in [29], although in this case the proposal is intended for Intel GPUs. Nevertheless, similar concerns to the ones described for the previous proposals also apply to this one.

Finally, the GPU migration implementation carried out within the rCUDA framework is also based on the set of wrappers to CUDA functions included in the rCUDA library, which provides the very same API as the CUDA one. In this way, whenever a memory allocation CUDA function is called, the rCUDA framework intercepts it and stores the required information for a possible future GPU migration. Additionally, as in the previous proposals, the implementation done within the rCUDA middleware also requires the kernels being in execution in the GPU to be completed before the migration begins. However, contrary to the previous proposals, the migration is not triggered by the application (source code is not modified) but by an external signal. This signal, in the form of a TCP/IP connection to a properly configured port in the rCUDA server, is sent by a job scheduler, which will also send across the connection the required information for the migration (which is the destination GPU, which specific client in that rCUDA server will be migrated, etc).

6.4 First results of GPU Migration within rCUDA

In this section we present some preliminary performance results of our implementation of GPU-job migration within the rCUDA middleware. These results belong to an ongoing work currently under development. In order to gather those results, we will use two different applications, that will be live migrated while they are in execution. The first application is a synthetic in-house program whereas the second application is GPU-BLAST [30].

The synthetic application performs the multiplication of a vector by a scalar. To that end, it initially allocates GPU memory for 1000 randomly-sized arrays and fills them by copying data from host memory to GPU memory. Then the application launches the necessary kernels to apply the multiplication to the 1000 vectors and finally results are copied back from GPU to host memory. Afterwards, GPU memory is finally released. The aggregated volume of memory used at the GPU for the 1000 arrays is 700 MB. Therefore, when migration is triggered, the rCUDA framework should perform 1000



(a) Synthetic application. (b) GPU-BLAST application.

FIGURE 6.4: Execution time using CUDA and rCUDA without migration.

allocations of GPU memory at the destination GPU, should perform 1000 memory copies between source and destination GPUs, and then should carry out 1000 memory releases at the source GPU, which is freed and thus no longer related to the execution of the application.

On the other hand, the GPU-BLAST application has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST (http://www.ncbi.nlm.nih.gov) implementation using GPUs. The behavior of the GPU-BLAST application can be simplified according to the following pseudo-code snippet:

```
// execution block 1
transfer 1300MB data to the GPU
execute kernel in the GPU
transfer 1300MB data from the GPU

// execution block 2
transfer 1300MB data to the GPU
execute kernel in the GPU
transfer 1300MB data from the GPU

// execution block 3
transfer 900MB data to the GPU
execute kernel in the GPU
transfer 900MB data from the GPU
```

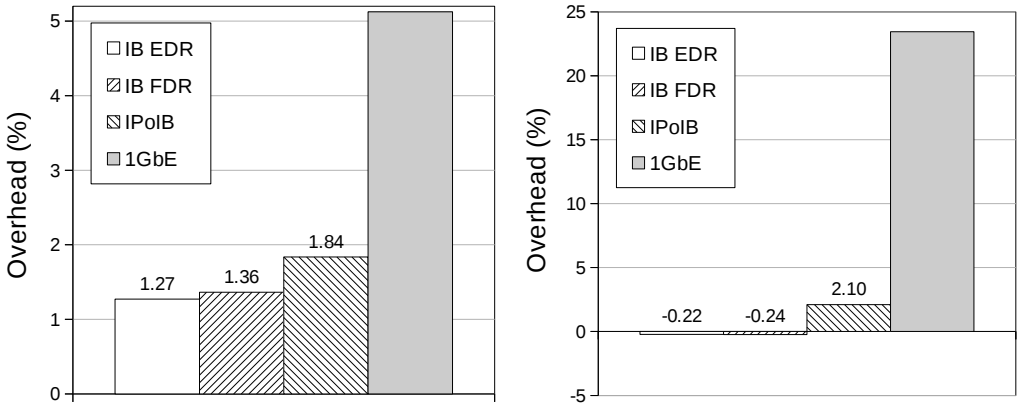
The 1300 MB of data in execution blocks 1 and 2 (as well as the 900 MB of data in execution block 3) are hold in 9 regions of GPU memory. Therefore, when the application is migrated, the rCUDA framework must allocate 9 memory regions in the destination

GPU, must copy the 9 memory regions from source to destination GPUs, and finally must release the 9 regions at the source GPU.

The testbed used in this analysis consists of a cluster of 1027GR-TRF Supermicro nodes featuring two Intel Xeon E5-2620v2 processors (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 memory at 1600 MHz. The nodes of the cluster also include one FDR and one EDR InfiniBand adapters, which provide 56 Gbps and 100 Gbps, respectively. Moreover, they include a Tesla K20 GPU. Linux CentOS 7.3 was used along with CUDA 8.0 (NVIDIA driver 367.48) and Mellanox OFED 3.4-2.0.0 (InfiniBand drivers and administrative tools).

Figure 6.4 shows the execution times for the synthetic and GPU-BLAST applications with CUDA and rCUDA when no migration has been performed. Executions have been repeated 5 times in order to average results. Furthermore, rCUDA executions have been carried out over several interconnects and communication protocols: RDMA over EDR InfiniBand, RDMA over FDR InfiniBand, TCP/IP over InfiniBand (in this case both the EDR and FDR InfiniBand adapters achieve the same performance), and TCP/IP over 1 Gb Ethernet. It can be seen in the figure that execution times for both applications when a remote GPU is used with rCUDA are very similar to the execution times with CUDA using a local GPU when RDMA is used to access the remote GPU. When TCP/IP is used over InfiniBand, execution time is slightly increased. On the other hand, the much lower bandwidth and higher latency of 1 Gb Ethernet causes that execution time is noticeably increased, specially in the case of the GPU-BLAST application given that, in total, 7000 MB of data are moved forth and back during the entire execution of the application. These overheads are detailed in Figure 6.5.

It can be seen in Figure 6.5(a) that the synthetic application increases execution time by 1.27% when it is executed using a remote Tesla K20 GPU using the EDR InfiniBand fabric. This overhead is slightly increased when an FDR InfiniBand network is used. In this case, the lower network bandwidth increases execution time by 1.36%. In these two cases, RDMA was used over InfiniBand. When TCP/IP is used instead, overhead is increased, lengthening execution time by 1.84%. Finally, the use of the much slower 1 Gb Ethernet network fabric increases execution time by 5%.

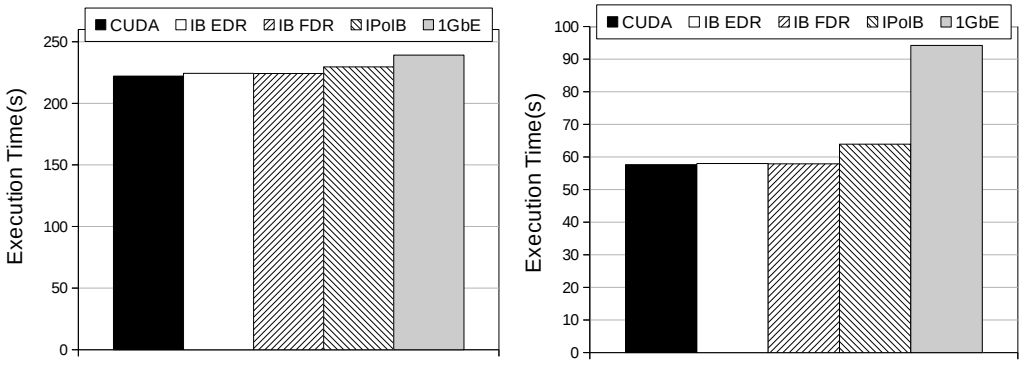


(a) Synthetic application. (b) GPU-BLAST application.

FIGURE 6.5: Overhead introduced in executions of Figure 6.4 because of executing the applications with rCUDA using a remote GPU instead of using a local one with CUDA. No migration has been performed.

Regarding the overheads experienced by the GPU-BLAST application, it is remarkable that when rCUDA is used with RDMA over EDR and FDR InfiniBand, execution time is slightly reduced. This reduction in execution time may seem to be senseless, given that the application is accessing a remote GPU across the network fabric instead of using a local GPU across the PCIe link. However, this reduction in execution time when the application is executed with rCUDA leveraging the RDMA features of InfiniBand is a well known effect [9] and it is due to higher bandwidth (with respect to CUDA) achieved by rCUDA when transferring pageable memory to/from the GPU as well as the better performance of synchronization points, such as calls to `cudaDeviceSynchronize` or `cudaStreamWaitEvent`, which take more time when using CUDA with a local GPU than when using the rCUDA middleware with a remote GPU. On the other hand, when TCP/IP is used, either over InfiniBand or over 1 Gb Ethernet, overhead is noticeably increased, specially in the latter case. The reason for the much larger overhead of the GPU-BLAST application with respect to the synthetic application when using 1 Gb Ethernet is based on the fact that the latter moves much more data to/from the GPU than the former.

Once the execution times of the two applications (without migration) have been revisited, next step is to analyze their execution time when they are live migrated during their execution. These execution times are depicted in Figure 6.6 (notice that execution times for CUDA are the same as in Figure 6.4; they have been included in the graph just for



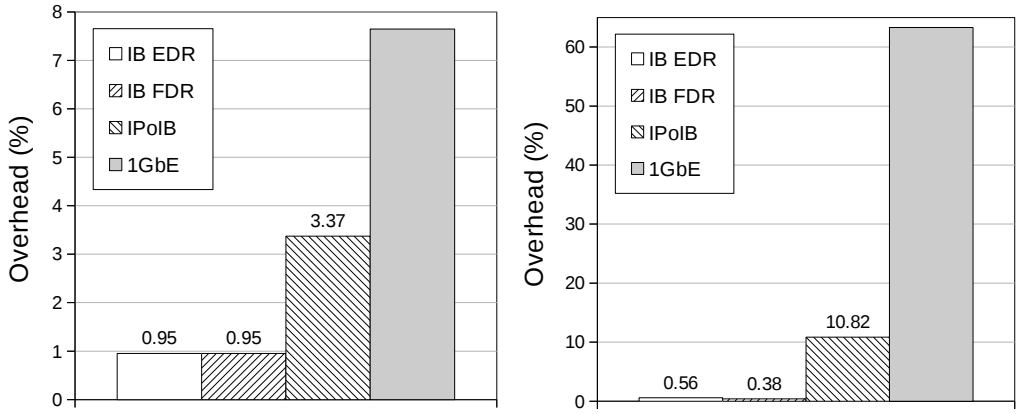
(a) Synthetic application.

(b) GPU-BLAST application.

FIGURE 6.6: Execution time using CUDA and rCUDA. Executions with rCUDA have suffered one live migration process in order to move the GPU-part of the application to another GPU.

comparison purposes). It can be seen in the figure that, as expected, the use of RDMA over InfiniBand provides the smallest migration overheads given the superior features of this communication mechanism. On the contrary, the use of TCP/IP increases execution time, as can be clearly seen if comparing the results in Figure 6.6 with those in Figure 6.4.

The exact values of the overhead introduced in each of the migration scenarios can be seen in Figure 6.7. It can be seen that migration overhead is negligible when RDMA is used. It is interesting to remark that in these RDMA-based cases, overhead for the synthetic application is larger than for the GPU-BLAST application, despite of having to move almost twice data for the latter than for the former. The reason is that the much larger amount of memory regions to be moved to the destination GPU, in the case for the synthetic application, contributes to increase latency (many more calls to CUDA, most of them not transferring data, such as `cudaMalloc` or `cudaFree`). This much larger amount of CUDA calls increase latency, which is not compensated by the smaller amout of data to be moved (700 MB vs 1300 MB) because moving that data using RDMA requires, in general, very low latency. In addition, the 9 memory regions of GPU-BLAST are much larger than the 1000 memory regions of the synthetic application. The noticeably difference in memory region size also contributes to overhead, given that attained bandwidth for small data transfers is always smaller than achieved bandwidth for bigger data transfers. Therefore, in the case for the synthetic application, 1000 small memory regions are copied from source to destination GPUs with lower bandwidth than the 9 much bigger memory regions of GPU-BLAST.



(a) Synthetic application.

(b) GPU-BLAST application.

FIGURE 6.7: Overhead introduced because of carrying out one live migration while executing the applications with rCUDA using a remote GPU.

On the contrary, migrating the GPU-part of the application by leveraging the TCP/IP protocols (either over InfiniBand or over 1 Gb Ethernet) presents the opposite trend. In these cases, it can be seen that the GPU-BLAST application presents more overhead than the synthetic application due to the lower available network bandwidth, which causes that the larger amount of data to be transferred for GPU-BLAST finally has a larger contribution to overhead. In the case of the GPU-BLAST application, this overhead is larger than 60% when 1 Gb Ethernet is used.

Next step should be to analyze the exact amount of time required to carry out the migration for each of the applications. This is, however, a difficult task, the reason being the way that migration has been implemented within the rCUDA framework, which introduces a big uncertainty when measuring the exact time required to perform the migration of the GPU part of an application. Effectively, remember that migration within rCUDA is triggered by a TCP/IP connection being received at the rCUDA server providing GPU services to the application to be migrated. This TCP/IP connection informs the rCUDA server about which of its many clients should be migrated and which should be the destination GPU for that particular client. In this way, receiving this TCP/IP connection at the rCUDA server is completely asynchronous with the execution of the application at the client node. Therefore, when the connection requesting migration is received at the rCUDA server, the application can be at any point of its execution and this could cause a lot of noise when measuring the time required to carry out the live migration. This noise might be caused by several reasons. For instance, if migration

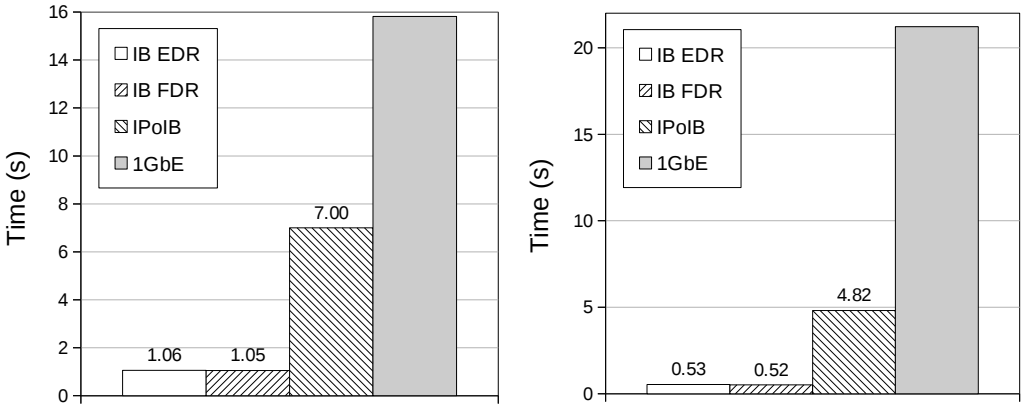
measurements are repeated several times, for each of those experiments the application could have allocated a different amount of memory regions, given that usually memory regions are not allocated immediately after execution is started. In this regard, a different number of memory regions (probably with different sizes) will render different migration times. In a similar way, for several migration experiments, the application could have launched different amounts of kernels. Given that one of the first steps of the migration process is to wait until all the kernels have completed their execution, then each of the migration experiments would provide a different measurement for the amount of time required to perform the migration process.

In order to avoid, to some extent, the noise in the measurements mentioned above, several possibilities are feasible. The first one is to insert, in the application source code, the necessary calls to trigger migration. Basically, the role of these calls would be to create the TCP/IP connection to the rCUDA server and send it the appropriate data to trigger the migration. However, although this possibility is feasible, it would mean to modify the source code of the application, which precisely is one of the actions not required by the rCUDA middleware, which does not require applications to be modified. Therefore, we prefer to use this option only if other possibilities are not available.

Another possibility for measuring migration time would be to randomly trigger several migrations along the same execution of an application. Let us say that n migrations are triggered during the execution of the application. Once the application has completed execution after carrying out the n migrations, the time measurement obtained would contain the time for executing the application with rCUDA plus the time for carrying out the n migrations. Let us refer to this time as $t_{migration}$. Once $t_{migration}$ has been obtained, we can subtract from it the amount of time required to execute the application leveraging rCUDA without performing any migration (let us refer to this time as t_{rCUDA}) and divide by the amount of migrations carried out (n). That is, with these time measurements, average migration time could be estimated as:

$$(t_{migration} - t_{rCUDA}) \ / \ n$$

Figure 6.8 shows the estimated average migration time measured as explained above, where n is equal to 5. Additionally, the experiment has been repeated 5 times in order



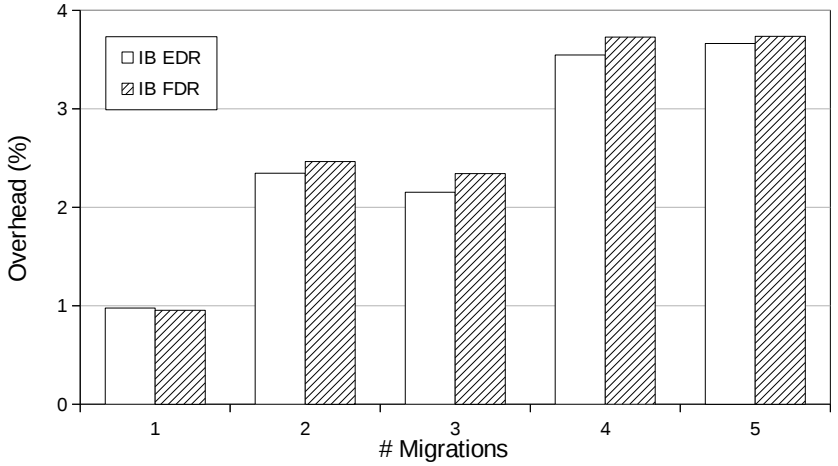
(a) Synthetic application. (b) GPU-BLAST application.

FIGURE 6.8: Estimated average time required to perform one live migration while applications are in execution.

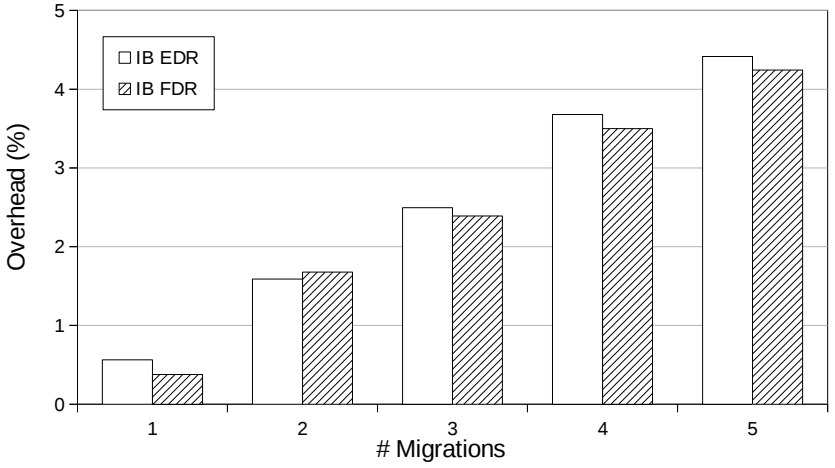
to gather more stable results. As in the previous figures, it can be seen that when RDMA is used over InfiniBand, the cost of migrating the GPU part of the application is very low. On the contrary, when TCP/IP is used, either over InfiniBand or over 1 Gb Ethernet, the cost of migrating the application is noticeably increased, specially when the low performance 1 Gb Ethernet network fabric is used. This can be seen in the case of the GPU-BLAST application, where 1300 MB of data has to be moved from source to destination GPUs. To that end, data (1300 MB) is first moved from source GPU to the node executing the application and afterwards it is moved from that node to the destination GPU. That is, two copies are performed[2] thus moving 2600 MB of data across the 1 Gb Ethernet fabric in total. Moving this amount of data takes 20 seconds approximately, which is the cost of the migration for the GPU-BLAST application when making use of the 1 Gb Ethernet network, as shown in Figure 6.8(b).

Finally, Figure 6.9 shows the overhead, with respect to the executions with CUDA using a local GPU, of a series of executions with rCUDA where an increasing number of randomly-triggered live migrations have been applied during the execution of the applications. Up to 5 migrations have been considered. For each of the amounts of

[2]Notice that migration support within rCUDA is based on P2P CUDA copies (copies between GPUs located in the same or different cluster nodes, depending on the exact migration scenario). This type of copies are implemented in a very efficient way when InfiniBand RDMA features are leveraged (source and destination GPUs in different nodes). In this case, data is directly moved from source to destination GPUs. However, when TCP/IP is used, P2P copies within rCUDA do not directly move data from source to destination GPUs but the node executing the application is used as an intermediate buffer. This inefficiency will be fixed in future implementations of the rCUDA middleware by also directly copying data among GPUs in the TCP/IP scenario.



(a) Synthetic application.



(b) GPU-Blast application.

FIGURE 6.9: Overhead with respect to CUDA when applications are live migrated up to five times during their execution.

migrations, experiments have been repeated 5 times in order to gather more stable results. It is very interesting to remark the results for the synthetic application, shown in Figure 6.9(a), which can be a very nice example of how difficult can it be to measure the overhead introduced by the migration process. In particular, it can be seen in the figure that when three consecutive migrations are applied during the execution of the synthetic application, overhead is lower than when 2 migrations are applied. In a similar way, when 5 migrations are applied during application execution, overhead is similar to when 4 migrations are considered. Several might be the reasons for these results, as discussed above (different amount of memory regions to be migrated, different waiting times until kernels under execution have finished, etc).

6.5 Conclusions

This paper has presented the first results of an ongoing work consisting on providing migration support within the rCUDA remote GPU virtualization middleware. Although providing this kind of support within GPU virtualization frameworks is not novel, the implementation carried out for the rCUDA middleware presents a better overall architecture, which is carefully devised to be integrated with job schedulers at different levels. In this regard, contrary to the rest of implementations of the GPU migration mechanism in other GPU virtualization frameworks, in the rCUDA implementation it is the job scheduler the one that triggers the migration process as well as the one that selects the destination GPU, according to the scheduling and energy efficiency policies implemented by the global scheduler. Additionally, the GPU migration implementation presented in this paper is the only one existing for modern CUDA versions.

Performance results show that migration is feasible and its overhead is very low when the InfiniBand network is used in the cluster. Similar extraordinary performance results are expected for other network fabrics that also provide RDMA capabilities, such as the RoCE interconnect. Regarding TCP/IP networks, it has been shown that the overall overhead is relatively low when the bandwidth provided by the network is in the order of a few tens of Gbps. Even when the 1 Gb Ethernet network fabric is used, migration overhead could be small if the application execution time is long enough when compared to the amount of data to be moved between source and destination GPUs.

Acknowledgments

This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/XX. Authors are also grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] VMware. VMware virtualization for desktop & server. `http://www.vmware.com/`, 2015. Accessed 15 April 2017.

[2] Xen. The Xen Project. `http://www.xenproject.org/`, 2013. Accessed 15 April 2017.

[3] KVM. Kernel-based Virtual Machine. `http://www.linux-kvm.org/`, 2010. Accessed 15 April 2017.

[4] VirtualBox. Oracle VM VirtualBox. `http://www.virtualbox.org/`, 2015. Accessed 15 April 2017.

[5] A.A. Semnanian, J. Pham, B. Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *2011 Eighth International Conference on Information Technology: New Generations (ITNG)*, pages 719–724, April 2011.

[6] Mellanox. ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual. `http://www.mellanox.com/`, 2013. Accessed 15 April 2017.

[7] Intel. Intel Ethernet Server Adapter I350. `http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-i350-server-adapter-brief.html`, 2013. Accessed 15 April 2017.

[8] NVIDIA. NVIDIA GRID ACCELERATED VIRTUAL DESKTOPS AND APPS. `http://images.nvidia.com/content/grid/pdf/188270-NVIDIA-GRID-Datasheet-NV-US-FNL-Web.pdf`, 2016. Accessed 26 March 2017.

[9] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, Middleware Industry '15, 2015.

[10] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proc. of the Euro-Par Parallel Processing, Euro-Par*, pages 379–391, 2010.

[11] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. DS-CUDA: A Middleware to Use Many GPUs in the

Cloud Environment. In *Proc. of the SC Companion: High Performance Computing, Networking Storage and Analysis, SCC*, pages 1207–1214, 2012.

[12] S. Iserte, J. Prades, Carlos Reaño, and F. Silla. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[13] I. S. Haque and V. S. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 691–696, 2010.

[14] NVIDIA. CUDA C Programming Guide. Design Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2017. Accessed 26 March 2017.

[15] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Proc. of the IEEE Parallel and Distributed Processing Symposium, IPDPS*, pages 1–11, 2009.

[16] Tyng Yeu Liang and Yu Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Proc. of the IEEE Advanced Information Networking and Applications Workshops, WAINA*, pages 141–146, 2011.

[17] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt*, pages 17–24, 2009.

[18] NVIDIA. CUDA Runtime API. API Reference Manual. `http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf`, 2016. Accessed 26 March 2017.

[19] Carlos Reaño and F. Silla. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *2015 IEEE International Conference on Cluster Computing*, 2015.

[20] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413, 2009.

[21] Paul Hargrove and jason Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics Conference Series*, 46(1), September 2006.

[22] Lin Shi, Hao Chen, and Ting Li. *Hybrid CPU/GPU Checkpoint for GPU-Based Heterogeneous Systems.*

[23] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent Accelerator Migration in a Virtualized GPU Environment. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 124–131, 2012.

[24] Khronos OpenCL Working Group. *OpenCL 1.2 Specification*, 2011.

[25] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. c. Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *2012 Innovative Parallel Computing (InPar)*, 2012.

[26] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012.

[27] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, 2010.

[28] P. Kegel et al. dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In *IPDPSW*, 2012.

[29] ZiZhuo Zhang, Xinhao Xu, Mochi Xue, Jiajun Wang, Zhengwei Qi, and Yaozu Dong. gHA: An Efficient and Iterative Checkpointing Mechanism for Virtualized GPUs. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, 2016.

[30] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: Using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

Chapter 7

GPU-Job Migration: the rCUDA Case

Javier Prades, Federico Silla. **IEEE Transactions on Parallel and Distributed Systems** - Volume: 30 - Issue: 12 - Dec. 1 2019 - Pages 2718 - 2729 [1]
https://doi.org/10.1109/TPDS.2019.2924433

Abstract

Virtualization techniques have been shown to report benefits to data centers and other computing facilities. In this regard, not only virtual machines allow to reduce the size of the computing infrastructure while increasing overall resource utilization, but also virtualizing individual components of computers may provide significant benefits. This is the case, for instance, for the remote GPU virtualization technique, implemented in several frameworks during the recent years.

The large degree of flexibility provided by the remote GPU virtualization technique can be further increased by applying the migration mechanism to it, so that the GPU part of applications can be live-migrated to another GPU elsewhere in the cluster during execution time in a transparent way.

In this paper we present the implementation of the migration mechanism within the rCUDA remote GPU virtualization middleware. Furthermore, we present a thorough performance analysis of the implementation of the migration mechanism within rCUDA. To that end, we leverage both synthetic and real production applications as well as three different generations of NVIDIA GPUs. Additionally, two different versions of the InfiniBand interconnect are used in this study. Several use cases are provided in order to show the extraordinary benefits that the GPU-job migration mechanism can report to data centers.

Keywords: CUDA, GPU, virtualization, migration, rCUDA.

7.1 Introduction

Virtualization has become a very important mechanism to increase the efficiency of data centers. Virtualization allows acquisition costs to be better tailored to the real computing needs while reducing energy footprint by consolidating servers. The concept of virtualization can be applied at different levels, as exposed below.

Firstly, the virtualization mechanism can be applied at the computer level, leading to the well known and widely used virtual machine frameworks, which allow several virtual machines to be concurrently executed in a real computer, sharing its resources and hence increasing overall utilization. As a consequence of the widespread use of virtual machines, processor manufacturers incorporate an increasing virtualization support into their products [1].

In the context of virtual machine solutions, virtualization can also be applied at the device level in order to provide support to virtual machines. For instance, some network adapters include virtualization features [2][3] which allow the adapter to be replicated, at the logical level, so that different replicas of the network card are assigned to different virtual machines. In a similar way, graphics processing units (GPUs) have recently included some virtualization support. For instance, the GRID GPU by NVIDIA [4] can be shared among virtual machines.

In addition to provide support to virtual machines, virtualization of individual devices may also be intended to provide an increased degree of flexibility at the cluster level. For example, networked disks enable sharing a file system across a cluster. In a similar way, the recent remote GPU virtualization technique, implemented in frameworks like rCUDA [5], GVirtuS [6], DS-CUDA [7], or FlexDirect by Bitfusion [8], allows GPUs to be logically detached from the node where they are installed thus creating a pool of GPUs that can be remotely accessed from any node in the cluster. This provides great flexibility when using GPUs.

The large degree of flexibility provided by the remote GPU virtualization technique can be further increased by allowing the GPUs assigned to a given application to move around in the cluster while the application is in execution. This movement means that the application is initially provided one or more GPUs in one or more nodes of the cluster

but, during application execution, the GPU part of the application is transparently migrated to other GPU (or GPUs) elsewhere in the cluster. This migration of the GPU part of an application can provide many different benefits to data centers and other computing facilities.

Probably, the most immediate benefit of migrating the GPU part of an application is to support GPU server consolidation. In this regard, notice that resource utilization in data centers evolves over time, depending on the exact workload applied at every moment. Therefore, at some point in time, the utilization of the GPUs in the cluster may be uneven. That is, some nodes may present high GPU utilization whereas GPUs located in other nodes may be much less utilized. In this scenario it would be useful to consolidate GPU jobs into a smaller number of servers, so that nodes becoming free can be switched off.

Other benefit of GPU-job migration is related to the efficient management of different user priorities in a data center, as it will be shown later in Section 7.5. Carrying out GPU load balancing across the cluster is also possible.

In this paper we present the implementation of the GPU migration mechanism within the rCUDA remote GPU virtualization middleware. Up to our knowledge, this is the first proposal for a remote GPU virtualization middleware to include migration capabilities in the context of CUDA. Our proposal provides more flexibility to data centers than previous proposals, as revisited in Section 7.3. Additionally, we present a thorough performance evaluation of this implementation when applied to different real applications. Three generations of NVIDIA GPUs and two versions of the InfiniBand network fabric are used in this performance analysis. This analysis is the main contribution of this paper with respect to [9], which showed a non-mature yet implementation of the migration mechanism within rCUDA as well as a naive performance analysis. Notice that the proposal in this paper is not only useful for cloud infrastructures but it can also be applied to applications running in bare metal.

The paper is organized as follows. Section 7.2 presents a brief revision of the rCUDA middleware. Next, Section 7.3 provides a review about how the GPU migration mechanism has been implemented within different GPU virtualization frameworks whereas

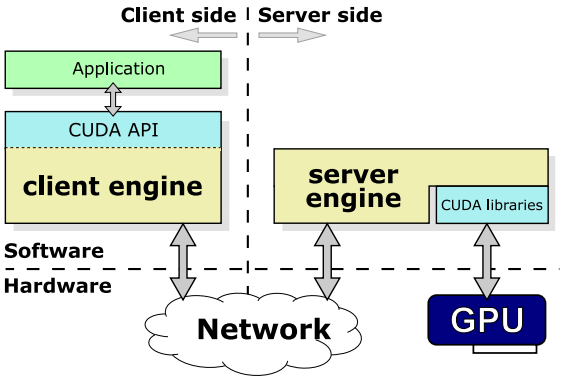


FIGURE 7.1: General organization of remote GPU virtualization frameworks.

Section 7.4 presents how migration is implemented in the context of the rCUDA middleware. Section 7.5 presents a thorough performance analysis of using the migration mechanism within the rCUDA middleware. Finally, Section 7.6 concludes this work.

7.2 About Remote GPU Virtualization

Several software-based GPU sharing solutions have been developed in the context of CUDA during the recent years. All of them aim to offer the same API as the NVIDIA CUDA Runtime API does. Figure 7.1 depicts the architecture usually deployed by these GPU virtualization frameworks, which follow a distributed client-server approach. The client part is installed in the cluster node executing the accelerated application whereas the server side runs in the node owning the actual GPU. The architecture depicted in Figure 7.1 is used in the following way: the client middleware receives a CUDA request from the application and forwards it to the server middleware. In the server side, the middleware receives the request and interprets and forwards it to the GPU, which completes the execution of the request and returns the execution results to the server middleware. Finally, the server sends back the results to the client side, which forwards them to the accelerated application.

Among the several remote GPU virtualization solutions, we focus on rCUDA (remote CUDA), which supports version 9.2 of CUDA, being binary compatible with it, what means that CUDA programs do not need to be modified in order to use rCUDA. Furthermore, it implements the entire CUDA API (except for graphics functions and NVIDIA's Unified Virtual Memory (UVM), which is partially supported). rCUDA provides specific

support for different interconnects. Currently, two modules are available: one intended for TCP/IP compatible networks, and another one specifically designed for the InfiniBand and RoCE interconnects, which make use of RDMA. Furthermore, security and isolation among applications sharing a given rCUDA server is achieved by creating a new GPU context for each of the client applications arriving at the server. In this way, different applications cannot see each other and, in case one of the clients die, the GPU contexts for the other clients can safely continue execution. Compared to other publicly available remote GPU virtualization frameworks developed in academia, the rCUDA middleware provides the best performance [10]. In this regard, the rCUDA middleware achieves near to native performance[11][12][13]. Also, contrary to commercial solutions such as FlexDirect, rCUDA provides support for a wide scope of applications.

7.3 Related Work on GPU Migration

Migrating GPUs has been addressed in the past in very few works, although none of them was proposed in the context of CUDA. One of these works is presented in [14]. This proposal, intended for OpenCL instead of CUDA, is implemented within the VOCL remote GPU virtualization framework. In this proposal, every time a memory allocation OpenCL function is called, it is intercepted and all the necessary information about the reserved memory areas (starting address, length, etc) is recorded, so that it can be later used for migration purposes. Furthermore, this framework requires that kernels running in the GPU are completed before migration begins. Moreover, a couple of functions are provided in order to trigger migration from the executing application, thus requiring the source code of applications to be modified. On the contrary, in our proposal, migration is not triggered by the application (source code is not modified) but by an external signal. This signal, in the form of a TCP/IP connection to the rCUDA server, is originated at the job scheduler, for instance.

Recent implementations of GPU live migration can be found in NVIDIA's GRID [4] and Intel's GPUs [15], which allow the whole virtual machine (including both its CPU part as well as its GPU part) to be migrated between nodes in a cluster. However, contrary to our proposal, these technologies do not decouple GPUs from CPUs but they

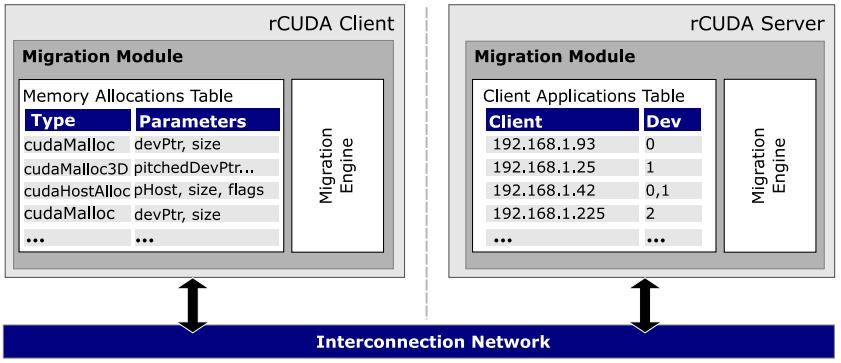


FIGURE 7.2: Migration modules inside rCUDA client and server.

are tied together and must be migrated at the same time, thus not allowing the benefits provided by our proposal, such as GPU server consolidation, GPU load balancing, efficient management of user priorities, etc. Furthermore, these solutions require the usage of virtual machines to work whereas our proposal can migrate GPUs regardless of using virtual machines or bare metal. The techniques to implement GPU migration and GPU checkpointing are similar. Thus, it is worth to also consider works on GPU checkpointing. In this regard, for instance, in [16] a prototype of a checkpointing framework, named CheCUDA, is presented. It only supports a fraction of the functions within the old CUDA 7.0. In order to know which are the GPU memory areas to be included in the checkpoint, CheCUDA provides a set of wrappers to some of the basic cuMemAlloc functions in the Driver and Runtime APIs. However, this solution does not support multi-threaded applications neither applications using several GPUs. Another proposal is described in [17], where a non-mature hybrid checkpointing technology intended to support checkpointing a running GPU kernel at any time during its execution is presented. The proposal is transparent to the programmer (no source code modification is required), although it is based on the debug interface of CUDA, therefore forcing kernels to run in synchronization mode and causing a large execution overhead. One more proposal, described in [18], supports UVM.

Finally, a proposal for checkpointing, named gHA, is described in [19] for Intel GPUs. gHA does not need any modification of the application source code. Also, no modification to the guest driver is required. Furthermore, gHA saves the Intel GPU registers during a kernel execution so that it does not have to wait for the running kernel to be completed.

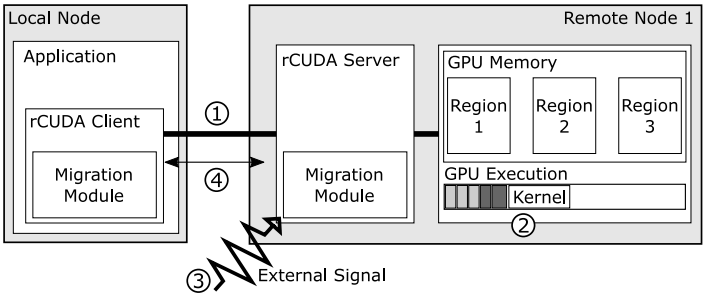
7.4 Implementing GPU Migration in rCUDA

In this section we present the main details of the implementation of the migration mechanism within the rCUDA middleware as well as its operation.

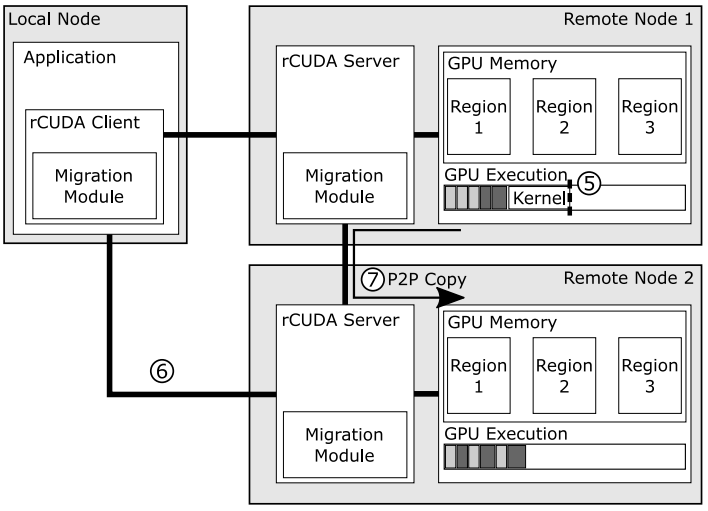
Figure 7.2 shows the migration module included both in the rCUDA client and in the rCUDA server. These modules comprise a migration engine where the logic that carries out the actual migration process is integrated.

The migration engines at the client and server sides are responsible for storing the necessary information to support migration. In the server side, the migration engine manages the information related to active client applications, which is stored in the Client Applications Table. The migration engine in the rCUDA server is also responsible for handling migration requests and coordinating them with the migration module in the corresponding client. In the client side, the migration engine tracks all memory allocation/deallocation functions. The Memory Allocations Table stores the GPU memory allocation information so that, whenever a migration between GPUs is requested, this information is used to recreate the memory allocations in the new GPU.

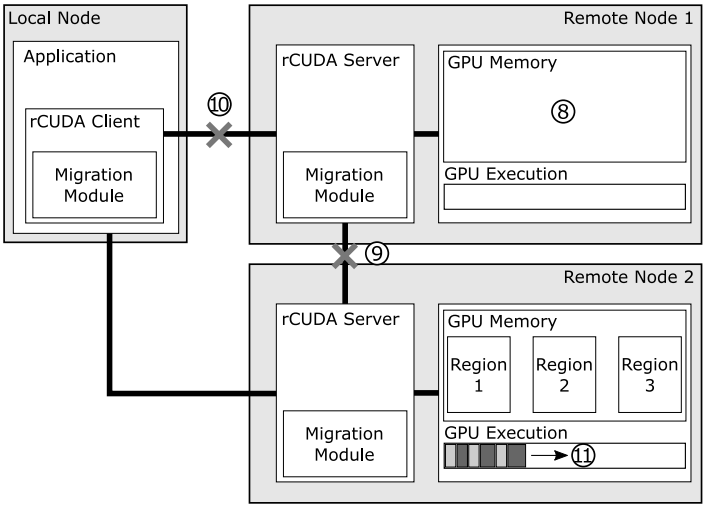
The operation of the migration module within rCUDA is shown in Figure 7.3. It can be seen in this figure how an accelerated application is migrated using the rCUDA middleware. At step 1 in Figure 7.3(a), the application starts execution and the connection between the rCUDA client and the rCUDA server is established. Once this initial connection is set up, the application continues its usual execution. In this particular example, as it can be seen in step 2, the application performs three memory allocations (light gray boxes in the "GPU execution" queue) followed by 2 copies from host to device (dark gray boxes) in order to fill memory regions 1 and 2 previously allocated in the GPU memory. Finally, the application launches a kernel, which will operate with the data located in regions 1 and 2. This kernel will store the results in region 3. Notice that the information about these three memory allocations was stored in the Memory Allocations Table of the client migration module when the associated CUDA calls were intercepted at the client node. Some time later, during the execution of the aforementioned kernel, an external signal (coming from a resource scheduler, for instance) arrives at the server migration module, as shown in step 3. This external signal is a TCP connection and has associated the necessary information to carry out the migration: client identifier as



(a) The application starts execution and, at some point in time, the migration signal, triggered by the resource scheduler, arrives at the rCUDA server.



(b) The memory is copied from source GPU to destination GPU in another node of the cluster.



(c) Resources at the initial rCUDA server are released and execution continues in the new GPU.

FIGURE 7.3: Complete operation of the migration module implemented within the rCUDA middleware.

well as source and destination GPUs. Finally, in step 4, the migration request will be communicated to the migration module in the corresponding client.

Figure 7.3(b) shows the core of the migration process. Once the migration request arrives at the rCUDA server and it is communicated to the client, a synchronization is performed in the source GPU, waiting for kernels to complete. In our example we can see in step 5 how the migration modules have to wait for the completion of the kernel being executed. Next, in step 6, a new connection between the rCUDA client and the new rCUDA server will be established. Once this connection is created, data must be copied between both GPUs (source and destination). To that end, for each of the regions stored in the Memory Allocations Table, a memory allocation will be performed in the destination GPU memory (light gray boxes) and afterwards the data for each of the regions is transferred directly from the memory of source GPU to the memory of the destination GPU by using the P2P copy module implemented in rCUDA [11] (dark gray boxes), as shown in step 7. This data copy is performed directly between the source and destination GPUs in case InfiniBand or RoCE are used (leveraging RDMA) whereas an intermediate copy involving the client node is required in case TCP/IP communications is used.

Figure 7.3(c) shows the final steps of the migration process. Memory regions in the original GPU are released in step 8. Then, the connection used for the P2P copies is destroyed (step 9) as well as the connection between the rCUDA client and the initial rCUDA server (step 10). Finally, the application continues execution in the new GPU (step 11).

There is an important final remark about migrating GPU jobs when different generations of GPUs are involved in the migration process. Notice that when using CUDA, there is no binary compatibility guarantee between GPU applications compiled for different generations of GPUs. That is, an application compiled for Kepler may not run on a Maxwell GPU and vice versa. Therefore, migrating a GPU application between different GPU generations may not be successful due to this lack of compatibility guarantee. Fortunately, the `nvcc` CUDA compiler provides options to generate binaries that can be run on different GPU generations. The `nvcc` compiler follows a compilation model based on two stages. In the first stage, an intermediate representation, called PTX, is generated. Later, in the second stage, it is used to generate the binary code for a

specific GPU generation. This binary code can either be generated at compile time or at execution time by using JIT (Just-in-Time) compilation. Each of the options presents pros and cons. If it is generated at runtime, then it will perfectly match the requirements of the GPU that is going to be used. However, some overhead will be introduced by the compilation during the execution of the application. On the other hand, if the binary code is generated at compile time, `nvcc` allows the generation of multiple translations of the same source code targeted for multiple GPU generations. At run time, these multiple translations, which are organized in Fatbinaries, will allow the CUDA driver to select the appropriate binary code based on the actual GPU. In summary, if a GPU binary code can be executed with CUDA in a set composed of several GPU generations (either because it is using JIT or Fatbinaries), then it will be possible to migrate that code with rCUDA among that very same set of GPU generations. The use cases presented in next section are an example of this, given that applications are migrated between Kepler and Pascal GPUs.

7.5 Performance Evaluation of GPU Migration with rCUDA

This section presents a performance study of our implementation of the GPU-job migration mechanism within the rCUDA middleware. We consider three different scenarios for this performance evaluation. In the first scenario, addressed in Section 7.5.1, a synthetic application will be used. In the second scenario, thoroughly introduced in Section 7.5.2, we consider real applications for the migration experiments. Finally, in Section 7.5.3 we leverage a series of use cases in order to exemplify the usefulness of migrating GPU jobs among cluster nodes.

The testbed used in all these analyses consists of a cluster of 1027GR-TRF Supermicro nodes which include one FDR and one EDR InfiniBand network adapters, which provide 56 Gbps and 100 Gbps, respectively. Moreover, they include three different generations of GPUs: an NVIDIA K20 GPU, an NVIDIA K40 GPU and an NVIDIA P100 device. Using these three different GPU models will allow us to better exercise the migration mechanism in this section.

7.5.1 Synthetic Application

Migrating a job among two GPUs located in different cluster nodes requires two different types of actions, both of them contributing to the migration overhead. First, every memory region allocated by the application in the source GPU has to be copied to the destination GPU. Second, it is required to properly manage these copies.

Regarding the first type of actions, the movement of the data in each region from the source GPU to the destination device consumes most of the migration time. This time depends not only on the exact network fabric used but it also depends on the exact size of the memory region to be moved, given that the maximum bandwidth attained by a network fabric is only achieved for data transfers beyond a minimum threshold. For instance, in the case of copying data with rCUDA among GPUs located in different cluster nodes, the maximum performance is achieved when data transfers are larger than 10 MB [11].

On the other hand, the time required for managing the data copies cannot be neglected. In this regard, the connection between rCUDA servers must be first established in order to later use P2P copies. Afterwards, for every memory region to be copied from source to destination GPUs, a call to a CUDA memory allocation function has to be carried out in the destination GPU prior to copying the data of that region from the source GPU. Additionally, once the data of that region has been copied, a CUDA memory deallocation function has to be executed in the source GPU. Calls to CUDA memory allocation/deallocation functions require some time to be executed and, therefore, the more memory regions the application allocated in the source GPU, the longer it will take to manage the migration process, as it will be shown later.

In order to understand the impact on performance of each of the parameters involved in the migration of GPU jobs, in this section we leverage a synthetic application so that different parameters can be controlled in an isolated way. The synthetic application implemented for this study takes as input parameters the total amount of GPU memory regions and the size of each region. Then, by using these input parameters, the application allocates n equally sized memory regions in the GPU. Although this application is extremely simple, using it in this first scenario will allow us to understand the behavior of the migration process.

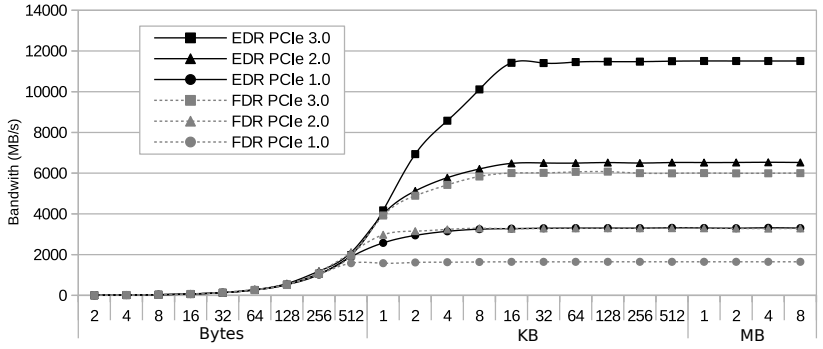


FIGURE 7.4: Bandwidth attained for several network configurations using different transfer sizes.

Regarding the network fabrics used in the experiments with the synthetic application, we have considered 6 different network throughputs in order to shed light to the performance results. First, we have leveraged FDR and EDR InfiniBand network fabrics, which make use of PCIe 3.0 x8 and PCIe 3.0 x16, respectively. Additionally, we have modified the PCIe settings in the testbed systems so that these network adapters were also used with PCIe 2.0 and PCIe 1.0 configurations. These additional configurations are intended to reduce network performance. The exact throughput of each of these configurations is shown in Figure 7.4 for transfer sizes ranging from 2 bytes up to 8 MB. It can be seen in this figure that we are considering effective transfer bandwidths ranging from 13.2 Gbps (FDR PCIe 1.0) up to 92 Gbps (EDR PCIe 3.0). Also, performance of EDR PCIe 1.0 and FDR PCIe 2.0 are almost identical.

Results obtained with the synthetic application are shown in Figure 7.5. For all the experiments depicted in this figure, a P100 GPU has been used. We have selected this GPU because it supports PCIe 3.0 x16, which provides a bandwidth equal to or larger than all the network configurations considered. In this way, the limiting factor in these experiments will be the exact network fabric configuration. Figure 7.5 also displays the performance of the migration process when the 1 Gbps Ethernet network is used. Notice that results for this network are presented only for comparison purposes, given that its low performance makes this network not to be an option for virtualizing GPUs among cluster nodes in production data centers.

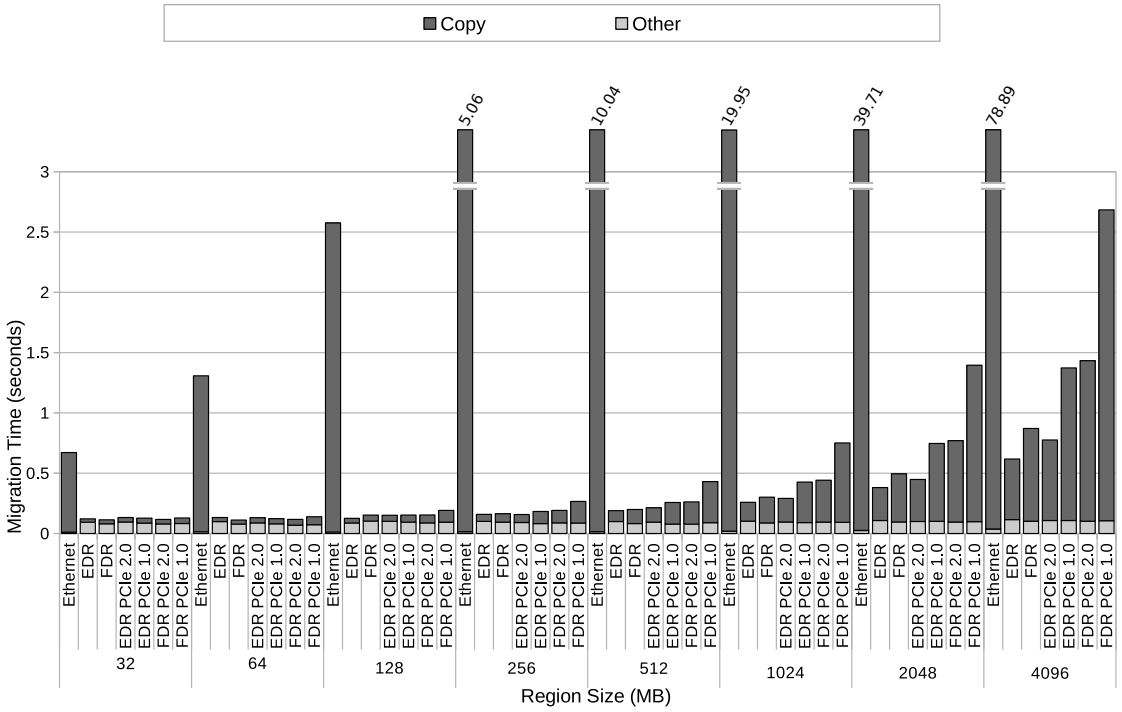
It can be seen in Figure 7.5(a) that the amount of time required by the migration process directly depends on the amount of data to be migrated. In this figure, the synthetic application has been configured to allocate only one memory region. Therefore, only one

call to the `cudaMalloc` function is carried out in the destination GPU. It can be seen in Figure 7.5(a) that total migration time has been split into copy time and "Other" time. Copy time refers to the time required to move the data from the original memory region in the source GPU to the newly allocated memory region in the destination GPU. "Other" time refers to the time required to manage the migration process (creation and destruction of the connection for P2P copies and calls to the `cudaMalloc` and `cudaFree` functions and other management tasks associated with the migration process in the particular implementation within rCUDA).

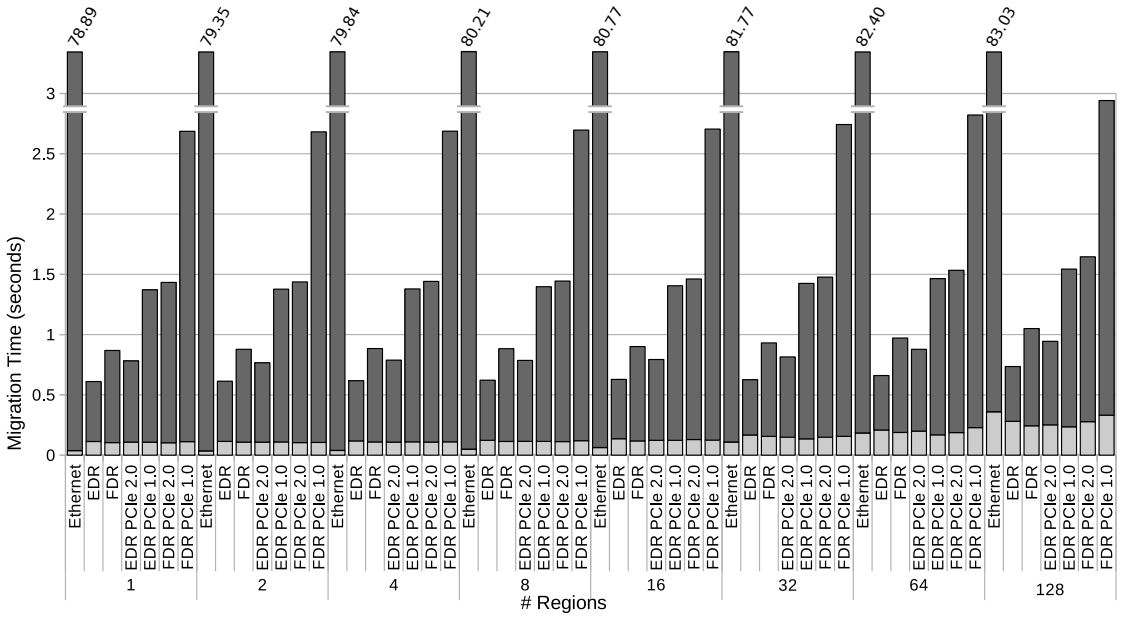
Figure 7.5(a) shows that the bandwidth attained by the underlying network directly impacts the performance of the migration process, as expected. It is worth noticing that a speed up of about 128x is attained in the case of EDR InfiniBand with respect to 1 Gbps Ethernet although difference in maximum bandwidth among both network fabrics is only 92x (1 Gbps bandwidth in the case of Ethernet versus 92 Gbps of effective bandwidth in the case of EDR InfiniBand). In a similar way, in the case of FDR InfiniBand, a speed up of about 91x is achieved although the theoretical speed up should be about 48x (FDR InfiniBand provides 48 Gbps of effective bandwidth). The reason for achieving a speed up much larger than the theoretical one is that when using InfiniBand networks we can directly copy data from the source GPU to the destination GPU by making use of the RDMA features included in these adapters whereas data transfers using the 1 Gbps Ethernet network require an intermediate copy because the RDMA feature is not present in the Ethernet adapters.

On the other hand, it is interesting to notice that the time for "Other" is noticeably larger for InfiniBand than for 1 Gbps Ethernet. The reason for these larger times is that the time "Other" when using InfiniBand includes the time for creating and destroying the TCP connections to the remote servers required to control data movement using RDMA. These TCP connections are not needed for 1 Gbps Ethernet as the RDMA feature is not available.

Figure 7.5(b) shows the impact on performance when varying the amount of memory regions that hold the data of a fixed size 4 GB memory area to be migrated. It can be seen that, for each of the network configurations considered, copy time remains almost constant regardless of the amount of memory regions. The reason is that even for the smallest region size, which is 32 MB when there are 128 regions, attained data transfer



(a) A single memory region is allocated in the GPU. Different region sizes are considered (from 32 MB up to 4 GB).



(b) Multiple memory regions are allocated in the GPU, accounting for a total of 4 GB GPU memory in all cases.

FIGURE 7.5: Time required to migrate a job among two P100 GPUs located in different nodes. A synthetic application is leveraged. Several configurations of the FDR and EDR InfiniBand network adapters are used. Performance for the 1 Gbps Ethernet network is also displayed.

TABLE 7.1: Amount of seconds required for management tasks in Figure 7.5(b).

	1	2	4	8	16	32	64	128
Eth	0.04	0.04	0.04	0.05	0.06	0.11	0.18	0.36
FDR	0.10	0.11	0.11	0.11	0.12	0.16	0.19	0.24
EDR	0.11	0.11	0.12	0.12	0.13	0.17	0.20	0.28

bandwidth is the maximum one because the size of data to be transferred is larger than 10 MB. On the contrary, time required for the migration management purposes (bar section "Other") increases as the amount of memory regions to migrate increases.

Table 7.1 shows the exact values for "Other" for the three main network fabrics. It can be seen in the table that management time increases as the amount of memory regions increases. Management times for FDR and EDR InfiniBand networks are similar. It is also noteworthy the fact that management times for 1 Gbps Ethernet are lower than for InfiniBand (due to the creation and destruction of the TCP connections as described before). However, as the number of migrated memory regions increases, the time required for migration management purposes increases more significantly when using 1 Gbps Ethernet. This is due to the worst latency of this network. The larger the number of regions to be migrated, the higher the number of memory allocation/deallocation calls. These calls do not include too much data (they simply notify the remote GPU) so they are very sensitive to the latency features of the network.

7.5.2 Real Applications

In this section we perform a study of the migration mechanism when it is applied to five different real applications. The applications are GPUBLAST [20], CUDASW++ [21], CloverLeaf [22], TeaLeaf [23] and CUDA-MEME [24]. Table 7.2 characterizes these applications. Data in this table has been gathered during the execution of the applications when using a remote K20 GPU with rCUDA along with FDR InfiniBand. Table 7.2 shows that the GPUBLAST application requires up to 1302 MB of GPU memory during its execution, which lasts for almost 134 seconds. Additionally, this application consists of 3 long running kernels that make a full usage of the GPU resources while in execution (see Figure 7.8). Average GPU utilization for the GPUBLAST application is about 33%. Similar data is presented for the other applications considered in this

TABLE 7.2: Characterization of the real applications used to analyze the migration mechanism.

Application	Execution Time (s)	rCUDA overhead (%)	# Kernels	Kernel Time (ms)			GPU Memory (Mbytes)			GPU Utilization (%)		
				Avg	Max	Min	Avg	Max	Min	Avg	Max	Min
GPUBLAST	134	2.14	3	14400	15600	12200	1207.4	1302	72	32.5	100	0
CUDASW++	15	-2.21	1	11500	11500	11500	762.5	931	72	70	100	0
TeaLeaf	156	9.81	1048557	0.03	0.11	0.0027	182.47	183	72	19	33	0
CUDA-MEME	213	7.4	2107	37.51	46. 29	22.38	157.63	162	72	38.4	69	0
CloverLeaf	271	3.35	405489	0.68	6.22	0.0027	1496.74	1502	72	97	99	0

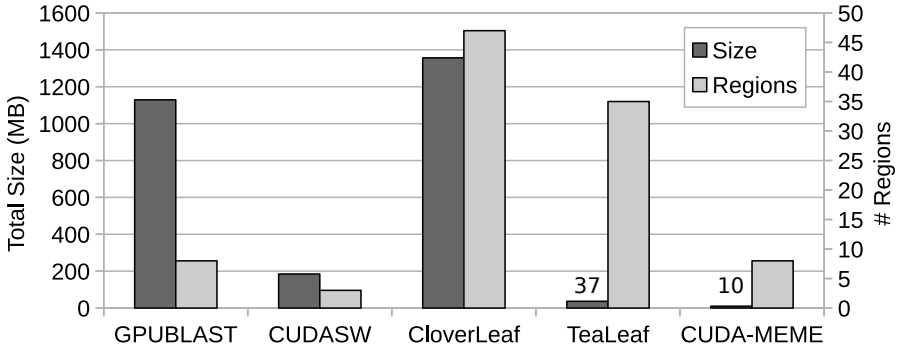


FIGURE 7.6: Memory configuration, in terms of total memory allocated and number of memory regions, for each of the applications considered.

section. Furthermore, Table 7.2 shows the overhead introduced by rCUDA with respect to the execution using a local K20 with CUDA.

Figure 7.6 presents additional information about the memory usage of these applications. In addition to show the GPU memory allocated by each of the applications, Figure 7.6 also displays the amount of memory regions allocated by each of them. In this regard, it can be seen that the GPUBLAST application allocates 8 different memory regions. This very same amount of regions is allocated by the CUDA-MEME application. On the contrary, CloverLeaf and TeaLeaf allocate a much larger number of memory regions. They allocate, respectively, 47 and 35 regions. Finally, the CUDASW++ application only allocates 3 memory regions.

Regarding the total amount of memory used by each of the applications, it can be seen, if comparing numbers in Figure 7.6 with numbers in Table 7.2, that values for memory usage seem not to match. The reason for the mismatch is that numbers in Figure 7.6 were gathered according to the information collected when intercepting the CUDA memory allocation calls with rCUDA. However, numbers in Table 7.2 were gathered by using the `nvidia-smi` application, which provides overall memory usage in the GPU, among many other parameters. In this regard, numbers in Figure 7.6 represent the exact amount of memory allocated by the application in the GPU. On the contrary, numbers in Table 7.2 represent total memory used in the GPU, which includes, for instance, the memory required to store the application context. Notice that this memory for the application context is allocated by the NVIDIA driver and not by the application. Therefore, when migrating the application, the memory used for the GPU context will not have to be moved to the destination GPU but a new context will be created in that GPU. After

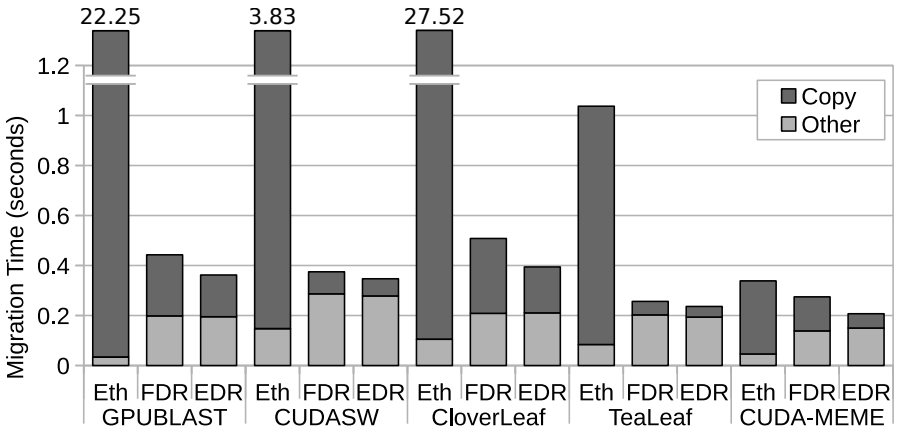


FIGURE 7.7: Service downtime for each of the applications considered. Migration was triggered at 25% execution time for each of the applications.

creating the new context in the destination GPU, all memory regions will be copied. In summary, memory sizes shown in Figure 7.6 can be seen as the amount of memory that has to be migrated among GPUs. That is, these memory regions, and memory sizes, are the only ones migrated in the experiments in this section, shown in Figure 7.7, for instance.

In order to measure migration time, an important concern is related to the exact moment when migration is triggered. Remember that after receiving the external signal triggering migration, kernels in execution in the GPU must be completed before beginning the migration process. In this manner, migrating an accelerated application among GPUs can be seen as a two step process where step 1 is just waiting for kernel completion and step 2 is moving data among GPUs. The first step has to do with kernels in execution at the time when the external signal triggering migration arrives whereas the second step has to do with the memory allocated in the GPU by the application.

It is important to notice that the time required for step 2 (moving data among GPUs) only depends on data size, amount of memory regions and underlying network fabric, as analyzed in previous section. However, the time required for step 1 (waiting for kernel completion) depends on the exact state of the execution of the application when the external signal arrives. As a consequence, we can differentiate among "total migration time" and "service downtime". The latter refers to how much time the GPU is out-of-service once migration begins after kernel completion. The former refers to the time required to restart the execution of the application in the target GPU since the arrival

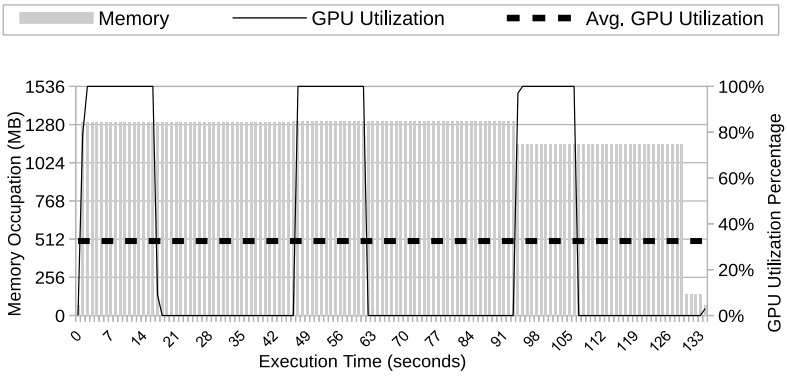
of the migration signal. Obviously, service downtime will always be less than or equal to total migration time given that total migration time includes service downtime plus the time waiting for kernel completion in the source GPU.

Regarding service downtime, notice that this amount of time is the overhead that the migrated application suffers due to the migration itself and it is independent of the execution time of kernels in the GPU. On the other hand, total migration time is the amount of time observed by the job scheduler when it triggers the signal to migrate the GPU part of an application.

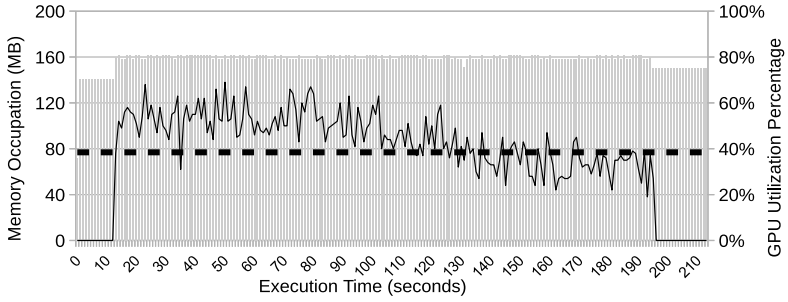
In order to perform a thorough analysis of service downtime for the applications under consideration, we triggered migration at three different points for each of the applications. These points were 25%, 50% and 75% of their execution time. Furthermore, as execution time of applications using remote GPUs depends on the exact network fabric used, these three points in time will thus depend on which network was leveraged for executing the application. Therefore, in order to analyze migration time for each application, we performed 9 experiments: migrating the application at 25%, 50% and 75% execution time when FDR InfiniBand and K20 GPU were used, migrating the application at 25%, 50% and 75% execution time when EDR InfiniBand and K40 GPU were used and, finally, migrating the application at the aforementioned execution time percentages when 1 Gbps Ethernet and K20 GPU were used. Notice that the exact points in time for each of the execution percentages vary depending on network fabric and GPU used.

Figure 7.7 shows the service downtime for each of the applications considered in our study. The three main network fabrics previously used in Section 7.5.1 are also employed in this figure. Remember that times in Figure 7.7 are the overhead experienced by the migrated applications. In order to gather the numbers in Figure 7.7, migration was triggered at 25% execution time. Results when migration was triggered at 50% and 75% execution times were almost the same. This fact points out that these applications have allocated very similar memory regions in the three points mentioned above, as it is shown in Figure 7.8 for two of the applications.

Regarding the results displayed in Figure 7.7, it can be clearly seen the impact on service downtime of the amount of data to migrate (shown in Figure 7.6). In this regard, the



(a) GPUBLAST.



(b) CUDA-MEME.

FIGURE 7.8: Evolution of memory occupancy and GPU utilization during execution time of two of the applications considered in this study. Average GPU utilization for each of the applications is also shown.

GPUBLAST, CUDASW++ and CloverLeaf applications present very different service downtimes depending on the exact network fabric used: the low performance of 1 Gbps Ethernet causes that service downtime is much larger than when InfiniBand is used. It can also be clearly seen that EDR InfiniBand reports smaller service downtime than FDR InfiniBand due to its much larger bandwidth. Nevertheless, it is important to remark that service downtime is very small (less than 0.5 seconds) when InfiniBand is used regardless of the exact version of this interconnect. On the other hand, when the amount of data to be migrated is very small, as it is the case for the CUDA-MEME application, service downtime is similar for both 1 Gbps Ethernet and InfiniBand.

Regarding the results for the CUDA-MEME application shown in Figure 7.7, there is an interesting issue regarding copy time. If transfer time is carefully analyzed (0.0921 seconds for FDR InfiniBand and 0.0563 seconds for EDR InfiniBand), it can be derived that transfer time is much larger than it should be, according to the bandwidth available in these networks. The reason for this higher transfer time is that this application

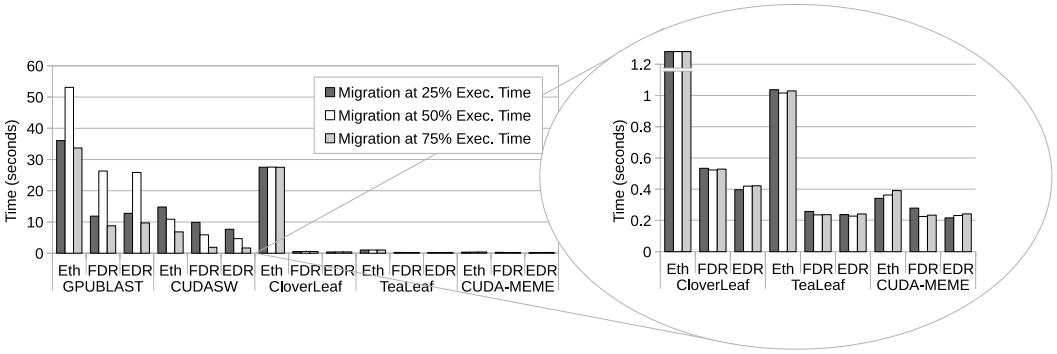


FIGURE 7.9: Total migration time for the five applications considered in this study. Time is measured since the arrival of the external signal triggering migration until the application resumes execution in the destination GPU.

allocates memory by using CUDA array memory instead of the regular memory. Transferring data allocated as a CUDA array with rCUDA is not as optimized as transferring regular data due to the geometry of the allocation. Therefore, a lower bandwidth is attained for these copies.

Figure 7.7 also shows the time required for managing the migration (bar section "Other"). For the CUDASW++ application, which only allocates 3 memory regions, management time is larger than for other applications with a much larger amount of regions, such as CloverLeaf or TeaLeaf. In order to explain this result, we analyzed the source code of the CUDASW++ application and found that this application makes use of host page-locked memory regions allocated with `cudaMallocHost` or `cudaHostAlloc` functions (in addition to the GPU memory regions). This type of regions need a special management given that not only GPU memory has to be migrated but also some host memory. The time required for managing these regions is accounted within the time required for managing the migration.

Finally, Figure 7.9 shows the delay between the arrival of the signal triggering migration until the application resumes execution in the destination GPU. This is total migration time. In this way, times displayed in Figure 7.9 include the waiting time until kernels in execution when the migration signal arrives are completed as well as the time to transfer the data from the source to the destination GPU. The figure displays the total migration times when migrating the applications at 25%, 50% and 75% of their execution times.

Two main conclusions can be derived from Figure 7.9. The first one is that total migration time greatly depends on the exact state of the application when the migration

signal arrives. This can be clearly seen for the GPUBLAST application. The second conclusion that can be derived from Figure 7.9 is that when an application executes a large amount of small kernels (as it was shown in Table 7.2 for the CloverLeaf, TeaLeaf and CUDA-MEME applications) then the waiting time for kernel completion is noticeably reduced and thus total migration time is decreased, as it is shown in the right side of the figure.

7.5.3 Use Cases for GPU-Job Migration with rCUDA

In the previous sections the performance of GPU migration was analyzed by using both synthetic and real applications. In this section we provide several use cases that show the usefulness of the GPU-job migration mechanism. Real applications will be used in this section.

In order to provide the reader with the right context for these use cases, it is important to understand that we envision GPU-job migration as a powerful tool that can be used by job schedulers to improve different metrics in the cluster. One of these metrics could be minimizing overall energy consumption, for instance. Another metric could be reducing application execution time. Furthermore, the job scheduler could deal with different user priorities. In this way, higher priority users should be provided better service whereas lower priority users may experience some delays depending on workload evolution. At the bottom stage of the priority stack, users with the lowest priority could just benefit from spare GPU cycles. That is, their jobs would execute as far as no other jobs belonging to higher priority users are present in the system. As soon as higher priority jobs enter the system, the lowest priority jobs should be preempted if required.

7.5.3.1 GPU Server Consolidation

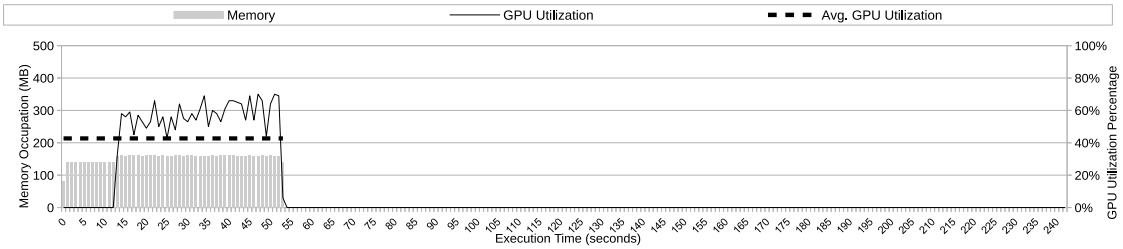
The first of the examples about the usefulness of the GPU-job migration mechanism is devoted to server consolidation. The consolidation technique is specially appealing when several GPU servers in the cluster present low to medium GPU utilization. In this scenario, GPU utilization in those servers could be increased by aggregating jobs from GPUs in different servers into a single GPU. By increasing GPU utilization, a better usage of energy is made. Additionally, if the servers that are emptied are later switched

off, then energy efficiency is noticeably increased. It is important to remind that only the GPU part of applications is migrated.

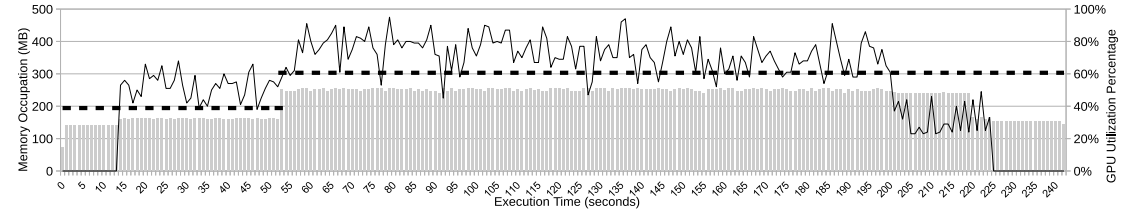
In order to implement this idea, the logic to decide whether to consolidate servers and which should be the GPU-jobs to migrate could be placed into the job scheduler. Additionally, the job scheduler should be enriched in order to gather information about the utilization of the GPUs in the cluster. In this way, once the job scheduler finds out that some of the GPUs in the cluster present a utilization under a given threshold, it could decide to consolidate the GPU-jobs from several servers into a single node, thus making a more efficient use of resources and saving energy.

Figure 7.10 presents an example of this idea. Two servers (Figures 7.10(a) and 7.10(b)) are executing, each of them, an instance of the CUDA-MEME application. The two nodes in Figure 7.10 are connected by the FDR InfiniBand network and include, each of them, an NVIDIA K20 GPU. As can be seen in Figures 7.10(a) and 7.10(b), average GPU utilization in both servers is about 40%. At time 55 seconds, the job scheduler realizes that GPU utilization in both servers is lower than the threshold (the threshold should be decided by the system administrator and could even be a composition of several parameters such as amount of jobs sharing the GPU, historical data about GPU utilization, etc). At that point in time, the job scheduler performs several checks prior to carry out the migration, such as making sure that the candidate destination GPU has enough free memory for holding both applications. Also, the job scheduler could check that aggregated GPU utilization for both applications does not exceed 100%. Once the job scheduler has carried out all the required checks, it migrates the application from the server in Figure 7.10(a) to the server in Figure 7.10(b). From that point in time, the server in Figure 7.10(b) begins executing both applications concurrently.

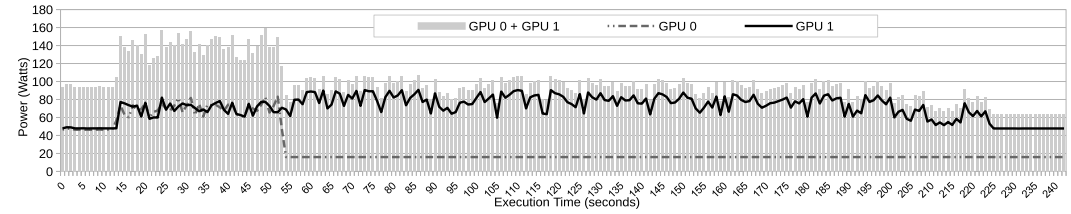
The effect of consolidating both GPU-jobs in the server can be seen in Figure 7.10(b). First, GPU utilization increases from 40% up to 60%. Theoretically, it should have increased up to 80%. However, the dynamics of applications is not so straightforward. Second, the execution of both applications is slightly lengthened. In this regard, Table 7.2 showed that a single instance of CUDA-MEME lasts for about 210 seconds. However, due to server consolidation, the concurrent execution of both applications lasts for about 240 seconds.



(a) GPU memory occupancy and GPU utilization in a server running the CUDA-MEME application. At time 55 seconds the GPU job is migrated to another server, shown in Figure 7.10(b), and thus the GPU is emptied.



(b) Server running the CUDA-MEME application. At time 55 seconds the migrated application from Figure 7.10(a) enters the GPU in this server. From that moment, the GPU in this server executes both applications concurrently.



(c) Power consumption of the two GPUs involved in the consolidation process. "GPU 0" is the source GPU whereas "GPU 1" is the destination GPU of the migration. Additionally, "GPU 1" is the GPU where both GPU-jobs are consolidated.

FIGURE 7.10: GPU migration used to consolidate servers. Two instances of the CUDA-MEME application are being executed in two servers and, at some point in time, the job scheduler decides to migrate one of the jobs to the other server. The emptied server can be later switched off if required.

An alternative point of view about this consolidation process is presented in Figure 7.10(c). This figure depicts the power consumed by each of the two GPUs during the execution of the applications. It can be seen in the figure that until second 55 both GPUs are active and consume between 60 and 80 watts. Gray bars display the aggregated power consumption of both GPUs, which can reach up to 140 watts. At second 55 migration occurs. At that point in time, the power consumption of GPU 1 (the server that receives the migrated job) slightly increases whereas the power consumption of the GPU sourcing the migration is noticeably reduced because it remains idle. The net result is a clear reduction in the power required to execute both applications, as can be seen by the gray bars in the figure. Furthermore, notice that Figure 7.10(c) only

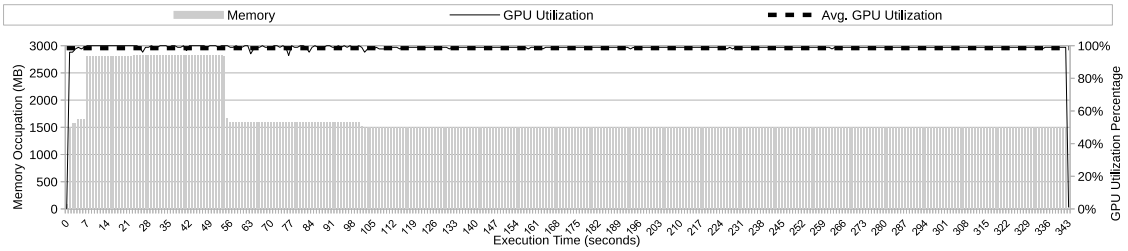
depicts power consumed by the GPUs. If we take into account the rest of the system, one can easily understand the large benefits that could be achieved if the node sourcing the migration is completely switched off.

7.5.3.2 GPU Load Balancing

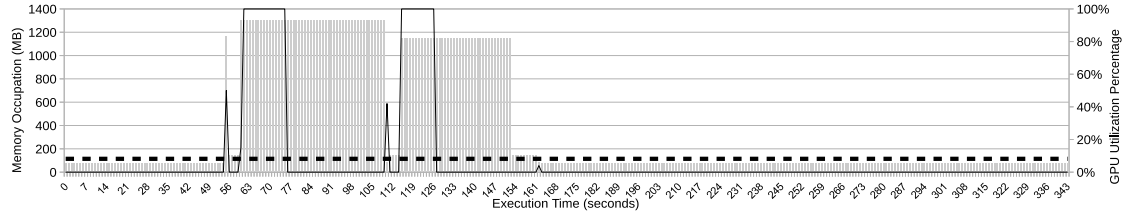
As part of the natural evolution of the workload in a data center, it could happen that, at a given point in time, a GPU is noticeably overloaded whereas other GPUs in the cluster remain idle. This situation could be desirable if the policy in the data center is to consolidate servers as much as possible, as it was reviewed in the previous section. However, other policies are feasible. For instance, the system administrator could decide to balance load among servers as much as possible in order to provide customers with execution times as low as possible. Contrary to consolidation, load balancing may not save energy. But customers might be more satisfied.

With rCUDA it is possible to balance the load of the GPUs in the cluster thanks to its migration mechanism. Actually, when several applications share a given GPU in the cluster, the migration implementation carried out within the rCUDA middleware allows to migrate each of the GPU jobs of these applications to different destination GPUs. That is, it is not required that GPUs are migrated as a whole but individual GPU jobs can be managed independently from each other. This individual migration of GPU jobs allows that load is balanced across the GPUs in the cluster.

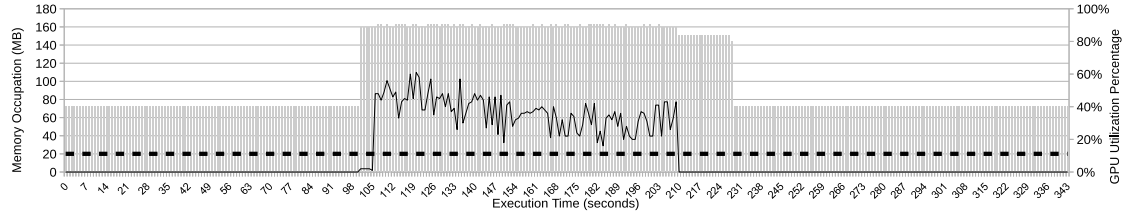
In this section we present an example of load balancing with rCUDA. Figure 7.11(a) shows a K20 GPU that is concurrently shared by three applications: CloverLeaf, GPUBLAST and CUDA-MEME. Sharing the GPU among these three applications means that all of them are executed slower than if they were executed in different GPUs. Let us assume that the policy in the cluster is to provide execution times as low as possible. Thus, the job scheduler would decide to look for idle GPUs across the cluster in order to balance load. We can see in Figures 7.11(a) and 7.11(b) that at time 50 seconds the job scheduler has found an idle GPU and thus it has migrated the GPUBLAST application to that GPU. Now in the original GPU there are only two applications: CloverLeaf and CUDA-MEME. Some time later, at second 100, a GPU in the cluster becomes idle and



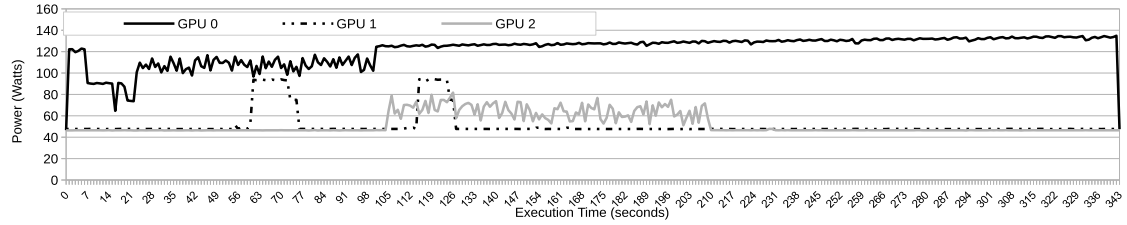
(a) "GPU 0" is concurrently executing three applications: CloverLeaf, GPUBLAST and CUDA-MEME. At time 50 seconds the GPUBLAST application is migrated to GPU 1 (Figure 7.11(b)). At time 100 seconds the CUDA-MEME application is migrated to GPU 2 (Figure 7.11(c)). The CloverLeaf application completes execution in this GPU.



(b) "GPU 1" receives the GPUBLAST application after migration at time 50 seconds.



(c) "GPU 2" receives the CUDA-MEME application after migration at time 100 seconds.



(d) Evolution of the power consumption of the three GPUs involved in the load balancing example.

FIGURE 7.11: Example of applying the GPU-job migration mechanism within rCUDA in order to balance the load among GPUs in the cluster.

thus the job scheduler decides to migrate the GPU part of the CUDA-MEME application to that GPU. This can be seen in Figures 7.11(a) and 7.11(c). The overall result is that the load of the three GPUs has been balanced and the execution of the three applications has been perfectly adapted to the resources available at every moment. In this way, application execution time has been reduced as much as the circumstances allowed. Notice that in Figures 7.11(b) and 7.11(c) there is a memory occupancy of

70 MB by default. This memory occupancy is due to the CUDA context of the rCUDA daemon.

Figure 7.11(d) presents a similar point of view of the previous process although from a power consumption perspective. The evolution of the power consumption of the three GPUs is presented. "GPU 0" refers to the initial GPU shared among the three applications. "GPU 1" refers to the GPU where the GPUBLAST application is migrated to. Finally, "GPU 2" refers to the GPU that receives the CUDA-MEME application. It can be seen in this figure that "GPU 1" and "GPU 2" remain idle until they receive the GPUBLAST and CUDA-MEME applications, respectively.

7.5.3.3 Improved Management of User Priorities

In the context of a cluster where a job scheduler deals with users having different priorities, a way to provide better service to higher priority users is to assign them the best GPUs in the cluster. However, due to the evolution of the cluster workload, it may be possible that by the time that a job from a high priority user must be placed into execution, all the powerful GPUs are already assigned to other high priority users. As a consequence, the job that is to be executed must finally use a regular GPU.

Once the job from that high priority user has entered execution in a regular GPU, it may eventually happen that some of the best GPUs become idle because they complete the execution of their jobs. At that moment, it could be possible to migrate the job that was in execution in a regular GPU so that it continues execution in one of the powerful GPUs in the cluster. This migration would satisfy the priority criteria of the cluster whereas execution time of that job would be reduced because of the better GPU.

The opposite scenario could also be possible. That is, an application is being executed in a powerful GPU but, during its execution, a higher priority user submits a job to the scheduler queues. As a consequence, the job scheduler looks for a suitable GPU to execute the higher priority job. However, it discovers that all the powerful GPUs are already in use. In this context, the job from the lower priority user can be moved out from the powerful GPU to a regular device in order to complete its execution. After moving the job out from the powerful GPU, the high priority job would enter the

TABLE 7.3: Execution time of the CloverLeaf application in different GPUs.

Application	Execution Time (s)		Execution Time (s)	
	K20	P100	K20 to P100	P100 to K20
CloverLeaf	271	80	150	227

powerful GPU and start execution. The net result would be that the priority policy in the cluster is fulfilled at the cost of slowing down the lower priority job.

Table 7.3 presents the execution time of the CloverLeaf application in the previous scenarios. First, Table 7.3 shows that this application requires 271 seconds to be executed in a K20 GPU whereas it needs 80 seconds to be completed in a P100 GPU. On the other hand, when this application is migrated from a K20 GPU to a P100 one after 33% of its execution time, we obtain a total execution time of 150 seconds. In the opposite scenario (from P100 to K20) we obtain 227 seconds.

7.6 Conclusions

This paper has presented a thorough performance analysis of the migration support implemented within the rCUDA remote GPU virtualization middleware. Although providing this kind of support within GPU virtualization frameworks is not novel, the implementation carried out for the rCUDA middleware presents a better overall architecture, which is carefully devised to be integrated with job schedulers at different levels, as it has been widely shown in the performance evaluation section. In this regard, contrary to the rest of implementations of the GPU migration mechanism in other GPU virtualization frameworks, in the rCUDA implementation it is the job scheduler the one that triggers the migration process as well as the one that selects the destination GPU, according to the scheduling and energy efficiency policies decided by the system administrator. Additionally, the GPU migration implementation presented in this paper is the only one existing for GPU virtualization solutions supporting modern CUDA versions.

Performance results show that migration is feasible and its overhead is very low when the InfiniBand network is used in the cluster. Similar extraordinary performance results are expected for other network fabrics that also provide RDMA capabilities, such as the

RoCE interconnect. Furthermore, the use cases shown in this paper clearly demonstrate that GPU-job migration is a powerful tool that can be used by the job scheduler in order to optimize the execution of accelerated applications in a cluster. In this regard, it is noteworthy that GPU-job migration provides job schedulers with an increased freedom degree when they carry out the scheduling process of accelerated applications. The reason is that, thanks to the GPU-job migration mechanism, job schedulers do not have to know, during the scheduling process, the exact amount of GPU memory used by the application being scheduled. In this way, job schedulers can assign GPUs to applications regardless of their GPU-memory footprint and, if they later experience GPU memory allocation problems due to lack of memory, the GPU jobs can be migrated to another GPU presenting more available memory. Furthermore, it could even be possible to store the GPU job in main memory in case no GPU is found with enough memory. This would stall the accelerated application until the required memory is available.

Acknowledgments

This work was funded by the Generalitat Valenciana under Grant PROMETEO/2017/77. Authors are grateful for the generous support provided by Mellanox Technologies Inc.

References

[1] A. A. Semnanian, J. Pham, B. Englert, and X. Wu. Virtualization technology and its impact on computer hardware architecture. In *2011 Eighth International Conference on Information Technology: New Generations*, 2011.

[2] Mellanox. Connectx-3 vpi single and dual qsfp+ port adapter card user manual. `http://www.mellanox.com/`, 2013. Accessed 27 September 2018.

[3] Intel ethernet server adapter i350. `http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-i350-server-adapter-brief.html`. Accessed 27 September 2018.

[4] Nvidia grid accelerated virtual desktops and apps. `http://images.nvidia.com/content/grid/pdf/188270-NVIDIA-GRID-Datasheet-NV-US-FNL-Web.pdf`. Accessed 27 Sept 2018.

[5] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. Local and remote gpus perform similar with edr 100g infiniband. In *Proceedings of the Industrial Track of the 16th International Middleware Conference*, 2015.

[6] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par 2010 - Parallel Processing*, 2010.

[7] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi. Ds-cuda: A middleware to use many gpus in the cloud environment. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012.

[8] bitfusion. The elastic ai infrastructure for multi-cloud. `https://bitfusion.io/`, 2019. Accessed 27 March 2019.

[9] J. Prades and F. Silla. Turning gpus into floating devices over the cluster: the beauty of gpu migration. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, 2017.

[10] Carlos Reaño and F. Silla. A performance comparison of cuda remote gpu virtualization frameworks. In *2015 IEEE International Conference on Cluster Computing*, 2015.

[11] C. Reaño and F. Silla. On the support of inter-node p2p gpu memory copies in rcuda. *Journal of Parallel and Distributed Computing*, 2019.

[12] Federico Silla, Sergio Iserte, Carlos Reaño, and Javier Prades. On the benefits of the remote GPU virtualization mechanism: the rCUDA case. *Concurrency and Computation: Practice and Experience*, 2017.

[13] Javier Prades, Blesson Varghese, Carlos Reaño, and Federico Silla. Multi-tenant virtual gpus for optimising performance of a financial risk application. *Journal of Parallel and Distributed Computing*, 2017.

[14] S. Xiao, P. Balaji, J. Dinan, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng. Transparent accelerator migration in a virtualized gpu environment. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012.

[15] Jiacheng Ma, Xiao Zheng, Yaozu Dong, Wentai Li, Zhengwei Qi, Bingsheng He, and Haibing Guan. gmig: Efficient gpu live migration optimized by software dirty page for full virtualization. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018.

[16] T. Suzuki, A. Nukada, and S. Matsuoka. Transparent checkpoint and restart technology for cuda applications. In *GPU Technology Conference (GTC)*, 20156.

[17] Lin Shi, Hao Chen, and Ting Li. Hybrid cpu/gpu checkpoint for gpu-based heterogeneous systems. In Kenli Li, Zheng Xiao, Yan Wang, Jiayi Du, and Keqin Li, editors, *Parallel Computational Fluid Dynamics*, 2014.

[18] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman. Crum: Checkpoint-restart support for cuda's unified memory. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[19] ZiZhuo Zhang, Xinhao Xu, Mochi Xue, Jiajun Wang, Zhengwei Qi, and Yaozu Dong. gha: An efficient and iterative checkpointing mechanism for virtualized gpus. In *APSys*, 2016.

[20] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 2010.

[21] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. Cudasw++ 3.0: accelerating smith-waterman protein database search by coupling cpu and gpu simd instructions. *BMC Bioinformatics*, 2013.

[22] M. Martineau and S. McIntosh-Smith. The arch project: physics mini-apps for algorithmic exploration and evaluating programming environments on hpc architectures. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.

[23] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne Gaudin. An evaluation of emerging many-core parallel programming models. In *Proceedings*

of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, 2016.

[24] Yongchao Liu, Bertil Schmidt, Weiguo Liu, and Douglas L. Maskell. Cudameme: Accelerating motif discovery in biological sequences using cuda-enabled graphics processing units. *Pattern Recognition Letters*, 2010.

Chapter 8

Conclusions

Abstract

In this last chapter of the thesis, the main contributions and conclusions of this work are summarized. In addition, the main lines for future work that we have planned are described. Notice that some of this future work is currently under development although it has not been included in this manuscript. Finally, all the publications that have emerged from this PhD thesis are also listed. First, the publications that have been presented in this compendium are listed together with others strictly related to the work developed in this dissertation. Finally, different collaborations around rCUDA, made during the development of the thesis, are enumerated. Notice that these collaborations are not as related to the goals of the PhD thesis as the previous ones.

8.1 Contributions

Several important contributions have been made in this PhD thesis. A first contribution of this thesis has been the analysis of GPU utilization in traditional computing clusters, either in typical Cloud Computing environments[1], where the use of virtual machines predominates and where throughput is essential or in HPC environments where we pursue maximum performance, that is, we seek the shortest possible run time for applications. Notice that it was already known that, in general, GPUs achieve low utilization. In this regard, our contribution is not disruptive. However, we have gone one step further by analyzing how GPU utilization evolves over execution time. This analysis has been carried out for different applications showing different average GPU utilization. As expected, and shown in Chapters 3, 4 or 7, the average utilization of the GPUs in a cluster is quite low. GPUs are underutilized and this is a problem as these devices are extremely expensive and highly energy efficient. Consequently, if its usage is low we will be losing money. On the one hand we will have a slower amortization of the GPUs, since the amortization speed of the computing resources depends largely on the use we make of them and, on the other hand, we will be using the GPUs for small periods of time. Notice that, in general, the GPUs are very energy efficient devices, and therefore, using them for small periods of time will surely have an impact on higher energy consumption per work. To solve all these problems, in this thesis we focus on the use of remote GPU virtualization through the rCUDA middleware and the application of several techniques to increase the average utilization of GPUs. In this regard, one important outcome of our analysis, thanks to knowing how GPU utilization evolves over execution time, is being able to efficiently share GPUs among several applications.

The second of the contributions made in this PhD thesis is provided in Chapters 2 and 3, where the feasibility of using rCUDA as a GPU virtualization tool in Cloud Computing environments is experimentally demonstrated, using different configurations: one or more GPUs, using local or remote physical GPUs, and finally, using different interconnection networks. Moreover, in Chapter 3, the feasibility of concurrently sharing physical GPUs among several virtual machines is also demonstrated and an extensive analysis of the impact this sharing has on throughput and the energy consumed by

[1] In this thesis we have evaluated the use of rCUDA in Cloud Computing under the virtualization solutions Xen [1] and KVM [2] but in the same way we can use rCUDA under any other virtualization environments.

the system under different workloads is carried out. Results show that throughput is increased up to 2x respect to traditional configurations whereas energy consumption is reduced by up to 15%. In the different analyses carried out in Chapter 3 it is also shown that, thanks to rCUDA, the virtualization of the GPUs is not dependent on GPU architecture and that the memory of the physical GPU can be divided so that we can get virtual instances with an amount of memory fully tailored to the needs of each of the virtual machines[2].

The third contribution of this PhD thesis can be found in Chapters 4 and 5, where we applied remote GPU virtualization to HPC. In these Chapters we exploited the concept of multi-tenancy, since rCUDA allows to generate several virtual instances from the same physical GPU. These virtual instances generate the illusion to applications that each of these virtual instances corresponds to a regular physical GPU. In Chapter 4 we generated virtual instances to provide additional GPUs to a single financial risk application. The experimental results show that both the execution time and the energy consumed are reduced. This is due to the fact that data transfers to/from GPU and computation are overlapped. In addition, we also generated a model that predicts both performance and energy consumed as the number of virtual instances per physical GPU is varied. In Chapter 5 we generated virtual instances of a physical GPU to provide them to different simulations of molecular dynamics, so that each simulation uses a different virtual instance of the GPU. That is, in the real GPU, different concurrent simulations are computed. At the end of Chapter 5, a model is made that predicts the sum of the benefits obtained by the entire set of molecular dynamics simulations executed in a computational cluster based on the number of nodes and the number of GPUs per node. The model predicts that rCUDA-based configurations obtain an increase in performance of around 21% with a similar energy consumption.

The last of the contributions made in this PhD thesis can be found in Chapters 6 and 7, where the GPU-job migration mechanism developed in this thesis is presented and evaluated. This mechanism allows to migrate the GPU part of the running jobs between different GPUs in the cluster, regardless of the location of these GPUs. This

[2]NVIDIA has developed the NVIDIA Virtual GPU technology, vGPU [3], that allows the NVIDIA GPUs to be concurrently shared between different virtual machines to perform CUDA computation. However, unlike rCUDA, that allows any memory partition on any GPU model, NVIDIA provides a limited and rigid memory partition options, and only specific GPU architectures and GPU models are supported by the vGPU technology.

mechanism has been designed to offer more versatility when scheduling jobs, in return we may experience a slight overhead in the execution time of the application, around 400 milliseconds in our tests. It is integrated within the rCUDA framework but it has been designed so that an external agent, the job scheduler, can trigger the entire migration process. With this mechanism, we are able to enhance the capabilities of the job scheduler either to balance the load on the cluster's GPUs, consolidate all the jobs in a few GPUs, provide different quality of service levels, etc.

8.2 Future Work

The research and experimentation carried out in this thesis shows that using remote GPU virtualization allows to greatly increase the flexibility in the usage of computing GPUs available in current clusters. In this way, as it has also been demonstrated, this larger degree of flexibility, used intelligently, provides great improvements in performance, throughput, energy consumption, etc. In addition, thanks to the GPU-job migration mechanism, we further increased the flexibility of the system and this, again, results in boosting the enhancements mentioned above. Finally, all these improvements will have an economic impact that should be quantified in order to make rCUDA attractive enough to companies, and in this way, being able to transfer this knowledge to industry and end up having a positive impact on society. Thus, we believe that the next steps in this research should consist of (i) applying all the knowledge acquired in this thesis in an automated way and (ii) quantifying the economic improvement that the use of this technology provides. In the next sections, these two directions are further elaborated.

8.2.1 GPU-Job Scheduler

In order to automate the application of the techniques developed in this thesis, we are implementing a GPU-job scheduler[3] that should obtain improvements similar to those we obtained in the works developed in this thesis although in a totally autonomous way. In general terms, we want this GPU-job scheduler to have the following characteristics:

- Must use virtualized GPU instances by leveraging the rCUDA middleware.

[3]The development of this GPU-job scheduler starts from the simple scheduler developed in Chapter 3.

- In order to apply the knowledge acquired in this thesis, the scheduler must monitor the status of the GPUs of the system in real time, and act accordingly in order to obtain the maximum benefits according to a pre-established policy. The policies used can be based on maximizing performance, reducing energy consumption, increasing throughput, etc.
- Management of the GPU memory used by the different applications that concurrently share a physical GPU.
- Must be able to take advantage of the new functionality provided by the GPU-job migration mechanism.
- Must be able to interact and work together with other more general purpose schedulers, such as Slurm [4] or OpenPBS [5].

8.2.2 Quantification of the Economic Impact of Applying the Mechanisms Developed in this Thesis

Technology transfer has been a strong concern to us from the beginning of the thesis since, from our point of view, it is very important that the improvements developed in this work are not kept in a drawer, but they can have a positive impact to society. To that end, it is essential that industry acquires and uses our technology. However, getting the industry to acquire a new technology is not always easy (even when this technology vastly improves on current technology). Therefore, once we have fully developed the GPU job scheduler, it will be essential to demonstrate that its usage has a positive impact from the economic point of view. That is, it allows to reduce costs compared to using current technology, thus having a solid argument that allows us to convince the industrial sector to adopt the new technology.

8.3 Publications

8.3.1 Main Publications

The publications related to this PhD thesis are listed below. All were submitted and accepted for publication in different international peer-reviewed journals, conferences and book chapters.

Journals:

- Javier Prades, Blesson Varghese, Carlos Reaño, Federico Silla: Multi-tenant virtual GPUs for optimising performance of a financial risk application. *Journal of Parallel and Distributed Computing.* 108: 28-44 (2017). `https://doi.org/10.1016/j.jpdc.2016.06.002`

- Javier Prades, Federico Silla: GPU-Job Migration: The rCUDA Case. *IEEE Transactions on Parallel and Distributed Systems.* 30(12): 2718-2729 (2019). `https://doi.org/10.1109/TPDS.2019.2924433`

- Javier Prades, Carlos Reaño, Federico Silla: On the effect of using rCUDA to provide CUDA acceleration to Xen virtual machines. *Cluster Computing.* 22(1): 185-204 (2019). `https://doi.org/10.1007/s10586-018-2845-0`

- Javier Prades, Baldomero Imbernón, Carlos Reaño, Jorge Peña-García, José Pedro Cerón-Carrasco, Federico Silla, Horacio Pérez Sánchez: Maximizing resource usage in multifold molecular dynamics with rCUDA. *The International Journal of High Performance Computing Applications.* 34(1) (2020). `https://doi.org/10.1177/1094342019857131`

Conferences and Workshops:

- Javier Prades, Carlos Reaño, Federico Silla: CUDA acceleration for Xen virtual machines in infiniband clusters with rCUDA. *PPoPP '16: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 35:1-35:2. `https://doi.org/10.1145/2851141.2851181`

- Javier Prades, Federico Silla: Turning GPUs into Floating Devices over the Cluster: The Beauty of GPU Migration. *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*: 129-136. https://doi.org/10.1109/ICPPW.2017.30

- Javier Prades, Federico Silla: A Live Demo for Showing the Benefits of Applying the Remote GPU Virtualization Technique to Cloud Computing. *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*: 735-738. https://doi.org/10.1109/CCGRID.2017.86

- Javier Prades, Federico Silla: Made-to-Measure GPUs on Virtual Machines with rCUDA. *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion*: 19:1-19:8. https://doi.org/10.1145/3229710.3229741

- Javier Prades, Carlos Reaño, Federico Silla, Baldomero Imbernón, Horacio Pérez Sánchez, José M. Cecilia: Increasing Molecular Dynamics Simulations Throughput by Virtualizing Remote GPUs with rCUDA. *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion*: 9:1-9:8 https://doi.org/10.1145/3229710.3229734

Book Chapters:

- Javier Prades, Fernando Campos, Carlos Reaño, Federico Silla: GPGPU as a Service: Providing GPU-Acceleration Services to Federated Cloud Systems. *Developing Interoperable and Federated Cloud Architecture.* IGI Global (2016) https://doi.org/10.4018/978-1-5225-0153-4.ch010

In addition, other related papers have been published in domestic conferences:

- Javier Prades, Carlos Reaño, Federico Silla, José Duato: Virtualización Remota de GPUs: Reduciendo el Coste del Hardware Ocioso. *Actas de las XXV Jornadas de Paralelismo*, 145-153, Spain, 2014

- Javier Prades, Carlos Reaño, Federico Silla, José Duato: Virtualización de GPUs en entornos XEN usando rCUDA. *Actas de las XXVI Jornadas de Paralelismo*, 102-110, Spain, 2015

- Javier Prades, Blesson Varghese, Carlos Reaño, Federico Silla: Reduciendo el Tiempo de Ejecución de una Aplicación de Cálculo de Riesgos Financieros a través del uso de GPUs Virtuales. *Actas de las XXVII Jornadas de Paralelismo*, 245-253, Spain, 2016

- Javier Prades, Federico Silla: Convirtiendo las GPUs en Dispositivos Flotantes sobre el Cluster: Migración de Trabajos de GPU. *Actas de las XXVIII Jornadas de Paralelismo*, 181-190, Spain, 2017

8.3.2 Main Collaborations

The result of all the collaborations made by the PhD candidate during the preparation of this thesis are listed below. These collaborations are not directly linked to this thesis, but they are fed by the knowledge acquired in it.

Journals:

- Federico Silla, Sergio Iserte, Carlos Reaño, Javier Prades: On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation Practice and Experience.* 29(13) (2017). `https://doi.org/10.1002/cpe.4072`

- Baldomero Imbernón, Javier Prades, Domingo Giménez, José M. Cecilia, Federico Silla: Enhancing large-scale docking simulation on heterogeneous systems: An MPI vs rCUDA study. *Future Generation Computer Systems.* 79: 26-37 (2018). `https://doi.org/10.1016/j.future.2017.08.050`

- Carlos Reaño, Javier Prades, Federico Silla: Analyzing the performance/power tradeoff of the rCUDA middleware for future exascale systems. *Journal of Parallel and Distributed Computing.* 132: 344-362 (2019). `https://doi.org/10.1016/j.jpdc.2019.04.021`

- Federico Silla, Javier Prades, Elvira Baydal, Carlos Reaño: Improving the performance of physics applications in atom-based clusters with rCUDA. *Journal of Parallel and Distributed Computing.* 137: 160-178 (2020). `https://doi.org/10.1016/j.jpdc.2019.11.007`

- Sergio Iserte, Javier Prades, Carlos Reaño, Federico Silla: Improving the management efficiency of GPU workloads in data centers through GPU virtualization. *Concurrency and Computation Practice and Experience* 33(2) (2021). https://doi.org/10.1002/cpe.5275

Conferences and Workshops:

- Sergio Iserte, Adrián Castelló, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, José Duato, Carlos Reaño, Javier Prades: SLURM Support for Remote GPU Virtualization: Implementation and Performance Study. *IEEE 26th International Symposium on Computer Architecture and High Performance Computing 2014*: 318-325. https://doi.org/10.1109/SBAC-PAD.2014.49

- Blesson Varghese, Javier Prades, Carlos Reaño, Federico Silla: Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application. *2015 IEEE 11th International Conference on e-Science*: 47-56. https://doi.org/10.1109/eScience.2015.15

- Federico Silla, Javier Prades, Sergio Iserte, Carlos Reaño: Remote GPU Virtualization: Is It Useful? *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*: 41-48. https://doi.org/10.1109/HIPINEB.2016.8

- Sergio Iserte, Javier Prades, Carlos Reaño, Federico Silla: Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*: 98-101. https://doi.org/10.1109/CCGrid.2016.26

- Federico Silla, Javier Prades, Carlos Reaño: Leveraging rCUDA for Enhancing Low-Power Deployments in the Physics Domain. *ICPP Workshops 2018*: 17:1-17:8. https://doi.org/10.1145/3229710.3229739

- Carlos Reaño, Javier Prades, Federico Silla: Exploring the Use of Remote GPU Virtualization in Low-Power Systems for Bioinformatics Applications. *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion*: 8:1-8:8. https://doi.org/10.1145/3229710.3229733

- Carlos Reaño, Javier Prades, Federico Silla: Improving the Efficiency of Future Exascale Systems with rCUDA. *2018 IEEE 4th International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*: 40-47. `https://doi.org/10.1109/HiPINEB.2018.00014`

Book Chapters:

- Federico Silla, Cristian Peñaranda, Javier Prades, Carlos Reaño, F. Elías Serrano, Jaime Sierra: rCUDA (remote CUDA). *Programación de GPUs usando Compute Unified Device Architecture (CUDA)*. RA-MA, S.A. Editorial y Publicaciones (2020)

Other collaborations published in domestic conferences:

- Sergio Iserte, Adrián Castelló, Antonio J. Peña, Carlos Reaño, Javier Prades, Federico Silla, Rafael Mayo, Enrique S. Quintana, José Duato: Extendiendo SLURM con Soporte para el Uso de GPUs Remotas. *Actas de las XXV Jornadas de Paralelismo*, 135-143, Spain, 2014
- Ferrán Pérez, Javier Prades, Carlos Reaño, Federico Silla, José Duato: Uso de rCUDA en Máquinas Virtuales KVM: Análisis y Prestaciones. *Actas de las XXV Jornadas de Paralelismo*, 327-334, Spain, 2014
- Sergio Iserte, Adrián Castelló, Rafael Mayo, Enrique S. Quintana, Carlos Reaño, Javier Prades, Federico Silla, José Duato: Comparativa de políticas de selección de GPUs remotas en clusters HPC. *Actas de las XXVI Jornadas de Paralelismo*, 67-72, Spain, 2015
- Rocío Alegre, Carlos Reaño, Javier Prades, Federico Silla: Uso de Rodinia y Parboil para evaluar las prestaciones de la virtualización remota de GPUs. *Actas de las XXVI Jornadas de Paralelismo*, 425-432, Spain, 2015
- Fernando Campos, Javier Prades, Carlos Reaño, Federico Silla: Uso de aceleradores CUDA en entornos cloud mediante rCUDA y KVM. *Actas de las XXVI Jornadas de Paralelismo*, 393-402, Spain, 2015

- Jaime Sierra, Javier Prades, José M. Rocher, Carlos Reaño, Federico Silla: Mejorando las prestaciones de motores de renderizado con rCUDA. *Actas de las XXVIII Jornadas de Paralelismo*, 401-406, Spain, 2017

- Jaime Sierra, Antonio Díaz-Román, Javier Prades, Carlos Reaño, Federico Silla: Análisis de prestaciones de Caffe y TensorFlow con rCUDA. *Actas de las XXVIII Jornadas de Paralelismo*, 407-412, Spain, 2017

References

[1] Xen Project. `http://www.xenproject.org/`, 2021. Accessed 10 January 2021.

[2] Kernel-based Virtual Machine, KVM. `http://www.linux-kvm.org`, 2021. Accessed 10 January 2021.

[3] NVIDIA Virtual GPU Software User Guide. `https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/index.html`, 2021. Accessed 10 January 2021.

[4] Slurm workload manager. `https://slurm.schedmd.com/documentation.html`, 2021. Accessed 10 January 2021.

[5] OpenPBS website. `https://www.openpbs.org`, 2021. Accessed 10 January 2021.