On the Enhancement of Remote GPU Virtualization in High Performance Clusters

Ph.D. Dissertation

Author: Carlos Reaño González
Advisors: Prof. Federico Silla Jiménez
Prof. José Francisco Duato Marín

June 2017

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática
de Sistemas y Computadores

On the Enhancement of Remote GPU Virtualization in High Performance Clusters

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

*Doctor of Philosophy
(Computer Engineering)*

Author

Carlos Reaño González

Advisors

*Prof. Federico Silla Jiménez
Prof. José Francisco Duato Marín*

June 2017

Doctoral Committee

- Prof. Zaid Al-Ars

  Delft University of Technology, Delft, The Netherlands.

- Prof. Ramón Doallo Biempica

  University of A Coruña, A Coruña, Spain.

- Prof. Antonio Robles Martínez

  Technical University of Valencia, Valencia, Spain.

Acknowledgements

I would like to sincerely thank all the people who helped make this thesis possible, from the beginning up to the end. If you feel you have contributed in one way or another, these lines are for you. Thank you very much indeed.

I would especially like to thank my advisors Federico Silla and José Duato for offering me the opportunity to learn, mature and develop not only my professional side, but also the personal one.

I would also like to thank my colleagues in the Parallel Architectures Group (GAP) of Universitat Politècnica de València (UPV) for providing an extremely pleasant and comfortable environment.

Last but not the least, I would like to finish by thanking my family and friends for their unconscious and selfless support.

Thank you all so much.

Contents

Abstract **xi**

Resumen **xiii**

Resum **xv**

List of Figures **xvii**

List of Tables **xxiii**

1 Introduction **1**
1.1 Preliminary Considerations and Background 2
1.2 Remote GPU Virtualization 3
1.3 Objectives and Contributions of the Thesis 6
1.4 Technology Transfer: An Industry-driven Thesis 7
1.5 Thesis Outline 9

2 Related Work **11**
2.1 Existing Remote GPU Virtualization Solutions 12
2.2 rCUDA: remote CUDA 14
2.3 Bandwidth Comparison 15
2.4 Summary 19

3 Improving the user experience of rCUDA **21**
3.1 Background 22
3.2 CU2rCU: A CUDA-to-rCUDA Converter 24
3.2.1 Need for a CUDA-to-rCUDA Converter 24
3.2.2 Implementing the `CU2rCU` Converter 27
3.2.2.1 Kernel Calls 28
3.2.2.2 Kernel Names 28
3.2.2.3 CUDA Symbols 29
3.2.2.4 Textures and Surfaces 29
3.2.3 Evaluation of the `CU2rCU` converter 30
3.3 Performance evaluation 33
3.4 Support for multithreaded applications and CUDA libraries 38
3.4.1 Support for Multithreaded Applications 38
3.4.2 Support for CUDA Libraries 43
3.5 Summary 44

4 Tuning rCUDA for InfiniBand Networks 45
4.1 Introduction . . . 46
4.2 Influence of FDR InfiniBand on the Performance of rCUDA . . . 47
4.2.1 Basic Performance Comparison of the Networks . . . 48
4.2.1.1 Testbed System . . . 48
4.2.1.2 Influence on Bandwidth . . . 49
4.2.1.3 Influence on Latency . . . 50
4.2.2 NVIDIA CUDA Samples . . . 51
4.2.3 Influence of FDR InfiniBand on Production Applications . . . 54
4.2.3.1 CUDASW++ . . . 54
4.2.3.2 GPU-BLAST . . . 56
4.2.3.3 LAMMPS . . . 58
4.2.4 Summary . . . 60
4.3 Enhancing rCUDA with Support for InfiniBand Dual-port Adapters . . . 61
4.3.1 Adding Dual-port Support to rCUDA . . . 62
4.3.2 Impact of Connect-IB on Remote GPU Usage . . . 62
4.3.2.1 From Theoretical to Real Bandwidth and Latency Figures 63
4.3.2.2 Influence on the Bandwidth of rCUDA . . . 67
4.3.2.3 Analyzing the Bandwidth of rCUDA . . . 69
4.3.3 Summary . . . 71
4.4 InfiniBand Verbs Optimizations for Remote GPU Virtualization . . . 71
4.4.1 Related Work . . . 72
4.4.2 Bandwidth Optimizations . . . 73
4.4.2.1 Number of Queue Pairs per Port . . . 74
4.4.2.2 Capacity of Send/Receive Queues . . . 76
4.4.2.3 Combining both Optimizations . . . 78
4.4.3 Experiments . . . 80
4.4.3.1 Impact of the Optimizations on rCUDA . . . 80
4.4.3.2 Impact of the Optimizations on Applications using rCUDA with Single-port Adapters . . . 82
4.4.3.3 Impact of the Optimizations on Applications using rCUDA with Dual-port Adapters . . . 84
4.4.4 Summary . . . 89
4.5 Influence of EDR InfiniBand on the Bandwidth of rCUDA . . . 90
4.5.1 Background . . . 90
4.5.2 Motivation . . . 91
4.5.3 Experiments . . . 95
4.5.3.1 Bandwidth analysis . . . 95
4.5.3.2 The Rodinia Benchmark Suite . . . 95
4.5.3.3 Production Applications . . . 99
4.5.4 Summary . . . 101
4.6 Conclusions . . . 101

5 Peer to Peer Memory Copies between Remote GPUs 103
5.1 Introduction . . . 104
5.2 Related Work on P2P Memory Copies . . . 106
5.3 Implementing Efficient P2P Memory Copies within rCUDA . . . 109

5.3.1 Version 1: Using GPUDirect RDMA 110
5.3.2 Version 2: Pre-allocating Intermediate Buffers 111
5.3.3 Version 3: Using Multiple Intermediate Buffers 114
5.3.4 Version 4: Adaptive Intermediate Buffer Size 115
5.3.5 Latency Analysis 119
5.3.6 Final Version: Hybrid Approach 121
5.4 Experiments with a Real Application 122
5.5 Conclusions 129

6 schedGPU: Fine-Grain Dynamic and Adaptive Scheduling for GPUs 131
6.1 Introduction 132
6.2 Related Work 134
6.3 GPU Scheduling Framework 136
6.4 Implementation Approaches 138
6.4.1 Client-Server 138
6.4.2 Shared Memory 139
6.4.2.1 Shared Memory Data 140
6.4.2.2 Synchronizing Access to Shared Memory 141
6.5 The Life Cycle 142
6.6 Notification Policies 145
6.7 Experimental Setup and Use-Cases 147
6.7.1 Hardware Platform 147
6.7.2 Use-cases 148
6.8 Evaluation 150
6.8.1 Overhead of the approaches 150
6.8.2 Performance Gain 151
6.8.2.1 Concurrent Execution of Individual Applications 151
6.8.2.2 Workloads Comprising Multiple Applications 156
6.8.3 Evaluation Summary 160
6.9 Conclusions 161

7 Conclusions 163
7.1 Contributions 164
7.2 Publications 164
7.3 Future Directions 171

References 173

Abstract

Graphics Processing Units (GPUs) are being adopted in many computing facilities given their extraordinary computing power, which makes it possible to accelerate many general purpose applications from different domains. However, GPUs also present several side effects, such as increased acquisition costs as well as larger space requirements. They also require more powerful energy supplies. Furthermore, GPUs still consume some amount of energy while idle and their utilization is usually low for most workloads. In a similar way to virtual machines, the use of virtual GPUs may address the aforementioned concerns. In this regard, the remote GPU virtualization mechanism allows an application being executed in a node of the cluster to transparently use the GPUs installed at other nodes. Moreover, this technique allows to share the GPUs present in the computing facility among the applications being executed in the cluster. In this way, several applications being executed in different (or the same) cluster nodes can share one or more GPUs located in other nodes of the cluster. Sharing GPUs should increase overall GPU utilization, thus reducing the negative impact of the side effects mentioned before. Reducing the total amount of GPUs installed in the cluster may also be possible.

In this dissertation we enhance one framework offering remote GPU virtualization capabilities, referred to as rCUDA, for its use in high-performance clusters. While the initial prototype version of rCUDA demonstrated its functionality, it also revealed concerns with respect to usability, performance, and support for new GPU features, which prevented its use in production environments. These issues motivated this thesis, in which all the research is primarily conducted with the aim of turning rCUDA into a production-ready solution for eventually transferring it to industry. The new version of rCUDA resulting from this work presents a reduction of up to 35% in execution time of the applications analyzed with respect to the initial version. Compared to the use of local GPUs, the overhead of this new version of rCUDA is below 5% for the applications studied when using the latest high-performance computing networks available.

Resumen

Las unidades de procesamiento gráfico (*Graphics Processing Units*, GPUs) están siendo utilizadas en muchas instalaciones de computación dada su extraordinaria capacidad de cálculo, la cual hace posible acelerar muchas aplicaciones de propósito general de diferentes dominios. Sin embargo, las GPUs también presentan algunas desventajas, como el aumento de los costos de adquisición, así como mayores requerimientos de espacio. Asimismo, también requieren un suministro de energía más potente. Además, las GPUs consumen una cierta cantidad de energía aún estando inactivas, y su utilización suele ser baja para la mayoría de las cargas de trabajo.

De manera similar a las máquinas virtuales, el uso de GPUs virtuales podría hacer frente a los inconvenientes mencionados. En este sentido, el mecanismo de virtualización remota de GPUs permite que una aplicación que se ejecuta en un nodo de un clúster utilice de forma transparente las GPUs instaladas en otros nodos de dicho clúster. Además, esta técnica permite compartir las GPUs presentes en el clúster entre las aplicaciones que se ejecutan en el mismo. De esta manera, varias aplicaciones que se ejecutan en diferentes nodos de clúster (o los mismos) pueden compartir una o más GPUs ubicadas en otros nodos del clúster. Compartir GPUs aumenta la utilización general de la GPU, reduciendo así el impacto negativo de las desventajas anteriormente mencionadas. De igual forma, este mecanismo también permite reducir la cantidad total de GPUs instaladas en el clúster.

En esta tesis mejoramos un entorno de trabajo llamado rCUDA, el cual ofrece funcionalidades de virtualización remota de GPUs para su uso en clusters de altas prestaciones. Si bien la versión inicial del prototipo de rCUDA demostró su funcionalidad, también reveló dificultades con respecto a la usabilidad, el rendimiento y el soporte para nuevas características de las GPUs, lo cual impedía su uso en entornos de producción. Estas consideraciones motivaron la presente tesis, en la que toda la investigación llevada a cabo tiene como objetivo principal convertir rCUDA en una solución lista para su uso entornos de producción, con la finalidad de transferirla eventualmente a la industria. La nueva versión de rCUDA resultante de este trabajo presenta una reducción de hasta el 35 % en el tiempo de ejecución de las aplicaciones analizadas con respecto a la versión inicial. En comparación con el uso de GPUs locales, la sobrecarga de esta nueva versión de rCUDA es inferior al 5 % para las aplicaciones estudiadas cuando se utilizan las últimas redes de computación de altas prestaciones disponibles.

Resum

Les unitats de processament gràfic (*Graphics Processing Units*, GPUs) estan sent utilitzades en moltes instal·lacions de computació donada la seva extraordinària capacitat de càlcul, la qual fa possible accelerar moltes aplicacions de propòsit general de diferents dominis. No obstant això, les GPUs també presenten alguns desavantatges, com l'augment dels costos d'adquisició, així com major requeriment d'espai. Així mateix, també requereixen un subministrament d'energia més potent. A més, les GPUs consumeixen una certa quantitat d'energia encara estant inactives, i la seua utilització sol ser baixa per a la majoria de les càrregues de treball.

D'una manera semblant a les màquines virtuals, l'ús de GPUs virtuals podria fer front als inconvenients esmentats. En aquest sentit, el mecanisme de virtualització remota de GPUs permet que una aplicació que s'executa en un node d'un clúster utilitze de forma transparent les GPUs instal·lades en altres nodes d'aquest clúster. A més, aquesta tècnica permet compartir les GPUs presents al clúster entre les aplicacions que s'executen en el mateix. D'aquesta manera, diverses aplicacions que s'executen en diferents nodes de clúster (o els mateixos) poden compartir una o més GPUs ubicades en altres nodes del clúster. Compartir GPUs augmenta la utilització general de la GPU, reduint així l'impacte negatiu dels desavantatges anteriorment esmentades. A més a més, aquest mecanisme també permet reduir la quantitat total de GPUs instal·lades al clúster.

En aquesta tesi millorem un entorn de treball anomenat rCUDA, el qual ofereix funcionalitats de virtualització remota de GPUs per al seu ús en clústers d'altes prestacions. Si bé la versió inicial del prototip de rCUDA va demostrar la seua funcionalitat, també va revelar dificultats pel que fa a la usabilitat, el rendiment i el suport per a noves característiques de les GPUs, la qual cosa impedia el seu ús en entorns de producció. Aquestes consideracions van motivar la present tesi, en què tota la investigació duta a terme té com a objectiu principal convertir rCUDA en una solució preparada per al seu ús entorns de producció, amb la finalitat de transferir-la eventualment a la indústria. La nova versió de rCUDA resultant d'aquest treball presenta una reducció de fins al 35% en el temps d'execució de les aplicacions analitzades respecte a la versió inicial. En comparació amb l'ús de GPUs locals, la sobrecàrrega d'aquesta nova versió de rCUDA és inferior al 5% per a les aplicacions estudiades quan s'utilitzen les últimes xarxes de computació d'altes prestacions disponibles.

List of Figures

1.1 Evolution of rCUDA, CUDA and InfiniBand during the implementation of this thesis. The figure shows the release date for each new version or product of those technologies relevant to this thesis. 8

2.1 Architecture usually deployed by remote GPU virtualization frameworks. 12
2.2 Overview of the rCUDA modular architecture, showing also the runtime-loadable specific communication modules. 14
2.3 Examples of possible rCUDA client-server combinations. 17
2.4 Performance comparison among three publicly available CUDA GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. The comparison is performed in terms of attained bandwidth. The performance of CUDA is also depicted for comparison purposes. 18

3.1 CUDA-to-rCUDA conversion process. 26
3.2 CUDA-to-rCUDA converter detailed view. 27
3.3 `nvcc` compilation time compared with `CU2rCU` conversion plus compilation time for CUDA SDK samples and LAMMPS. 32
3.4 Performance of the SDK samples converted in the previous section. 34
3.5 Execution time of the CL sample in CUDA compared with rCUDA using different intervals for polling the network devices. The value for the polling interval within CUDA is the default one used by the CUDA driver. Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by a QDR InfiniBand network. 36
3.6 CUDA SDK normalized execution time with the GPU previously initialized by other process (CUDA initialized) compared with rCUDA using different intervals for polling the network devices (0, 200 μs, 400 μs, 600 μs, and 1000 μs). Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by a QDR InfiniBand network. 37
3.7 Using rCUDA in different scenarios. 39
3.8 Total computation time for the MonteCarloMultiGPU sample from the NVIDIA GPU Computing SDK. All the executions operate with the same problem size (2,048 stock options). "Trend rCUDA" refers to the trend line with power regression type for rCUDA results. 40

3.9 Total execution time for a matrix-matrix multiplication using the `libflame` library. All the executions operate with the same problem size (matrix of 18 million of single-precision elements). Each GPU executes the part of the computation associated with a different thread. “Trend rCUDA” refers to the trend line with power regression type for rCUDA results. . . 41
3.10 NVIDIA profiling results for the different applications tested. In particular, time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcpy). Notice that the time depicted in the plots is the aggregation of the time used by each of the GPUs leveraged in the experiments. 42
3.11 Integration of CUDA libraries in CUDA 5. 43

4.1 Comparison between the theoretical bandwidth of different versions of PCI Express x16 and those of commercialized InfiniBand (IB) fabrics. . . 47
4.2 Bandwidth test for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over different networks. 49
4.3 Bandwidth test for copies from host to device with pageable host memory, using CUDA and the rCUDA framework over different networks. 50
4.4 Latency test varying the number of iterations for CUDA and the rCUDA framework over different networks. X-axis in logarithmic scale. 51
4.5 (a) CUDA SDK samples normalized execution time using CUDA and rCUDA over different networks. (b) rCUDA profiling measurements for CUDA SDK samples. Primary Y-axis shows MB sent/received by samples to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. Both Y-axes in logarithmic scale. 52
4.6 CUDASW++ execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA’s overhead and secondary Y-axis execution time. 55
4.7 rCUDA profiling measurements for CUDASW++ executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by CUDASW++ to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. 55
4.8 NVIDIA profiling result for CUDASW++ executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy). 55
4.9 GPU-BLAST execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA’s overhead, while secondary Y-axis represents execution time. . . 57
4.10 rCUDA profiling measurements for GPU-BLAST executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by GPU-BLAST to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. 57
4.11 NVIDIA profiling result for GPU-BLAST executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy). 57

4.12 rCUDA profiling measurements for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Primary Y-axis shows MB sent/received by LAMMPS to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. . . . 59
4.13 NVIDIA profiling result for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcopy) is shown. . . . 60
4.14 Scheme of the use of dual-port network adapters. (a) CUDA requests and responses, as well as data transfers, make use of the single available network port. (b) CUDA requests and responses to/from the remote server employ a single logic channel across one of the ports of the network card. Data transfers use two different logic channels across both ports, thus doubling the attained bandwidth for bulk transfers. . . . 62
4.15 InfiniBand bandwidth test using FDR over different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). The CUDA line shows bandwidth attained when copying data from main memory to the memory of the K40 GPU, located within the same node. . . . 65
4.16 Three-stage pipeline modeling the client-server data transfer process within a remote GPU virtualization framework. . . . 66
4.17 InfiniBand RDMA write latency test using FDR over ConnectX-3 and Connect-IB with 1 port. The CUDA line shows latency for copying data from main memory to the memory of a local Tesla K40 GPU (i.e. just using the local PCIe link). . . . 67
4.18 Bandwidth test for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). . . . 68
4.19 Latency for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over InfiniBand employing ConnectX-3 and Connect-IB with 1 port. . . . 69
4.20 Analysis of rCUDA with different hardware generations. . . . 70
4.21 Presence of Ethernet and InfiniBand in the TOP500 list. . . . 72
4.22 InfiniBand Queue Pair (QP) scheme. . . . 74
4.23 InfiniBand bandwidth tests varying the number of queue pairs (QP) per port. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs over using only 1 QP. . . . 75
4.24 InfiniBand bandwidth tests varying the capacity of the send/receive queues (i.e., number of work requests that can be allocated) from 2 requests to 256. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 128 queues over using only 2 queues. Notice the logarithmic scale of the secondary Y-axis. 77
4.25 InfiniBand bandwidth tests varying the capacity of the send/receive queues from 2 requests to 128, and number of queue pairs per port from 1 QP to 2. Primary Y-axis shows the benchmark bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs and 128 queues over using only 1 QP and 2 queues. Notice that secondary Y-axis is in logarithmic scale. . . . 79

4.26 Bandwidth test for regular CUDA (using the GPU within the host executing the benchmark) and also for rCUDA (using a remote GPU). Four different versions of rCUDA are considered: the original rCUDA, rCUDA optimized by increasing the capacity of send/receive queues, rCUDA optimized by using two QPs, and rCUDA optimized combining both optimizations. 80
4.27 Performance of rCUDA before and after optimization. 81
4.28 Performance evaluation of HOOMD-Blue, MAGMA, and GROMACS. . . 83
4.29 CUDA-MEME analysis. 86
4.30 Primary y-axis shows CTA execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). Secondary y-axis presents the rCUDA improvement of using Connect-IB (with 1 and 2 ports) with respect to ConnectX-3. 86
4.31 Profiling of the CTA application. 87
4.32 Test varying the number of CUDA calls. Each CUDA call translates into one rCUDA small message (shorter than 64 bytes). Primary y-axis shows execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). Secondary y-axis shows the increased factor of rCUDA overhead with respect to CUDA. 88
4.33 Bandwidth test for copies between host memory and GPU memory using pinned and pageable host memory. H2D refers to the copies from host to the GPU, whereas D2H refers to the opposite direction. 92
4.34 Performance of the InfiniBand (IB) bandwidth tests, compared to the performance of the NVIDIA K40 GPU. 94
4.35 Bandwidth test for copies between host memory and GPU memory using CUDA and the rCUDA middleware over the EDR InfiniBand fabric. . . . 96
4.36 Execution time of several Rodinia benchmarks using CUDA and the rCUDA middleware over EDR InfiniBand. 98
4.37 Execution time of several applications using CUDA and the rCUDA middleware over the EDR InfiniBand fabric. 100

5.1 Possible scenarios when carrying out P2P memory copies with CUDA and rCUDA. 105
5.2 Scenarios with and without NVIDIA GPUDirect RDMA used with an InfiniBand network adapter. 106
5.3 Sequence diagram of rCUDA version 1 for memory copies between different remote GPUs. This version uses GPUDirect RDMA to transfer the data. 110
5.4 Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and version 1 of rCUDA are shown. 111
5.5 Sequence diagram of rCUDA version 2A for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2B in Figure 5.6. This version uses GPUDirect RDMA to transfer data, and pre-allocates intermediate buffers at initialization. 112

5.6 Sequence diagram of rCUDA version 2B for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2A in Figure 5.5. This version uses regular RDMA to transfer data and then copy it to GPU memory. Intermediate buffers are pre-allocated at initialization. 113
5.7 Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 2A and 2B of rCUDA are shown. 114
5.8 Sequence diagram of rCUDA version 3A for memory copies between different remote GPUs. Gray boxes depict differences with respect to version 3B shown in Figure 5.9. This version is similar to version 2A, but uses multiple intermediate buffers at the GPU. 115
5.9 Sequence diagram of rCUDA version 3B for memory copies between different remote GPUs. Gray boxes refer to differences with respect to version to 3A shown in Figure 5.8. This version is similar to version 2B, but uses multiple intermediate buffers at host memory. 116
5.10 Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 3A and 3B of rCUDA are shown. 117
5.11 Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 4A and 4B of rCUDA are shown. 118
5.12 Bandwidth comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter. 118
5.13 Latency obtained for transfer sizes up to 200KB when copying data between remote GPUs. Results from CUDA and different versions of rCUDA are shown. 120
5.14 Latency comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter. 121
5.15 CPU and GPU utilization of TRICO when running the largest graph. . . 123
5.16 Primary Y-axis shows TRICO application execution time using CUDA and rCUDA. Secondary Y-axis presents the speed-up of rCUDA with respect to CUDA. 124

6.1 Example execution of applications on a node with two CPUs and one GPU. In Figure 6.1(a), two applications need to access the GPU, but can only be executed sequentially using existing workload managers. Figure 6.1(b) shows the proposed approach, in which GPU memory is accounted for to maximize utilization allowing for co-scheduling of multiple applications on the same GPU. 133
6.2 Architecture of the client-server model 139
6.3 Architecture of the shared memory model 140
6.4 Initialization of a schedGPU client in the shared memory approach 142
6.5 Pre-allocation of memory by a schedGPU client in the shared memory approach 143
6.6 Post-free of memory by a schedGPU client in the shared memory approach 144
6.7 Shutdown of a schedGPU client in the shared memory approach 144
6.8 CPU and GPU utilization and GPU memory used for one execution of the applications. 149

6.9 Comparison of the stages of the schedGPU life cycle for the client-server and shared memory approaches. 151
6.10 Execution time and speed-up obtained using schedGPU when varying number of instances of the same application that are concurrently executed. 152
6.11 CPU and GPU usage when running concurrent instances of the applications using schedGPU. 154
6.12 Frequency distribution of GPU utilization when executing the application with and without schedGPU. 155
6.13 CPU and GPU usage when running a workload using schedGPU for different client notification policies. 157
6.14 Usage per CPU core when running a workload using schedGPU for different client notification policies. 158

List of Tables

2.1 Comparison of existing CUDA-based virtualization solutions at the beginning of this thesis (July 2012) and current version of rCUDA (June 2017). 13
2.2 Evolution of rCUDA showing, for each version, the release date and the main new features included in the version. Major contributions of this thesis are highlighted in bold type, as well as the dates in which Carlos Reaño joined the rCUDA Team. 16

3.1 CU2rCU Conversion Statistics. 31
3.2 Comparison of CUDA and rCUDA Compilation Phases. 32
3.3 Initialization time of the CUDA environment in the analyzed CUDA SDK Samples. 35
3.4 Overhead introduced by rCUDA in MonteCarloMultiGPU sample. 41
3.5 Overhead introduced by rCUDA in the matrix-matrix multiplication using the `libflame` library. 42

4.1 Latency test using CUDA and the rCUDA framework over different networks. 50
4.2 LAMMPS execution time for benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. 59
4.3 rCUDA overhead for the benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. 59
4.4 Summary of rCUDA overhead using different networks, related with average number of rCUDA transfers and requests to rCUDA server, for the studied applications. 61
4.5 Parameters used for CUDA-MEME application. 85

5.1 Summary of the bandwidth results reported for GPUDirect RDMA in the study "*Benchmarking GPUDirect RDMA on Modern Server Platforms*", by Davide Rossetti. 107
5.2 Summary of the different rCUDA versions implemented for supporting memory copies between remote GPUs. 119
5.3 Minimum latency achieved by the different rCUDA versions implemented for supporting memory copies between remote GPUs. 121
5.4 Graphs used in the experiments with the TRICO application. 123
5.5 Profiling of the TRICO application when running the largest graph with CUDA. 125
5.6 Time employed in the copies between GPUs by CUDA and rCUDA when running the TRICO application with the largest graph. 125

5.7 Time employed by CUDA and rCUDA in two consecutive copies of the same size between GPUs. 126
5.8 Time employed by CUDA and rCUDA in two consecutive copies of the same size between GPUs. Synchronization calls are leveraged in order to get accurate measurements. 127
5.9 Breakdown of the TRICO application when running the largest graph with CUDA and rCUDA. Stages marked with * include synchronization points. 128

6.1 Comparison of GPU utilization and GPU memory utilization when executing the use-cases. 156
6.2 Comparison of GPU utilization and GPU memory utilization when executing a workload comprising multiple applications. 159

Chapter 1

Introduction

This chapter introduces some preliminary considerations and the background required to better understand the work developed in this thesis. Additionally, the main motivation for this study is presented. The concept of Remote GPU Virtualization is also introduced and motivated, presenting and discussing some of its benefits. Once the motivation for this work is introduced, the objectives and contributions of this thesis are exposed. Finally, an outline of the rest of this dissertation is provided.

1.1 Preliminary Considerations and Background

The market for servers is one of the fastest-growing segments of the information technologies (IT) industry. This is mainly due to the current trends in production and consumption, as described below.

On the one hand, the number of Internet-enabled devices (not only personal computers and laptops, but also portable devices such as mobile phones, tablets and even gaming consoles) continues to grow dramatically. Further, new services and applications are being continuously introduced which, combined with both a reduction of Internet connection fees and a continuous increase in the available bandwidth, have led to an unprecedented growth in the number of transactions per time unit that must be serviced by Internet servers. Google is the most representative example, but there are many other companies that are experiencing very fast demand growth in the services they offer. One of the trends that is booming and that consumes more server resources is Cloud Computing. Cloud Computing allows, for instance, storing data in the cloud and accessing them from any device. It is also possible to run applications directly on servers using a web browser as interface. All these possibilities happen without the user being aware where the data is actually stored or where the applications are being run.

On the other hand, during the last years there is also an increasing trend by large corporations to outsource IT services to data centers. Typically, these services involve the virtualization of large servers, which reduces the costs by sharing hardware resources among many customers. Furthermore, there is a noticeably increment in the use of simulation applications, such as risk estimation, stock market prediction, weather forecast, seismic data search or simulations of vehicle collisions. This kind of applications requires huge computing power to obtain results within an acceptable time period. Additionally, databases used by these large corporations usually contain multimedia material, such as images or videos. This type of contents requires processing and storage capacities that are orders of magnitude larger than the ones needed only a few years ago.

As a consequence of the previous trends, the demand for servers is growing very rapidly, likewise the need to build increasingly larger and more powerful servers. The trend followed during more than one decade to increase the power of the servers has been to add more resources to the systems, such as processors, memory and storage, and then interconnecting them to obtain large computing power at low cost. Notice that this is the same approach followed in the case of supercomputers which are currently based on connecting multiple individual mainstream computing servers together. In addition, there has been also a notable technological evolution: higher number of cores in

processors, higher memory and storage capacity, higher bandwidth in the interconnection network. This has thereby fostered the emergence of new server formats much more compact for assembly in cabinets, as for example 2U, 1U or blade. This compactness not only reduces the space occupied by each server node, but also drastically reduces the number of necessary cables and the number of auxiliary components, such as disk players, power supplies or network interfaces, which are now shared by several nodes.

However, in spite of the remarkable technological evolution, there has been in general little evolution in the system architecture, which is still based on the interconnection of server nodes similar to personal computers. These interconnected server nodes are usually grouped in what is referred to as computer cluster or just cluster. Among the few changes in the architecture of such systems, it is possible to emphasize the one that has occurred at the I/O level. In this way, the bandwidth has been increased with respect to the one of personal computers achieving greater disk accessing speed.

Another remarkable architectural change is the evolution in the interconnection network between processors, with the appearance of several technologies both standard, such as InfiniBand [1], or proprietary, as for instance Cray Aries [2] or Quadrics [3]. These new technologies offer message transfer rates significantly higher than those achieved with traditional networks (i.e., Ethernet).

Finally, the most recent change introduced by server manufacturers has consisted in taking advantage of the enormous computing power of the gaming graphics processing units (GPUs). This approach consists in including in each server node one or more accelerators based on these gaming GPUs. These devices are similar to the ones used for gaming, but without graphic output, with a more robust design that allows many more hours of operation, and incorporating mechanisms for detecting and correcting GPU memory errors.

1.2 Remote GPU Virtualization

Currently, the massive parallel capabilities of GPUs are leveraged to accelerate specific parts of applications. In this regard, programmers exploit GPU resources by off-loading the computationally intensive parts of applications to them. To that end, although programmers must specify which parts of the application are executed on the CPU and which parts are off-loaded to the GPU, the existence of libraries and programming models such as CUDA (Compute Unified Device Architecture) [4] noticeably ease this task. In this context, GPUs significantly reduce the execution time of applications from

domains as different as Big Data [5], chemical physics [6], computational algebra [7], image analysis [8], finance [9], and biology [10], for instance.

Current computing facilities typically include one or more GPUs at every node of the cluster. However, using GPUs in such a configuration is not exempt from side effects. For example, let us consider the execution of a distributed MPI (Message Passing Interface) application which does not require the use of GPUs. Typically, this application will spread across several nodes of the cluster flooding the CPU cores available in them. In this scenario, the GPUs in the nodes involved in the execution of such an MPI application would become unavailable for other applications because all the CPU cores in those nodes would be devoted to the non-accelerated MPI application. This would force those GPUs to remain idle for some periods of time.

Another example of the concerns associated with the use of GPUs in clusters is related to the way that job schedulers such as Slurm [11] perform the accounting of resources in a cluster. These job schedulers use a fine granularity for resources such as CPUs or memory, but not for GPUs. For instance, job schedulers can assign CPU resources in a per-core basis, thus being able to share the CPU sockets present in a server among several applications. In the case of memory, job schedulers can also assign, in a shared approach, the memory present in a given node to the several applications that will be concurrently executed in that server. However, in the case of GPUs, job schedulers use a per-GPU granularity. In this regard, GPUs are assigned to applications in an exclusive way. Hence, a GPU cannot be shared among several applications even when it has enough resources to allow the concurrent execution of those applications, causing that overall GPU utilization is, in general, low. This fact not only reduces the effective computing power of computing facilities but also causes that a non-negligible amount of energy is wasted, being both aspects key concerns in the context of the future exascale computing.

In order to address the side effects related to the use of GPUs, the remote GPU virtualization mechanism could be used. This software mechanism allows an application being executed in a computer which does not own a GPU to transparently make use of accelerators installed in other nodes of the cluster. In other words, the remote GPU virtualization technique allows physical GPUs to be logically detached from nodes, thus allowing that decoupled (or virtual) GPUs are concurrently shared by all the nodes of the computing facility in a transparent way to applications. This not only increases overall GPU utilization but also allows to create cluster configurations where not all the nodes in the cluster own a GPU, thus reducing the costs associated with the acquisition and later use of GPUs. In this regard, the total energy required to operate a computing

facility would be decreased, thus loosening the energy concerns of current and future computing facilities.

Several are the benefits that remote GPU virtualization presents [12–14], namely:

- More GPUs are available for a single application. An application being executed in a single node can use all the GPUs in the cluster, thus boosting its performance. In this case, the only limitation is the ability of the programmer to code the application in the proper way so that it takes advantage of as many GPUs as they are available.
- Increased cluster throughput. GPUs can be concurrently shared among several applications as far as there are enough memory resources available in the GPUs for the applications being executed. Additionally, given that a GPU can be used by applications being executed in a node other than the one where the GPU is installed, when all the CPU cores in the node owning the GPU are busy with a non-accelerated application, the GPU can still be used from a remote node. These features contribute to a higher GPU utilization, what translates into an increased cluster throughput (i.e., more jobs per time unit), at the same time that energy consumption is reduced.
- Cheaper cluster upgrade. Clusters which do not include GPUs can be easily and cheaply updated for using GPUs just by attaching to them servers populated with GPUs, and using remote GPU virtualization to accelerate applications running at any of the nodes in the rest of the cluster.
- Virtual machines can easily access GPUs. Current solutions for providing GPU acceleration to virtual machines environments present some issues, such as not allowing simultaneously sharing a GPU among several virtual machines. With the remote GPU virtualization mechanism, it is possible to concurrently assign a given GPU to several virtual machines, so that the applications being executed inside them can share the GPU resources.
- Migration of GPU jobs. Migrating jobs in a cluster is an effective way of reducing overall energy consumption. In this manner, jobs running in servers with lower workload are moved to other servers in order to switch off the former computers. Migrating applications that make use of GPUs is a very complex task because it is necessary to track the GPU resources used by the application and their status. Remote GPU virtualization solutions store all this information, making it simple to migrate GPU jobs.

Notice, however, that using remote GPUs also presents some difficulties. In this regard, probably the most important one is the overhead introduced by this approach, mainly due to the virtualization framework and the network. The GPU virtualization framework increases the latency to the real GPU, as requests must be forwarded to the remote GPU and responses delivered back to the application demanding GPU services. Furthermore, as GPUs are no longer located at the other end of a PCI Express (PCIe) link within the host, but in a remote node, data have to traverse at least two PCIe links and two network interfaces, as well as the entire network fabric between the node requesting GPU services and the node where the actual GPU resides. Therefore, in addition to latency, bandwidth also suffers, given that the PCIe bandwidth is usually noticeably larger than network bandwidth, thus increasing the performance gap between the local and remote uses of GPUs.

As it will be covered more thoroughly later in Chapter 2, at the beginning of this thesis, there were several frameworks offering remote GPU virtualization capabilities. As starting point for this dissertation, we used a framework called rCUDA [15]. It is also a development of our group, the Parallel Architectures Group[1] from Universitat Politècnica de València[2] (Spain). The initial version of rCUDA developed prior to this dissertation was the result of a previous thesis [16]. As it will also be shown later in Chapter 2, rCUDA was the framework presenting the best features both in terms of performance and compatibility with applications developed for local GPUs. While this initial prototype version of rCUDA demonstrated its functionality, it also revealed concerns with respect to usability, performance, and support for new GPU features. These issues motivated this thesis, as described in the following section.

1.3 Objectives and Contributions of the Thesis

Using the rCUDA remote GPU virtualization framework as the starting point for this thesis, the main objective of this dissertation is to enhance it for its use in high-performance clusters. In particular, the major contributions of this thesis are:

- Improving the user experience of the rCUDA framework. Initial versions of the remote GPU virtualization solution used in this thesis required modifying the source code for using it. This constraint has been removed and now existing application codes can be used without any modification.

[1] `http://www.gap.upv.es`
[2] `http://www.upv.es/index-en.html`

- Tuning rCUDA for InfiniBand networks. Apart from providing for the first time a performance analysis on real scenarios with real applications and production codes, in this thesis we explore optimizations for improving the communications over one of the most popular network fabrics in high performance computing (HPC) environments: InfiniBand.

- Extending rCUDA with support for peer to peer memory copies between remote GPUs. rCUDA, the remote GPU virtualization solution aim of this thesis, has also been enhanced by allowing copies between remote GPUs located in different nodes of the cluster.

- A fine-grain dynamic and adaptative scheduling for GPUs. The knowledge acquired while doing this thesis has also been applied to develop a fine-grain intra-node GPU scheduler, dynamically blocking and releasing GPUs, and capable of adapting quickly to the changing requirements.

1.4 Technology Transfer: An Industry-driven Thesis

The link between industry and the research carried out at the university during the implementation of this dissertation has been very strong. In this manner, all the research is primarily conducted with the aim of turning rCUDA into a production-ready solution for eventually transferring it to industry. Taking this goal in mind, chapters addressing major contributions of this thesis include at their beginning a section entitled "Link with Industry", to put the chapter into a meaningful context for the reader. For better understanding these opening sections, in some of them we will also refer to Figure 1.1. This figure shows a time line highlighting each new version released by CUDA and rCUDA, and also new InfiniBand products announced that are relevant to this dissertation.

Notice that turning research into a product involves tasks beyond research, which are not commonly done in a doctoral thesis, and which require important efforts. By way of example, some of these tasks are:

- Technical support. Offering remote assistance for installing and using rCUDA. Currently, more than 750 users worldwide have requested the software.

- Periodical software releases. Keeping the rCUDA software updated to support the latest versions and features of CUDA. Since this thesis started in July 2012, twelve new versions of rCUDA have been released (see Figure 1.1). More detailed information in this regard will be shown in Chapter 2, Table 2.2.

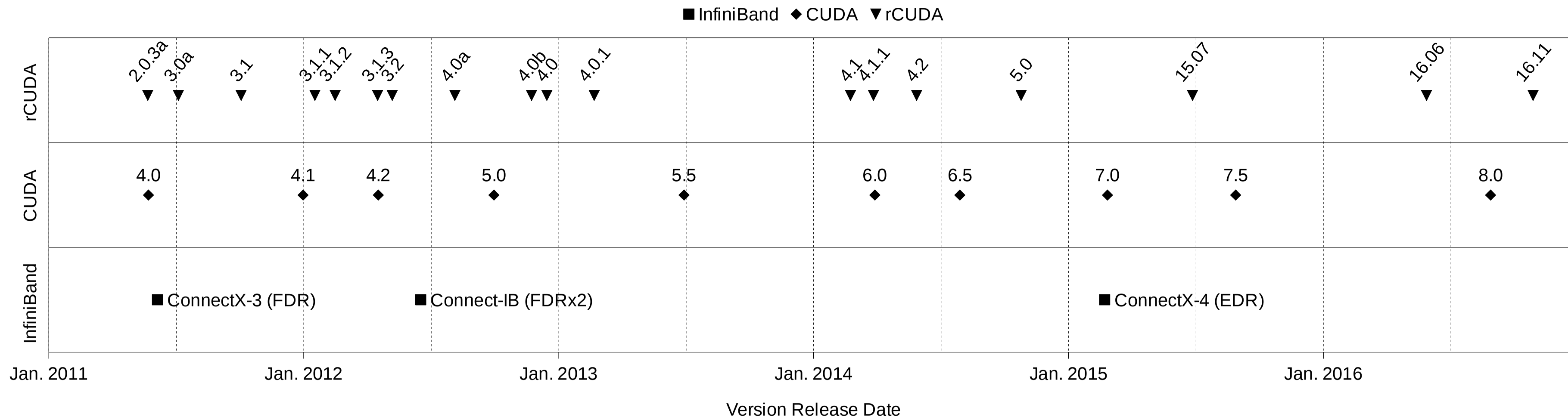


FIGURE 1.1: Evolution of rCUDA, CUDA and InfiniBand during the implementation of this thesis. The figure shows the release date for each new version or product of those technologies relevant to this thesis.

- Tutorials. Providing training on the use of rCUDA at international conferences. Four tutorials have been given, as it will be detailed in Chapter 7.
- Demonstrations at exhibitions. rCUDA has been exhibited annually in the two major events in the field, namely the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), and the International Supercomputing Conference (ISC). Refer to Chapter 7 for more details.
- Presence in social media. Keeping rCUDA website (`www.rcuda.net`) and its account in the social media Twitter (`http://twitter.com/rcuda_`) up to date.

As of this writing, the research presented in this dissertation has turned the initial prototype version of rCUDA into a commercial product, and the university has started the process to spin off this research to a company[3]. In addition, other commercial solutions providing remote GPU virtualization have also recently appeared in the market [17], which demonstrates, from an industry point of view, the research addressed in this thesis.

1.5 Thesis Outline

In order to properly address the objectives of this work, this dissertation is composed of six additional chapters. Chapter 2 introduces and compares existing remote GPU virtualization frameworks, reinforcing the motivation for using rCUDA in this thesis. Next, Chapter 3 describes how the user experience of the rCUDA framework has been improved. Afterwards, Chapter 4 details the performance analysis and optimizations carried out for InfiniBand networks. Then, Chapter 5 shows the development done to allow peer to peer memory copies between remote GPUs located in different nodes of the cluster. Later, Chapter 6 presents the work concerning the fine-grain dynamic and adaptative scheduling for GPUs. Finally, Chapter 7 summarizes this thesis, discusses future work, and enumerates the related publications.

[3] Remote Libraries (rLIBs). More information at `www.remotelibraries.com`.

Chapter 2

Related Work

Currently, many different remote GPU virtualization frameworks exist, all of them presenting very different characteristics. These differences among them may lead to differences in performance. In this chapter a review of the virtualization frameworks currently available is presented, placing particular emphasis on rCUDA. Additionally, we show a performance comparison among the only three CUDA remote GPU virtualization frameworks publicly available at no cost at the beginning of this thesis (July 2012). Results show that performance greatly depends on the exact framework used, being the rCUDA virtualization solution the one that stands out among them. The work presented in this chapter has been published in [18].

2.1 Existing Remote GPU Virtualization Solutions

The use of CUDA GPUs allows reducing the execution time of parallel applications and presents several side-effects, as explained in the previous chapter. In this context, several remote GPU virtualization frameworks have been created to overcome these drawbacks, such as VGPU [19], GridCuda [20], DS-CUDA [21], GVirtuS [22], vCUDA [23], GViM [24], rCUDA [15], and Shadowfax II [25].

Figure 2.1 depicts the architecture underlying most of these virtualization solutions, which follow a client-server distributed 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 computer owning the actual GPU. Generally, the client middleware offers the same application programming interface (API) as does the NVIDIA CUDA API [26]. 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 serviced by a remote GPU instead of a local one.

Different virtualization frameworks provide different features, as shown in Table 2.1. For example, the vCUDA technology supports the old CUDA 3.2 version and implements an unspecified subset of the CUDA API. Moreover, its communication protocol presents a considerable overhead. GViM is based on the old CUDA version 1.1 and does not implement its entire API. The gVirtuS approach is based on the old CUDA version 2.3

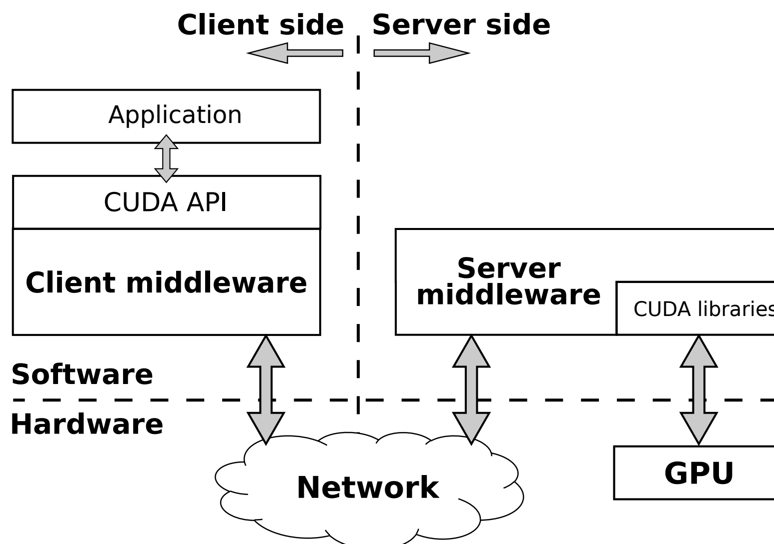


Figure 2.1: Architecture usually deployed by remote GPU virtualization frameworks.

Table 2.1: Comparison of existing CUDA-based virtualization solutions at the beginning of this thesis (July 2012) and current version of rCUDA (June 2017).

Feature (support)	Virtualization Solutions						
	vCUDA	GViM	gVirtuS	VGPU	DS-CUDA	GridCuda	rCUDA
CUDA version	3.2	1.1	2.3	4.0	4.1	3.2	8.0
Full CUDA API	no	no	no	N/A	no	N/A	yes
Latest activity year	2012	2009	2010	2012	2012	2011	2017
Multi-GPU	N/A	N/A	N/A	N/A	yes	yes	yes
Multi-thread	N/A	N/A	N/A	N/A	yes	yes	yes
TCP/IP	yes	no	yes	yes	yes	yes	yes
HPC networks	no	no	no	yes	yes	no	yes

and implements only a small portion of its API. VGPU is a tool supporting CUDA 4.0 although the information provided by its authors is fuzzy. GridCuda supports the old CUDA 2.3 version. Regarding DS-CUDA, it integrates a more recent version of CUDA (4.1) and includes specific communication support for InfiniBand. 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. In the case of rCUDA, its latest version 16.11 supports CUDA 8.0, implementing specific communication modules for different interconnects, namely, the general TCP/IP protocol stack and InfiniBand, as it will be detailed in the next section. Note that as of this writing, other solutions have appeared [17], but they are very recent ones and we were not able to include them in this study.

In the rest of this chapter we first introduce rCUDA in more detail, and then we present a performance comparison among the publicly available CUDA remote GPU virtualization frameworks: gVirtuS, DS-CUDA, and rCUDA. The performance of CUDA is also included for comparison purposes.

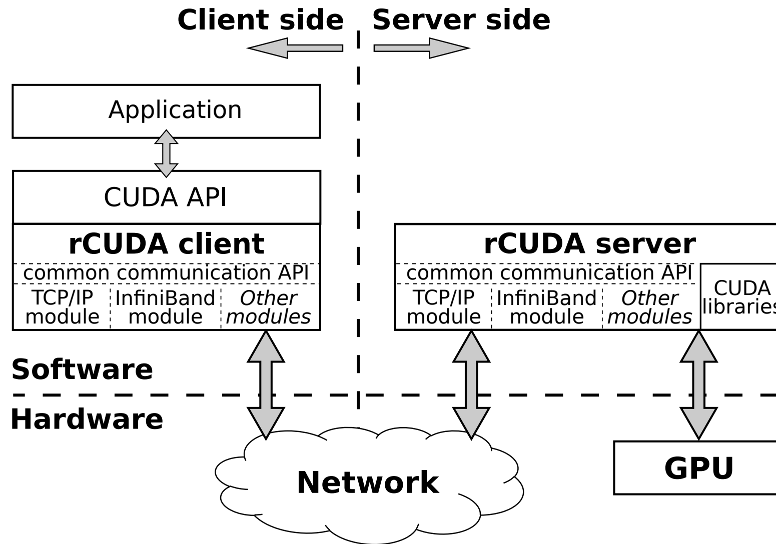


FIGURE 2.2: Overview of the rCUDA modular architecture, showing also the runtime-loadable specific communication modules.

2.2 rCUDA: remote CUDA

The rCUDA framework[1] grants applications transparent access to GPUs installed on remote nodes, so that they are not aware of the use of an external device. This framework is organized following a client-server architecture; see Figure 2.2. The client middleware is used by the application demanding general-purpose computing on GPUs (GPGPU) services and presents to the application the same interface as the regular NVIDIA CUDA API does. Upon receiving a request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server middleware, running on a remote node. The server interprets the requests and performs the required processing by instructing the real GPU to execute the corresponding request. Once the GPU has completed the execution of the requested command, the results are gathered by the rCUDA server, which sends them back to the client middleware. There, the output is forwarded to the demanding application.

The communication between rCUDA clients and remote GPU servers is carried out via a customized application-level protocol tailored for the underlying network; see [27–29]. As can be seen in Figure 2.2, it is based on a modular communication architecture which supports runtime-loadable specific communication modules. Communication between clients and servers has been improved by accommodating a pipelined approach, in order to increase effective throughput and, therefore, squeeze as much bandwidth as possible from the underlying interconnect. All this functionality, which is transparent to rCUDA users, has been made possible by a carefully designed proprietary common internal

[1] The rCUDA framework can be requested for download from `http://www.rcuda.net/index.php/software-request-form.html`

API for rCUDA communications. That API enables rCUDA clients and servers to (1) communicate through different underlying communication technologies (Ethernet, InfiniBand, etc.) and (2) to do so efficiently, since the communication functionality can be specifically implemented and tuned up for each different communication technology.

rCUDA currently supports efficient communication over Ethernet and InfiniBand and opens the door to other interesting future network technologies as well as to virtual machine environments such as Xen [30]. Furthermore, regardless of the specific communication technology used, data transfers between rCUDA clients and servers are pipelined in order to improve performance. For this purpose, rCUDA uses preallocated buffers of pinned memory, exploiting the higher throughput that this kind of storage provides. The reader can refer to [16] for a detailed analysis.

In general, the performance offered by rCUDA is lower than that of the original CUDA, since with rCUDA the GPU is farther away from the invoking application than it is with CUDA, thus introducing some overhead. Depending on the network bandwidth, however, this penalty is very low for most applications, as it will be seen in later chapters. Moreover, the performance of applications using rCUDA is still noticeably higher than that provided by computations on regular CPUs. Taking into account the flexibility provided by rCUDA, in addition to the reduction in energy and acquisition costs it enables, the benefits of rCUDA outweigh the overhead it introduces, as will be shown in later chapters. Figure 2.3 depicts different possibilities that the use of rCUDA provides, showing the flexibility it introduces.

The first version of rCUDA was released in April, 2010. Since then, the rCUDA framework has evolved to adapt to new CUDA features. Table 2.2 presents detailed information about this evolution. The table shows all the versions published as of writing. In addition, in order to better understand the context of this thesis, the table also includes the major contributions of the dissertation, highlighted in bold type, as well as the dates in which Carlos Reaño, the author of the thesis, joined the rCUDA Team[2].

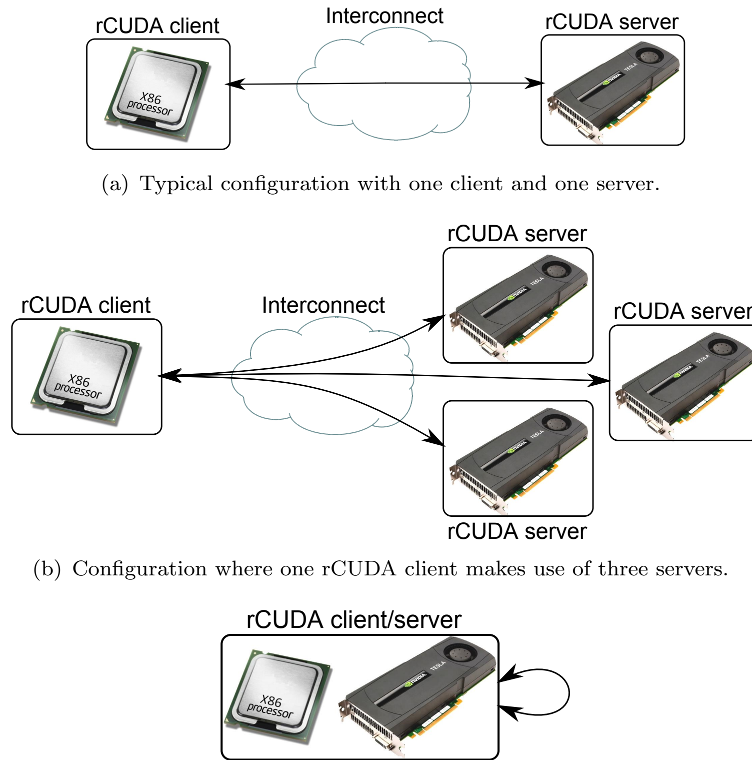
2.3 Bandwidth Comparison

Figure 2.4 presents a performance comparison of the three GPU virtualization solutions publicly available at no cost at the beginning of this thesis (July 2012), showing also the performance of CUDA as the baseline reference. The well-known `bandwidthTest`

[2]The rCUDA Team includes the several students that contribute to the rCUDA technology. The team, which is led by Federico Silla since it was created in 2008, has evolved over time trying to reach a stable configuration, which will help transferring the rCUDA framework to industry

Table 2.2: Evolution of rCUDA showing, for each version, the release date and the main new features included in the version. Major contributions of this thesis are highlighted in bold type, as well as the dates in which Carlos Reaño joined the rCUDA Team.

Version	Release date	New features
16.11	Oct. 21, 2016	Support for CUDA 8.0 **Support for peer to peer memory copies between remote GPUs**
16.06	June 10, 2016	Support for CUDA 7.5 **InfiniBand optimizations**
15.10	Oct. 3, 2015	**InfiniBand optimizations**
15.07	July 3, 2015	Support for CUDA 7.0 **Support for dual-port InfiniBand cards** **InfiniBand optimizations** Support for cuDNN
5.0	Oct. 28, 2014	Support for CUDA 6.5
4.2	May 31, 2014	Support for CUDA 6.0
4.1.1	Mar. 28, 2014	Bug fixes and enhancements
4.1	Feb. 25, 2014	Support for CUDA 5.5
4.0.1	Feb. 22, 2013	Support for CUDA mapped memory
4.0	Dec. 14, 2012	Support for CUDA 5.0
4.0b	Nov. 24, 2012	**Binary compatibility with CUDA**
4.0a	Aug. 6, 2012	Support for multi-thread applications New communication architecture
3.2	Apr. 27, 2012	Support for CUDA 4.2
3.1.3	Apr. 17, 2012	Bug fixes and enhancements
3.1.2	Feb. 16, 2012	Support for CUDA 4.1
3.1.1	Jan. 18, 2012	Support for CUBLAS (simple precision)
3.1	Oct. 4, 2011	Support for Unified Virtual Addressing
3.0a	Jul. 6, 2011	Support for CUDA 4.0 *First version including Carlos Reaño developments*
2.0.3a	May 23, 2011	Bug fixes and enhancements
Feb. 17, 2011 - Carlos Reaño joins the rCUDA team		
2.0.2	Jan. 26, 2011	Bug fixes and enhancements
2.0.1	Jan. 19, 2011	Bug fixes and enhancements
2.0	Nov. 19, 2010	Support for CUDA 3.1
1.0.3	June 2, 2010	Bug fixes and enhancements
1.0.2	May 24, 2010	Bug fixes and enhancements
1.0.1	Apr. 22, 2010	Bug fixes and enhancements
1.0	Apr. 1, 2010	Support for CUDA 3.0



(a) Typical configuration with one client and one server.

(b) Configuration where one rCUDA client makes use of three servers.

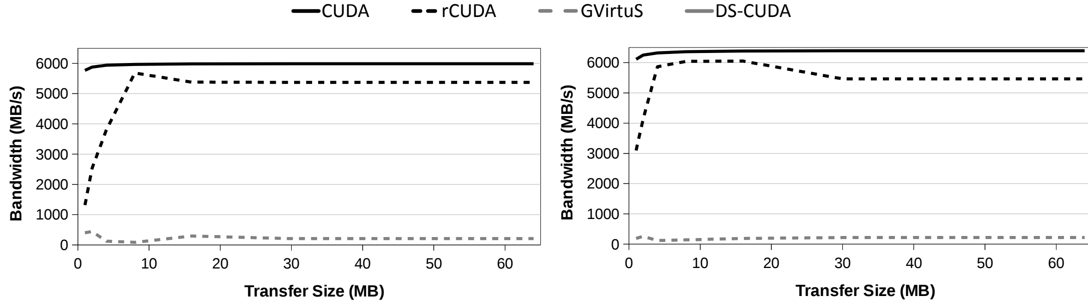
(c) A single computer acting as both client and server. The local GPU is accessed through the rCUDA middleware.

FIGURE 2.3: Examples of possible rCUDA client-server combinations.

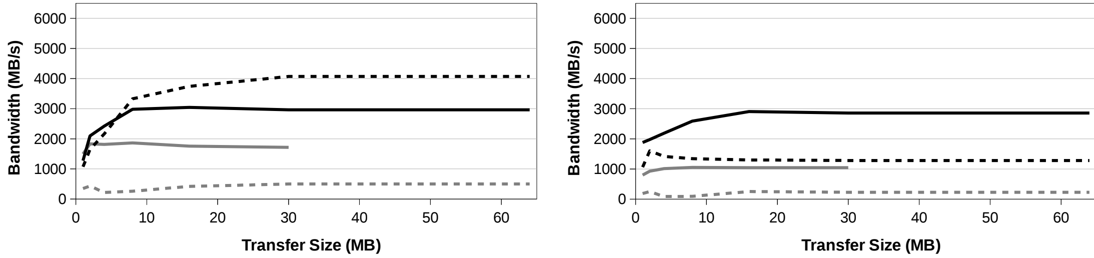
benchmark from the NVIDIA CUDA Samples [31] has been employed. The reason for using bandwidth for comparing performance is that, when transferring data in CUDA applications between main memory and GPU memory, data copy sizes are, in general, large (in the order of MB). These large data transfers are 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 used in the experiments consists of two servers featuring two Intel Xeon E5-2620 processors (Sandy Bridge) operating at 2.00 GHz and 32 GB of DDR3 memory at 1333 MHz. The computers also include an FDR InfiniBand adapter. One of the nodes owns an NVIDIA Tesla K20c GPU. Linux CentOS 6.3 was used along with NVIDIA driver 285.05 and Mellanox OFED 1.5.3 (InfiniBand drivers and administrative tools). Given that the FDR InfiniBand network technology was used to connect both computers, both rCUDA and DS-CUDA made use of the InfiniBand Verbs API. In the case of gVirtuS, TCP/IP over InfiniBand was used because it is not able to take advantage of the InfiniBand Verbs API.

Notice that the three GPU virtualization solutions analyzed support different versions



(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 2.4: Performance comparison among three publicly available CUDA GPU virtualization solutions: gVirtuS, DS-CUDA, and rCUDA. The comparison is performed in terms of attained bandwidth. The performance of CUDA is also depicted for comparison purposes.

of CUDA: DS-CUDA is compatible with CUDA 4.1, gVirtuS supports the old CUDA 2.3 version and rCUDA supports CUDA 4.2 at the time of the experiments (July 2012). 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 and the result is that differences in performance for the bandwidth test are negligible from one CUDA version to another.

Regarding the `bandwidthTest` benchmark used in the experiments, it allows different memory copy tests depending on the memory kind used and the copy direction. In this way, it is possible to differentiate between main memory, also referred to as host memory, and GPU memory, also referred to as device memory. With respect to the host memory, it can be pageable memory or page-locked memory. The later is also referred to as pinned memory, and offers a higher bandwidth than the pageable one.

Results in Figure 2.4 deserve some discussion. First, it can be seen in Figures 2.4(a) and 2.4(b) that CUDA achieves the highest performance when pinned memory is used, attaining a bandwidth around 6,000 MB/s. Notice that this bandwidth is noticeably reduced for copies using pageable memory, as show in Figures 2.4(c) and 2.4(d). Second, Figure 2.4 shows that rCUDA outperforms the other two remote

GPU virtualization solutions. Actually, for copies from host pageable memory to device memory, Figure 2.4(c), using pageable memory rCUDA also performs better than CUDA. This is a well-known effect thoroughly described in previous works on rCUDA [15] 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, DS-CUDA presents a very poor performance. Also, notice that DS-CUDA supports neither memory copies larger than 32MB nor the use of pinned memory. Regarding gVirtuS, its performance 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, TCP over InfiniBand FDR provides around 1,190 MB/s, which is a noticeably larger bandwidth than the one attained by gVirtuS. Hence, the low performance of this middleware is not due to the use of TCP/IP over InfiniBand but to the way it internally manages communications.

2.4 Summary

In this chapter we have presented and compared the performance of the publicly available CUDA remote GPU virtualization frameworks. Furthermore, their throughput has been put into the context of the performance of CUDA. Results show that the rCUDA framework outperforms the other two remote GPU virtualization solutions. Given that rCUDA seems to be the current best solution, it has been selected as the starting point for this thesis, in order to enhance it for its use in high-performance clusters.

Chapter 3

Improving the user experience of rCUDA

While the initial prototype versions of rCUDA developed prior to this thesis already demonstrated its functionality, they also revealed concerns with respect to usability and support for new CUDA features. In response to these issues, in this chapter we present the first of the rCUDA versions developed along this thesis. This version (1) improves usability by including a new component that allows an automatic transformation of any CUDA source code so that it conforms to the needs of the rCUDA framework, and (2) supports multithreaded applications and CUDA libraries. As a result, rCUDA allowed the use of remote GPUs within a cluster, for any CUDA-compatible program and in a transparent way to the user. The work discussed in this chapter has been published in [32, 33].

Link with Industry

The initial prototype of rCUDA forced the source code of applications to meet very strict requirements thus making rCUDA uncomfortable to use. In this way, the first step for turning rCUDA into a production-ready solution was making it binary compatible with CUDA, without requiring modification to the source code of applications.

This chapter makes a major contribution in this sense. It presents a CUDA-to-rCUDA converter developed to automatically analyze the application source code in order to find which parts must be modified and adapted to the requirements of rCUDA without human intervention, thus making the use of rCUDA much more comfortable.

Nevertheless, the research and analysis carried out for the development of the converter presented in this chapter finally derived, in subsequent versions of rCUDA, in the binary compatibility of any CUDA application. This binary compatibility means that applications can be executed with rCUDA without being necessary to modify their source code (no conversion needed). In the following chapters, we will be using versions of rCUDA including this binary compatibility.

3.1 Background

The rCUDA remote GPU virtualization framework grants access to GPUs installed in remote nodes to CUDA-based applications. In this manner, applications remain unaware that they deal with virtualized GPUs instead of real ones. The initial rCUDA prototype [27–29] focused on demonstrating that the exploitation of remote CUDA devices is feasible but revealed the following drawbacks:

- The usability of the rCUDA framework was limited by its lack of support for the CUDA C extensions. As shown in Section 3.2, the reason is the use of the CUDA runtime library, which includes several hidden and undocumented functions within these extensions. Therefore, in order to avoid the use of these undocumented functions, our framework offered support only for the plain CUDA C API, thus making it necessary to rewrite those lines of the application source files that employ the CUDA C extensions. For applications comprising large amounts of source code, performing this process manually was cumbersome.
- Our rCUDA virtualization solution had to evolve to support the new versions of CUDA being periodically released. In this regard, the initial rCUDA prototype presented in [27–29] supported the now-obsolete CUDA 2 and 3 versions. After

those initial versions of rCUDA, NVIDIA introduced support for multithreaded applications in CUDA 4 and later a new way to internally manage CUDA libraries in CUDA 5, resulting in significant changes with respect to prior versions. Subsequent versions of CUDA also introduced additional changes.

Because of these drawbacks, the user's experience with the rCUDA remote GPU virtualization framework was far from ideal, where users can seamlessly access remote GPUs without noticing any significant performance loss or having to adapt their application codes to the requirements of the remote GPU virtualization framework. Thus, when using the rCUDA prototype presented in [27–29] (rCUDA v3), the user's experience was relatively unsatisfactory. rCUDA users suffered from a hard limitation about which CUDA programs they were able to remotely execute with rCUDA, given that the CUDA C extensions were not supported by the rCUDA framework. Additionally, the new features introduced in versions 4 and 5 of CUDA could not be leveraged. The result was that rCUDA users had to manually modify the source code of their programs, which was hardly acceptable.

In this chapter we present how we enriched rCUDA (creating rCUDA v4) by addressing the concerns above, thus narrowing the distance between the actual user's experience and the ideal case. In this way, we improved rCUDA with the following additions:

- A complementary tool, `CU2rCU`, to automatically analyze and modify the application source code in order to find which parts, containing CUDA C extensions, must be modified and adapted to the requirements of rCUDA. This tool automatically performs the required changes, without the manual intervention of a programmer. Moreover, the `CU2rCU` tool has been integrated into the compilation flow, so that rCUDA users can effectively replace the call to NVIDIA's `nvcc` compiler with the `CU2rCU` command, which will internally make use of the required backend compilers after analyzing and adapting the source code files.
- Support for multithreaded applications and for all CUDA libraries, thus allowing users to leverage all the new features included in CUDA versions 4 and 5. Additionally, the new version of rCUDA is now able to offer a single application, being executed in a single node, access to GPUs in many different nodes of the cluster (multinode configuration). This is an important improvement over CUDA, where the number of GPUs provided to an application running in a given node is limited to the GPUs that can be attached to that single node (typically up to 4 and usually never more than 8). In the new version, rCUDA enables an application to directly access all the GPUs in the cluster, thus boosting its performance.

Therefore, the only limit is imposed by the characteristics of the applications and the ability of the programmer to extract parallelism.

In summary, this chapter presents a CUDA-compatible (supporting all CUDA libraries) GPGPU virtualization solution that, in addition to promoting green computing, provides applications access to a virtually unlimited number of GPUs, thus making the use of GPU accelerators in the HPC context even more beneficial. This chapter also presents, for the first time in the rCUDA project, a comprehensive performance evaluation and analysis of the rCUDA framework. This performance analysis is essential to provide a production ready solution.

The rest of the chapter is organized as follows. The `CU2rCU` tool is described and analyzed in Section 3.2; a performance evaluation of different benchmarks is carried out in Section 3.3; and the evolution of rCUDA to support multithreaded applications and CUDA libraries is presented in Section 3.4. In Section 3.5 we summarize the conclusions of this chapter.

3.2 CU2rCU: A CUDA-to-rCUDA Converter

This section motivates the need for a CUDA-to-rCUDA source-to-source converter; presents `CU2rCU`, a tool developed for that purpose; and describes the experiments carried out to evaluate the tool.

3.2.1 Need for a CUDA-to-rCUDA Converter

A CUDA program can be viewed as a regular C program where some of its functions have to be executed by the GPU or device instead of the traditional CPU or host. Programmers usually control the CPU-GPU interaction via the CUDA Runtime API for GPGPU programming. This API includes CUDA extensions to the C language, which are constructs that follow a specific syntax designed to make CUDA programming more accessible, usually leading to fewer lines of source code than its plain C equivalent (though both codes tend to look similar). The code in Listing 3.1 shows an example of a "hello world" program in CUDA. In this example, the functions *cudaMalloc* (line 13), *cudaMemcpy* (lines 15 and 19), and *cudaFree* (line 21) belong to the plain C API of CUDA, whereas the kernel launch sentence in line 17 uses the syntax provided by the CUDA extensions.

CUDA programs are compiled with the NVIDIA `nvcc` compiler [34], which detects fragments of GPU code within the program and compiles them separately from the

```
1 #include <cuda.h>
2 #include <stdio.h>

4 // Device code
5 __global__ void helloWorld(char* str) {
6   // GPU tasks.
7 }

9 // Host code
10 int main(int argc, char **argv) {
11   const char h_str[] = "Hello World!";
12   // ...
13   cudaMalloc((void**)&d_str, size);
14   // copy the string to the device
15   cudaMemcpy(d_str, h_str, size, cudaMemcpyHostToDevice);
16   // launch the kernel
17   helloWorld<<< BLOCKS, THREADS >>>(d_str);
18   // retrieve the results from the device
19   cudaMemcpy(h_str, d_str, size, cudaMemcpyDeviceToHost);
20   // ...
21   cudaFree(d_str);
22   printf("%s\n", h_str);
23   return 0;
24 }
```

Listing 3.1: "Hello world" program.

CPU code. During this compilation process, references to structures and functions not made public in the CUDA documentation are automatically inserted into the CPU code. These undocumented functions impair the creation of tools that need to replace the original CUDA Runtime Library from NVIDIA. To overcome this problem, we initially decided not to support these undocumented functions in rCUDA; instead, we offered a compile-time work-around that avoids their use. Notice that doing so requires bypassing `nvcc` for CPU code generation, since this compiler automatically inserts references to them into this code. Therefore, the CPU code in a CUDA program should be directly managed by a regular C compiler (e.g., GNU `gcc`). On the other hand, since a plain C compiler cannot deal with the CUDA extensions to C, they should be *unextended* back into plain C. Since manually performing these changes for large programs is a tedious, sometimes error-prone task, we have developed an automatic tool that modifies a CUDA source code with CUDA extensions and transforms it into its plain C equivalent. This automatic tool is a major contributor to improving the experience of those using the rCUDA framework. With this new tool, when a given CUDA source code is to be compiled for execution within the rCUDA framework, it is split into two parts:

- Host code: processed with a backend compiler such as GNU `gcc` (for either C or C++ languages), after being unextended, and executed on the host
- Device code: compiled with the `nvcc` compiler and executed on the device.

```
#define ALIGN_UP(offset, align) (offset) = \
((offset) + (align) - 1) & ~((align) - 1)

int main() {
  // ...
  // The following code lines replace line 17 of the previous
  // piece of code, where kernel "helloWorld" was launched:
  cudaConfigureCall(BLOCKS, THREADS);
  int offset = 0;
  ALIGN_UP(offset, __alignof(d_str));
  cudaSetupArgument(&d_str, sizeof(d_str), offset);
  cudaLaunch("helloWorld");
  // ...
}
```

Listing 3.2: Unextending line 17 of "Hello world" program.

Revisiting the previous "hello world" CUDA example, the code snippet in Listing 3.2 shows the transformation into plain C of the kernel call in line 17 employing the extended syntax.

To separately generate CPU and GPU code, we leverage a feature of `nvcc` that allows one to extract and compile only the device code from a CUDA program and generate a binary file containing only the GPU code. For the host code, once the CUDA extensions to C have been transformed into code using only the plain C CUDA API, we generate the corresponding binary file with a backend C compiler. Notice that prior to using a regular C compiler, the GPU code should additionally be removed from the program code, since regular C compilers cannot cope with such code. The separation and transformation process is illustrated in Figure 3.1.

Notice that the code transformations we propose, along with the specific usage of the compilers mentioned above, modify the compilation flow. In the original CUDA compilation flow, the input program was separated into host code and device code. During that process, the device code was transformed into binary code and embedded into the previously separated host code. The host code with the binary device code embedded was then compiled with `gcc`, generating an executable that had the binary device code embedded. On the contrary, in the rCUDA compilation flow, the `CU2rCU` converter is initially applied to the input program in order to obtain its equivalent source

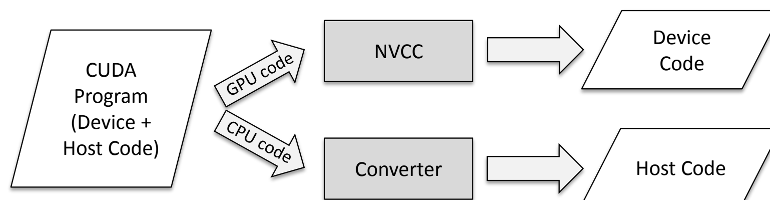


Figure 3.1: CUDA-to-rCUDA conversion process.

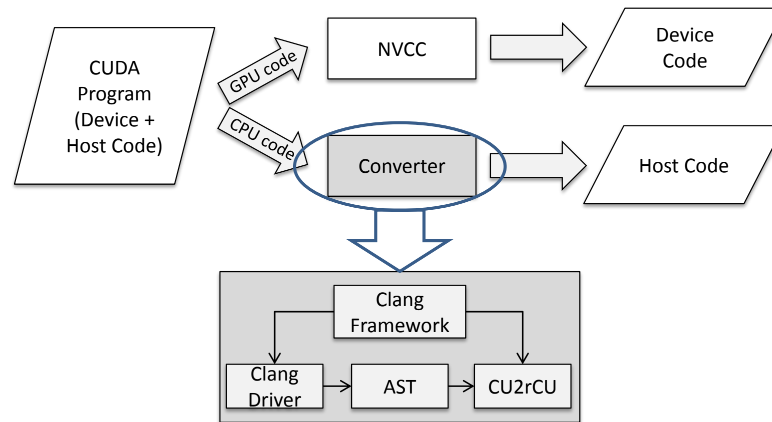


FIGURE 3.2: CUDA-to-rCUDA converter detailed view.

code using only plain C and without device code. From this converted code, the tool produces an executable that has references to device code stored in an external repository generated with `nvcc` by using the appropriate option.

3.2.2 Implementing the `CU2rCU` Converter

A source-to-source transformation framework was leveraged in order to implement the automatic tool that transforms source code employing CUDA extensions into plain C code. Different options for this class of source transformations are available, from simple pattern string replacement tools to frameworks that parse the source code into an abstract syntax tree (AST) and transform the code using that information. Since our tool needs to do complex transformations involving semantic C++ code information, we have selected the latter approach.

Several frameworks are available to tackling complex source transformations, for example ROSE [35], GCC [36], and Clang [37]. We have chosen Clang because it is widely used and explicitly supports programs written in CUDA. Moreover, some converters of CUDA source code exist that are also based on Clang, such as `CU2CL` [38], which transforms code from CUDA to OpenCL.

Clang, one of the primary subprojects of LLVM [39], is a C language family compiler that aims at providing a platform for building source code level tools, including source-to-source transformation frameworks.

Figure 3.2 shows how the developed converter interacts with Clang. The inputs to the converter are CUDA source files containing device and host code with CUDA extensions, as explained in the previous subsection. The Clang driver (a compiler driver providing access to the Clang compiler and tools) parses those files generating

an AST. The Clang plugin that we have developed, `CU2rCU`, then uses the information provided by the AST and the libraries contained in the Clang framework to perform the needed transformations, generating new source files that contain only host code with the plain C syntax. During the conversion `CU2rCU` automatically analyzes user source files included by the input files, converting them when necessary. Some of the most important transformations are detailed next.

3.2.2.1 Kernel Calls

A kernel call employing CUDA C extensions, as for example:

```
1 kernelName <<< Dg, Db >>>(param_1 ,... , param_n);
```

must be transformed in order to use the plain C API as follows:

```
1 cudaConfigureCall(Dg, Db);
2 int offset = 0;
3 setupArgument(param_1, &offset);
4 setupArgument(..., &offset);
5 setupArgument(param_n, &offset);
6 cudaLaunch("MangledkernelName");
```

The function `setupArgument()` is provided by the rCUDA framework. It is a wrapper of the plain C API function `cudaSetupArgument()` and, therefore, it just simplifies the inserted code by avoiding the need to explicitly handle argument offsets.

3.2.2.2 Kernel Names

In the `cudaLaunch()` call inserted in the previous transformation, the mangled kernel name must be used if it is not a function with external C linkage. Otherwise, the kernel name as written must be used. For instance, if we have the following kernel declaration:

```
1 __global__ void increment_kernel(int* x, int y);
```

its mangled name may be used when launching this kernel:

```
1 cudaLaunch("_Z16increment_kernelPii");
```

However, if the kernel is declared with external C linkage:

```
1 extern "C"
2 __global__ void increment_kernel(int* x, int y);
```

the original kernel name has to be used instead.

Determining the mangled kernel name becomes a complex task when there are kernel template declarations with type dependent arguments. For example, for the kernel template declaration:

```
1 template<class TData> __global__ void testKernel(
2   TData *d_odata, TData *d_idata, int numElements);
```

the mangled kernel name used to launch it depends on the type of `TData`:

```
1 if((typeid(TData) == typeid(unsigned char))) {
2   cudaLaunch("_Z10testKernelIhEvPT_S1_i");
3 } else if((typeid(TData) == typeid(unsigned short))) {
4   cudaLaunch("_Z10testKernelItEvPT_S1_i");
5 } else if((typeid(TData) == typeid(unsigned int))) {
6   cudaLaunch("_Z10testKernelIjEvPT_S1_i");
7 }
```

3.2.2.3 CUDA Symbols

When using CUDA symbols as function arguments, they can be either a variable declared in device code or a character string naming a variable that was declared in device code. As the device code has been removed, only the second option becomes feasible. For this reason, those occurrences that fall into the first category have to be transformed. For instance, in the following function call:

```
1 __constant__ float symbol[256];
2 float src[256];
3 cudaMemcpyToSymbol(symbol, src, sizeof(float)*256);
```

the argument `symbol` has to be surrounded by quotation marks to transform it into a character string:

```
1 cudaMemcpyToSymbol("symbol", src, sizeof(float)*256);
```

3.2.2.4 Textures and Surfaces

Similarly to CUDA C extensions, in order to use the C++ high level API functions from the CUDA Runtime API, an application needs to be compiled with the `nvcc` compiler. However, as within the rCUDA framework application source code needs to be compiled with a GNU compiler, we need to transform these functions. This is the case of CUDA textures and surfaces. Thus, textures declared using this API, like:

```
1 texture<float, 2> textureName;
```

are transformed as follows:

```
1 textureReference *textureName;
2 cudaGetTextureReference((const textureReference **) &textureName,
          "textureName");
```

A consequence of this transformation is that texture variables become pointers, and access to their attributes such as:

```
1 textureName.attribute = value;
```

will now result in:

```
1 textureName->attribute = value;
```

The same transformations explained for CUDA textures apply to CUDA surfaces.

Finally, it is worth to mention that we have validated the correctness of the implemented converter on several CUDA programs, as explained in next section, confirming that the functionality of the original programs has been fully preserved.

3.2.3 Evaluation of the CU2rCU converter

To test the new `CU2rCU` tool, we have used sample codes from the NVIDIA GPU Computing SDK [31] and the production code of the LAMMPS Molecular Dynamics Simulator [40].

Our first experiments dealt with a representative set of examples from the NVIDIA GPU Computing SDK. Table 3.1 shows the time[1] required for their conversion in seconds. In the experiments we employed a desktop platform equipped with an Intel(R) Core(TM) 2 DUO E6750 processor (2.66 GHz, 2 GB RAM) and a GeForce GTX 590 GPU, running Linux OS (Ubuntu 10.04). We used `nvcc` version 5, as well as Clang version 3.0. Table 3.1 also reports the number of lines of the original application and the modified sources obtained by our tool. The total time required for the automatic conversion of all these examples, 10.26 seconds, compared with the time spent on a manual conversion by an expert from the rCUDA team, 30.5 hours, clearly shows the benefits of using the converter. We note that the tasks of manually converting the code, on the one hand, and those for creating the converter, on the other hand, were fully disconnected among them, being carried out by different people, in order to avoid a bias in the comparison. We note also that an automatic source code conversion leads to a slightly larger amount of modified lines (though some of them correspond to sentences split into two lines). Nevertheless, the code automatically generated is similar to the one obtained from a manual conversion.

In addition to testing the converter with NVIDIA SDK codes, we have evaluated it on a real-world production code: the LAMMPS molecular dynamics simulator. This is a classic molecular dynamics code that can be used to model the interactions among atoms or, more generically, as a parallel particle simulator at the atomic, meso, or continuum

[1] Conversion and compilation time shown in this section have been gathered in an iterative way so that a given compilation (or conversion) has been repeated until the standard deviation of the measured time was lower than 5%.

Table 3.1: CU2rCU Conversion Statistics.

CUDA SDK Sample		Time (s)	Lines of Code		
			CUDA	Modified/Added	
Name	Acronym		Code	Num.	%
alignedTypes	AT	0.259	186	32	17.20
asyncAPI	AP	0.191	78	6	7.69
bandwidthTest	BT	0.407	708	0	0.00
BlackScholes	BS	0.364	281	13	4.63
clock	CL	0.196	75	8	10.67
concurrentKernels	CK	0.196	100	11	11.00
convolutionSeparable	CS	0.591	319	18	5.64
cppIntegration	CI	0.685	129	12	9.30
dwtHaar1D	DH	0.221	266	11	4.14
fastWalshTransform	FW	0.360	241	20	8.30
FDTD3d	FD	1.082	860	13	1.52
matrixMul	MM	0.394	272	34	12.50
mergeSort	MS	0.917	1124	105	9.34
scalarProd	SP	0.358	138	10	7.25
scan	SC	0.548	359	26	7.24
simpleAtomicIntrinsics	SA	0.367	211	6	2.84
simpleMultiCopy	SM	0.202	211	22	10.43
simpleTemplates	ST	0.211	241	13	5.39
simpleVoteIntrinsics	SV	0.196	222	19	8.56
SobolQRNG	SQ	1.278	10586	8	0.08
sortingNetworks	SN	0.761	571	70	12.26
template	TE	0.357	97	7	7.22
vectorAdd	VA	0.192	88	8	9.09
LAMMPS Molecular Dynamics Simulator					
Package USER-CUDA	LA	6.910	14742	1409	9.56

scale. The entire application comprises more than 300,000 lines of code distributed over 30 packages. Some of those packages are written for CUDA, such as *GPU* or *USER-CUDA*, which are mutually exclusive. We have evaluated our tool against the USER-CUDA package, with over 14,000 lines of code. The bottom part of Table 3.1 shows the results of the conversion. The time spent by a CUDA expert to adapt the original code was two weeks with full-time dedication.

Moreover, we have compared the time spent in the compilation of the original CUDA source code of the SDK samples and LAMMPS with the period spent in their conversion and subsequent compilation of the converted code by our tool. The results, shown in Figure 3.3, demonstrate that the time of converting the original code and later compiling it is similar to the compilation time of the original sources. To explain why both times

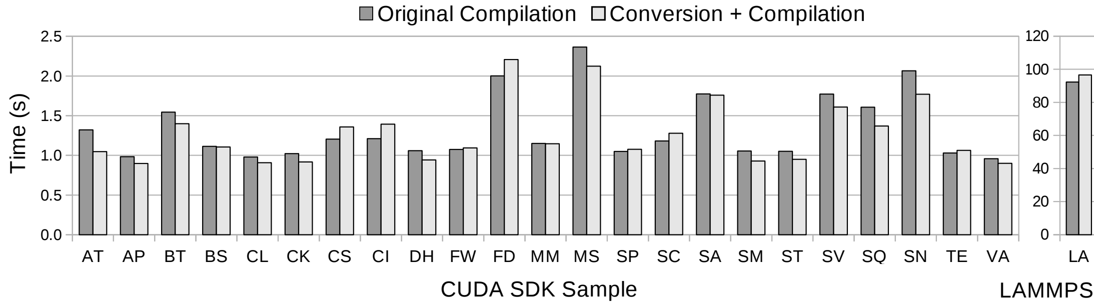


FIGURE 3.3: `nvcc` compilation time compared with `CU2rCU` conversion plus compilation time for CUDA SDK samples and LAMMPS.

TABLE 3.2: Comparison of CUDA and rCUDA Compilation Phases.

Compilation Phase	Times Each Phase is Executed	
	CUDA	rCUDA
`cu2rcu`	**0**	**1**
`gcc`	**6**	**5**
`cudafe`	2	2
`cudafe++`	1	1
`filehash`	**1**	**0**
`nvopencc`	1	1
`ptxas`	1	1
`fatbin`	1	1

are similar, in Table 3.2 we present a comparison of the phases in the CUDA and rCUDA compilation flows, in terms of how many times each phase of `nvcc` is invoked[2]. The `CU2rCU` compilation phase can be regarded as a `gcc` one because we are really calling Clang, which is also a C compiler. Therefore, we could state that, in both flows, the `gcc` compilation phase is actually executed six times. In the `CU2rCU` compilation phase we must also take into account that, apart from compilation time, we are doing source transforms, which also take time. In the rCUDA compilation flow, this time is compensated for, since (1) it does not require the `filehash` phase [34], present only in the CUDA flow, and (2) Clang is faster than `gcc`. Hence, the total time is similar in both cases.

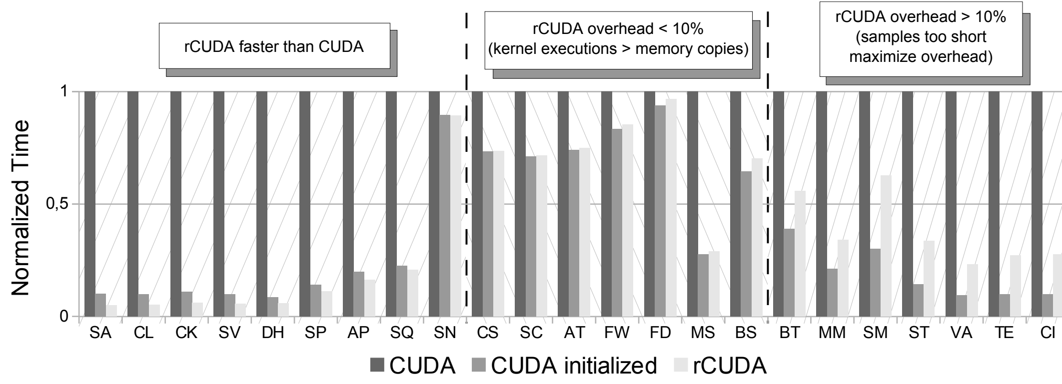
[2] The states `cudafe`, `cudafe++`, `filehash`, `nvopencc`, `ptxas`, and `fatbin` refer to calls to NVIDIA internal compilation tools, while `gcc` refers to calls to the GNU compiler. All these calls are automatically performed by the NVIDIA `nvcc` compiler and, therefore, are transparent to users. For details about why each of the tools are called and about what happens at each step, see [34].

3.3 Performance evaluation

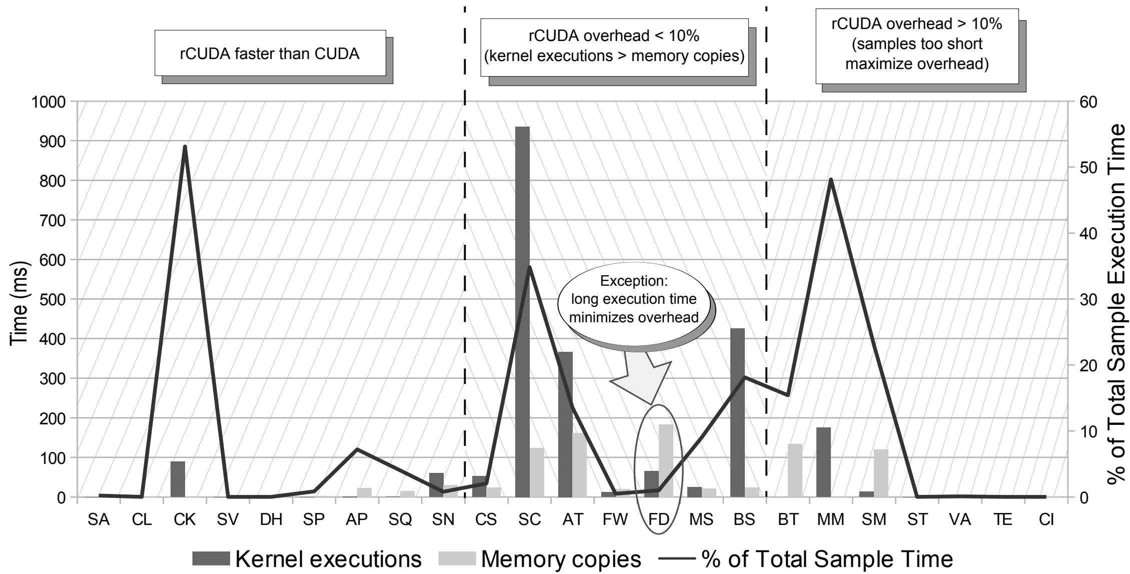
Once we can run any CUDA source code with the rCUDA framework thanks to the `CU2rCU` tool described in the previous section, we are now in the position to carry out a thorough performance evaluation. Notice that apart from the referred tool to adapt programs to the rCUDA framework, the version of rCUDA used in the following experiments (version 4) also includes an enhanced internal architecture (see Table 2.2 in Chapter 2), which is part of a previous thesis [16].

In Figure 3.4(a) we introduce the performance of the SDK samples converted in the previous section. This figure presents normalized execution time for CUDA and rCUDA over a QDR InfiniBand fabric. Notice that a raw comparison between rCUDA and CUDA is intrinsically unfair, since the rCUDA framework initializes the CUDA environment in the remote GPU server only once, during the rCUDA daemon start-up, whereas CUDA does the initialization each time an application is executed. Hence, when an application leverages CUDA, it has to wait for the initialization of the CUDA environment. On the contrary, applications making use of rCUDA do not have to wait for such initialization because it is already done in the rCUDA server. Therefore, to provide a complete view of the rCUDA performance, we have also included in Figure 3.4(a) the execution time for the local CUDA but with the CUDA environment already initialized (i.e., the GPU environment previously initialized by other process), labeled as *CUDA initialized* in the figure.

As shown in Figure 3.4(a), the execution time for rCUDA is, in general, noticeably smaller than that of CUDA, despite the access to a remote GPU. The reason is the initialization overhead. This can be clearly observed when introducing into the comparison the CUDA-initialized execution time, which is also noticeably smaller than that of CUDA and, at the same time, similar to the rCUDA execution time in most of the samples. Table 3.3 shows that the CUDA initialization time is almost constant for all these samples. Since the execution time for the major part of the samples is short (less than 2 seconds), this initialization time (over 1.3 seconds) impairs the aim of these experiments, which is showing rCUDA's overhead. In this regard, we can also observe that, for long samples, the difference between CUDA and rCUDA is smaller. For example, in the FD and SN samples (over 26 and 12 seconds, respectively), the influence of the initialization time is blurred. Hence, to unmask the actual delay introduced by the network, as well as the overhead of the rCUDA framework, from now on we will compare rCUDA execution time versus that of CUDA initialized. In this regard, we can classify the relative performance between CUDA initialized and rCUDA into three main cases: one including those samples for which rCUDA presents an overhead higher than 10% (right side of the plot), one including the samples presenting an overhead lower



(a) CUDA SDK normalized execution time compared with CUDA leveraging the GPU previously initialized by other process (CUDA initialized) and rCUDA. Executions for CUDA were carried out using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by a QDR InfiniBand network.



(b) Bars represent the time employed in kernel executions (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcpy) for each one of the CUDA SDK samples, whereas the line reports the percentage that the sum of them (CUDA kernels + CUDA memcopy) represent with regard to the total execution time of the CUDA initialized samples.

FIGURE 3.4: Performance of the SDK samples converted in the previous section.

than 10% (middle part of the plot), and one including the samples that execute faster within the rCUDA framework than with regular CUDA initialized (left side of the plot). Next, we analyze these three cases in more detail.

To explain the results in Figure 3.4(a), we have used the NVIDIA profiling tools [41] to measure the time spent in transfers (i.e., time spent in memory copies between host memory and the device memory, also referred to as *CUDA memcopy*) and the time employed by computations (i.e., time employed by CUDA kernels). Figure 3.4(b) details these measurements, as well as the cost that this time (CUDA memcopy + CUDA kernels) represents with respect to the total execution time of the samples (plotted

Table 3.3: Initialization time of the CUDA environment in the analyzed CUDA SDK Samples.

CUDA SDK Sample	Time (s)			
	CUDA	CUDA Initialization	CUDA Initialized	rCUDA
Samples that run faster with rCUDA than with CUDA				
simpleAtomicIntrinsics	1.490	1.340	0.150	**0.076**
clock	1.488	1.341	0.147	**0.079**
concurrentKernels	1.519	1.351	0.168	**0.093**
simpleVoteIntrinsics	1.488	1.341	0.147	**0.084**
dwtHaar1D	1.491	1.363	0.128	**0.089**
scalarProd	1.559	1.340	0.219	**0.175**
asyncAPI	1.675	1.341	0.334	**0.273**
SobolQRNG	1.732	1.340	0.392	**0.359**
sortingNetworks	12.735	1.325	11.410	**11.369**
Samples that present an overhead with rCUDA <10%				
convolutionSeparable	5.057	1.342	**3.715**	3.717
scan	4.269	1.229	**3.040**	3.059
alignedTypes	5.274	1.367	**3.907**	3.926
fastWalshTransform	8.062	1.335	**6.727**	6.887
FDTD3d	26.506	1.648	**24.858**	25.646
mergeSort	1.854	1.340	**0.514**	0.538
BlackScholes	3.839	1.361	**2.478**	2.696
Samples that present an overhead with rCUDA >10%				
bandwidthTest	2.215	1.351	**0.864**	1.237
matrixMul	1.725	1.360	**0.365**	0.589
simpleMultiCopy	1.920	1.342	**0.578**	1.205
simpleTemplates	1.564	1.341	**0.223**	0.528
vectorAdd	1.490	1.350	**0.140**	0.346
template	1.488	1.341	**0.147**	0.404
cppIntegration	1.488	1.340	**0.148**	0.412

with the continuous black line). As shown in Figure 3.4(a), for samples CS, SC, AT, FW, FD, MS, and BS, the overhead introduced by rCUDA is bearable (i.e., lower than 10%). The reason is that all these samples spend, in general, more time in computations than in transfers. This property benefits rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating for the overhead of rCUDA due to the transfers across the network. The only exception is sample FD, in which the time spent transferring data is greater than the computations time. The explanation for the low overhead of this sample is that it has a long duration (over 24 seconds with CUDA initialized), so that the overhead of transferring data across the network is minimized.

In Figure 3.4(a), there are also samples for which the overhead introduced by rCUDA is

considerable (i.e., higher than 10%). This is the case of BT, MM, SM, ST, VA, TE, and CI. The explanation for samples BT and SM is that they spend more time in transfers than in computations (see Figure 3.4(b)). The execution time for the rest of the samples is too short, less than 0.5 seconds, which maximizes their overheads just opposite the way long samples minimize it.

Surprisingly, some samples run faster with rCUDA than with CUDA initialized. This is the case of the SA, CL, CK, SV, DH, SP, AP, SQ, and SN. Even if we assume that the overhead of the rCUDA framework is negligible, they still should present worse results than those of CUDA initialized, given that the additional delay introduced by the network will still be present. Figure 3.4(b) reveals that these samples present few memory transfers, so that the network overhead is negligible, but this fact still does not explain why rCUDA outperforms CUDA initialized.

A deeper profiling revealed that the analyzed samples have synchronization points, such as calls to `cudaDeviceSynchronize` or `cudaStreamWaitEvent`, that take more time when using CUDA than when using our framework. For instance, the unique call to `cudaDeviceSynchronize` in sample AP takes 529 microseconds in CUDA, whereas it takes only 41 microseconds in rCUDA. The reason for this lies in the internal algorithm used in the rCUDA framework used to determine the finalization of the CUDA tasks, which favors rCUDA in these simple samples. Briefly, this algorithm performs a nonblocking wait during a small period of time before calling the synchronization function. If this nonblocking wait is successful, the synchronization function is not called, and control is immediately returned to the program. Thus, less time is used at synchronization points.

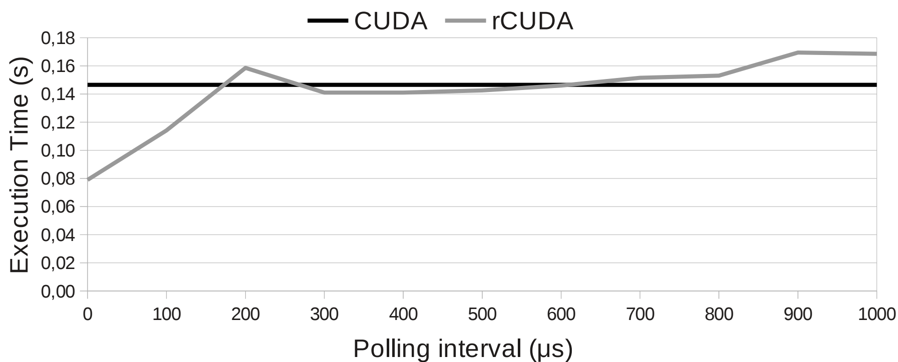
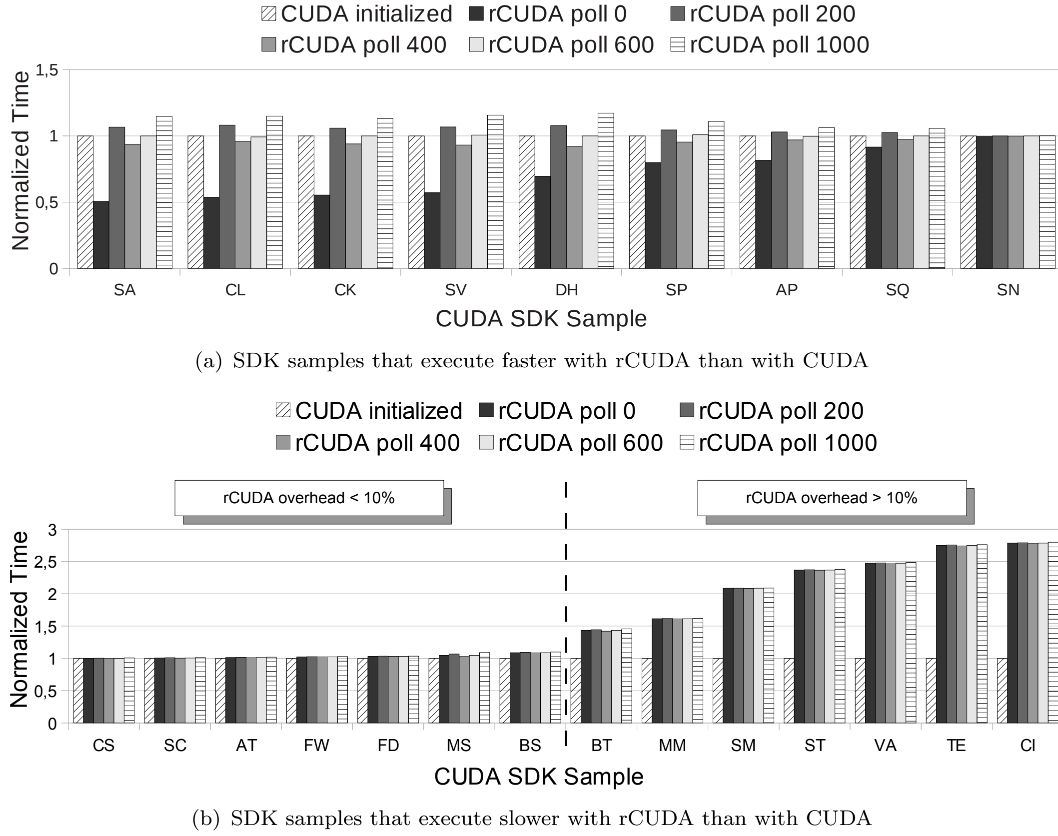


FIGURE 3.5: Execution time of the CL sample in CUDA compared with rCUDA using different intervals for polling the network devices. The value for the polling interval within CUDA is the default one used by the CUDA driver. Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by a QDR InfiniBand network.



(a) SDK samples that execute faster with rCUDA than with CUDA

(b) SDK samples that execute slower with rCUDA than with CUDA

FIGURE 3.6: CUDA SDK normalized execution time with the GPU previously initialized by other process (CUDA initialized) compared with rCUDA using different intervals for polling the network devices (0, 200 μs, 400 μs, 600 μs, and 1000 μs). Executions for CUDA were done using one node equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. For the rCUDA experiments, two nodes with the same specifications as the node employed with CUDA were used, both connected by a QDR InfiniBand network.

Nevertheless, the time saved in synchronization points in rCUDA does not fully explain the shorter execution time of these samples. For instance, for sample AP, the execution time with rCUDA is 61 ms faster than with CUDA initialized (see Table 3.3), while the time saved (for the reasons explained above) represents only 0.488 ms. After further research, we found an additional factor that affects execution time: the network polling interval (i.e., the frequency used to poll the network for work completions). This interval in the rCUDA implementation is lower than the default polling interval to the PCIe made by CUDA. To demonstrate this, in Figure 3.5 we report the execution time of the CL sample when incrementing the rCUDA network polling interval. As we can observe, this factor clearly affects the final execution time, making rCUDA perform better or worse than CUDA initialized depending on its value. Figure 3.6(a) presents the normalized execution time of all samples for which rCUDA is faster than CUDA initialized, but using different polling intervals, confirming the relevance of this interval. Figure 3.6(b) presents similar results for the rest of SDK samples analyzed in Table 3.3.

3.4 Support for multithreaded applications and CUDA libraries

Typically, providing support within rCUDA for the new features introduced in a given CUDA version just requires the implementation of new functions in the rCUDA framework. The exact effort needed to address each new version of CUDA depends on the number of new functions to be introduced into rCUDA and the impact of each new CUDA feature on the rCUDA framework. In this regard, introducing the new functionalities provided by CUDA 4 and 5 into rCUDA basically meant, from a simplistic point of view, increasing the rCUDA code. However, in practice, some of the new features introduced in CUDA 4 and 5 affected rCUDA far more deeply. More specifically, supporting CUDA 4 implicitly implied dealing with multithreaded applications, whereas the appearance of CUDA 5 brought a new way to deal with CUDA libraries. In this section we highlight how rCUDA was evolved in order to integrate both new features.

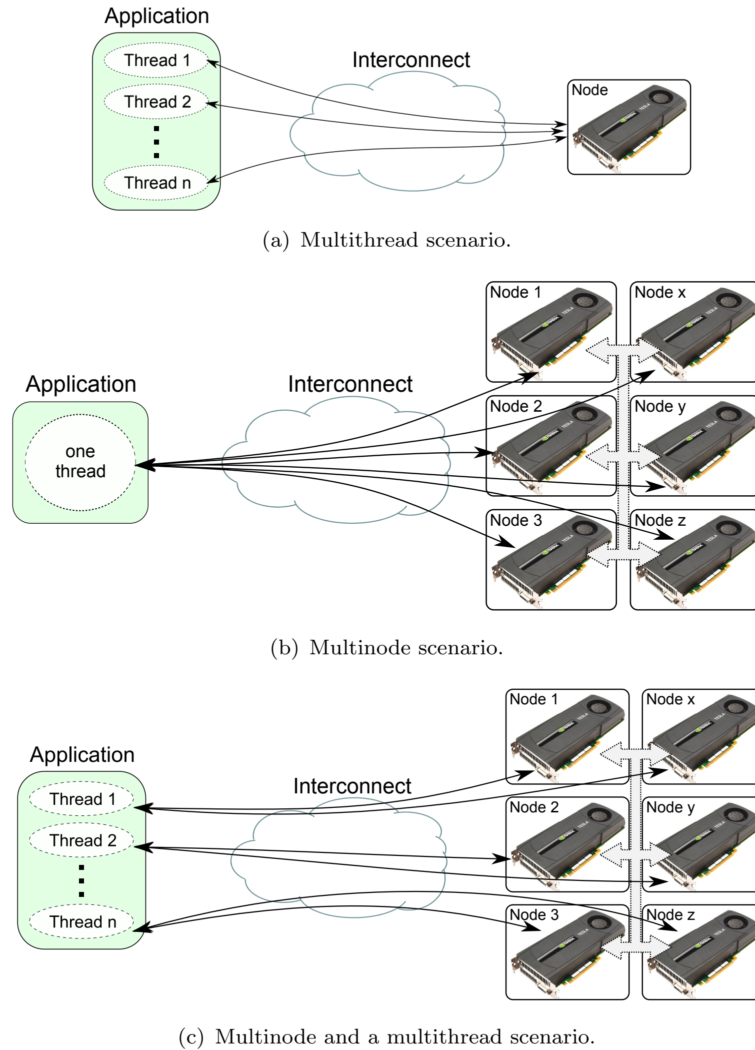
3.4.1 Support for Multithreaded Applications

The support for multithreaded applications (first introduced in CUDA 4) has made rCUDA evolve from a non-thread-safe framework to a thread-safe one. This evolution required deep changes in the internal structure of rCUDA. The way rCUDA provides support for multithreaded applications will be thoroughly analyzed next.

Figure 3.7(a) illustrates one possible scenario, where all the threads of a multithreaded application access the same remote GPU, which is shared among them. The improvements to rCUDA reach further than just supporting new CUDA features; indeed, the new release also allows an application to access all remote GPUs located in different nodes. We refer to this new feature as multinode support, as shown in Figure 3.7(b).

The combination of the new capabilities of rCUDA enables the scenario represented in Figure 3.7(c), where each thread of a multithreaded application can access remote GPUs located in different nodes.

In the following we present some experiments to analyze these new features. In all the results shown in this section, CUDA experiments were carried out in a node equipped with 2 Quad-Core Intel Xeon E5440 processors and a Tesla S2050 computing system (4 Tesla GPUs); and rCUDA executions were done using 8 nodes, each equipped with 4 Quad-Core Intel Xeon E5520 processors and one NVIDIA Tesla C2050. The QDR InfiniBand fabric was leveraged for rCUDA.



(a) Multithread scenario.

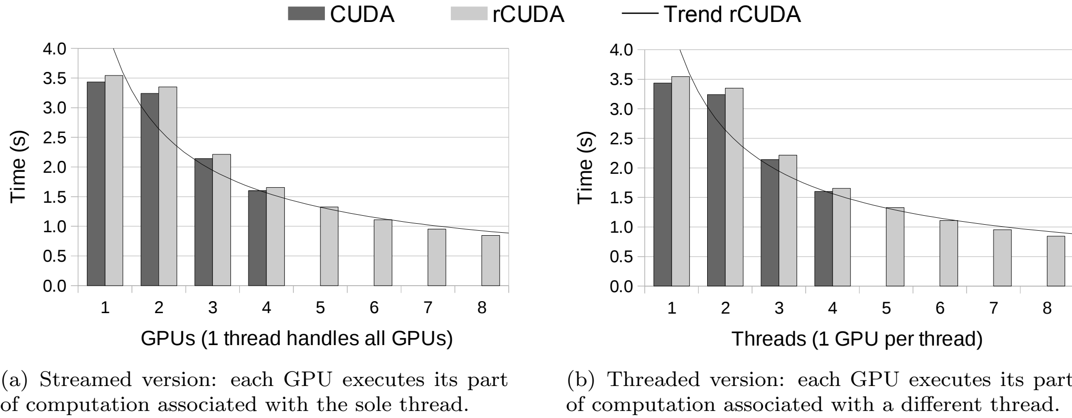
(b) Multinode scenario.

(c) Multinode and a multithread scenario.

FIGURE 3.7: Using rCUDA in different scenarios.

In the first experiment, we used the MonteCarloMultiGPU sample from the NVIDIA GPU Computing SDK, which evaluates fair call price for a given set of European stock options using the Monte Carlo method. We have modified this sample in order to operate on a bigger problem size (a set of 2,048 stock options) and to allow us to specify the number of GPUs to use in the evaluations.

Figures 3.8(a) and 3.8(b) display the total computation time for this SDK sample. In the experiments, the problem size is constant for all the executions (2,048 stock options), while the number of GPUs and threads involved in the computation varies. This approach allows us to compare the results from different executions in order to assess whether the use of more GPUs and/or threads really reduces the total computation time. It also allows us to compare the scalability features of regular CUDA with those of rCUDA.



(a) Streamed version: each GPU executes its part of computation associated with the sole thread.

(b) Threaded version: each GPU executes its part of computation associated with a different thread.

Figure 3.8: Total computation time for the MonteCarloMultiGPU sample from the NVIDIA GPU Computing SDK. All the executions operate with the same problem size (2,048 stock options). "Trend rCUDA" refers to the trend line with power regression type for rCUDA results.

Figure 3.8(a) shows the results of the streamed version of the MonteCarloMultiGPU sample, where one CPU thread handles all GPUs. The problem size is divided among the employed GPUs. Hence, each GPU computes a part of the problem. In the single-GPU case, all the computation is done by that accelerator. Figure 3.8(b) shows the results for the multithreaded version of the MonteCarloMultiGPU sample in which there is one CPU thread for each GPU. The problem size is also divided among the employed GPUs, but this time each GPU is handled by a different thread.

As shown in Figures 3.8(a) and 3.8(b), both versions of the MonteCarloMultiGPU sample, the streamed one and the multithreaded one, provide similar results. Additionally, the total computation time in both versions improves when increasing the number of GPUs. Furthermore, rCUDA not only mimics the behavior of the original CUDA in terms of scalability but also allows an application to use a larger number of GPUs. In this regard, remember that the four GPUs used in these experiments with CUDA were in the same node, while the eight GPUs used in the executions with the rCUDA framework were located in eight different nodes. The experiments with CUDA for 5, 6, 7, and 8 GPUs were not feasible because of the lack of a node which such an extraordinary equipment. Thus, Figures 3.8(a) and 3.8(b) clearly show how rCUDA allows the aggregation of all the GPUs in a cluster and makes them available to a single application. In summary, the limit is no longer the number of available GPUs, but the capability of the application programmer to exploit all of them.

Table 3.4 presents the overhead introduced by rCUDA in these experiments. In the worst case, a mere 3.4% of overhead is introduced. Notice that these tables consider only the cases up to four GPUs; data for regular CUDA beyond that number is not available because of the equipment limitations.

Table 3.4: Overhead introduced by rCUDA in MonteCarloMultiGPU sample.

	Streamed Version		Multi-threaded Version	
Num. of GPUs	Num. of Threads	rCUDA overhead (%)	Num. of Threads	rCUDA overhead (%)
1	1	3,162	1	3,159
2	1	3,424	2	3,361
3	1	3,384	3	3,421
4	1	3,381	4	3,373

To further illustrate the features of the new rCUDA version, we consider a more complex application, which extends the `libflame` library [42] to perform dense matrix computations taking advantage of multiple GPUs. As in the previous tests, CUDA experiments leveraging more than four GPUs were not feasible because of the lack of that class of equipment. Performance results for this application are depicted in Figure 3.9. As this figure shows, using more GPUs does not translate into higher performance in this case. The best results are achieved by using CUDA with 3 GPUs; the results were slightly worse with 4 GPUs. With rCUDA, timings improve up to 4 GPUs and remain almost constant when using more GPUs. Once again, rCUDA's behavior is similar to that of the original CUDA in terms of scalability. In this case, however, using a larger number of GPUs is not beneficial because of the nature of the application.

With respect to the overhead introduced by rCUDA in these experiments, Table 3.5 shows that it is higher than in the MonteCarloMultiGPU case. To explain this, we have measured separately the time spent in transfers and in computations for both applications (MonteCarloMultiGPU and the `libflame` matrix-matrix multiplication). The results are shown in Figure 3.10. Collecting data for more than four GPUs was

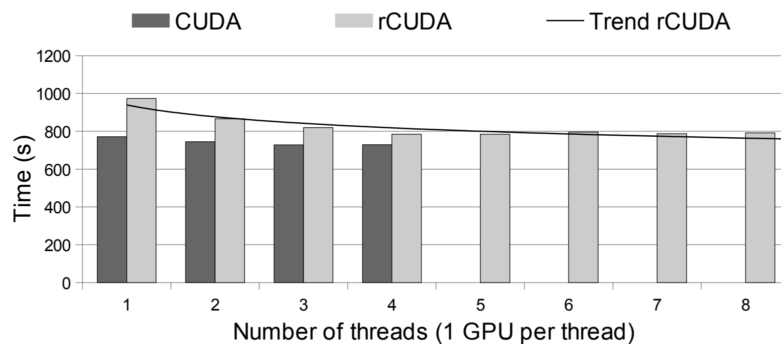
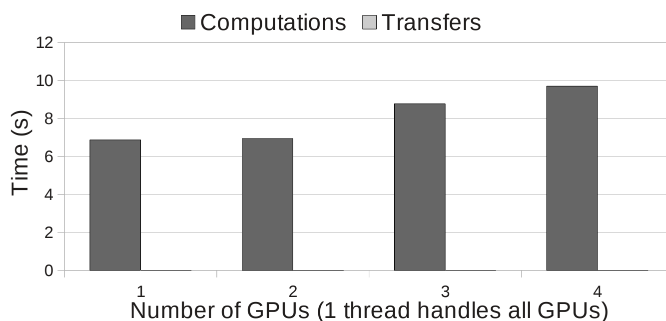


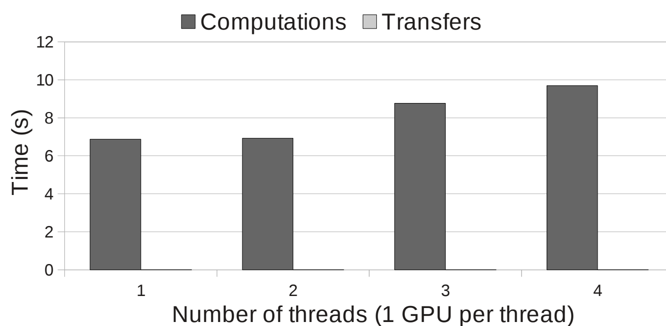
Figure 3.9: Total execution time for a matrix-matrix multiplication using the `libflame` library. All the executions operate with the same problem size (matrix of 18 million of single-precision elements). Each GPU executes the part of the computation associated with a different thread. "Trend rCUDA" refers to the trend line with power regression type for rCUDA results.

TABLE 3.5: Overhead introduced by rCUDA in the matrix-matrix multiplication using the `libflame` library.

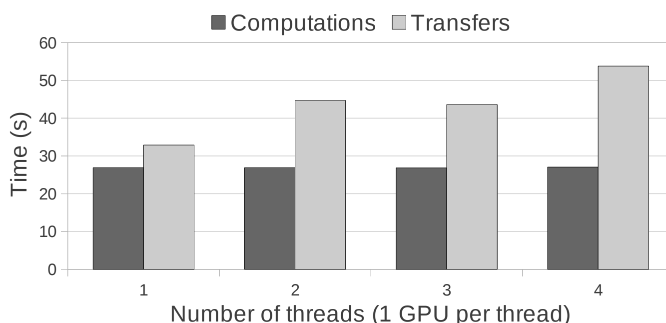
Num. of Threads	Num. of GPUs	rCUDA overhead (%)
1	1	26,245
2	2	16,284
3	3	12,597
4	4	7,668



(a) MonteCarloMultiGPU sample (streamed version).

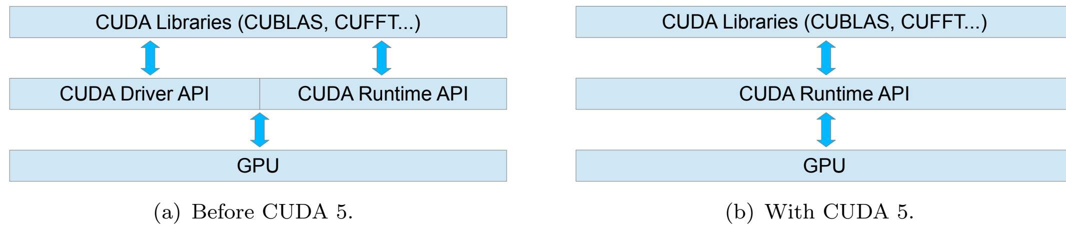


(b) MonteCarloMultiGPU sample (threaded version).



(c) Matrix-matrix multiplication using the `libflame` library.

FIGURE 3.10: NVIDIA profiling results for the different applications tested. In particular, time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcpy). Notice that the time depicted in the plots is the aggregation of the time used by each of the GPUs leveraged in the experiments.



(a) Before CUDA 5.

(b) With CUDA 5.

FIGURE 3.11: Integration of CUDA libraries in CUDA 5.

not possible because we leveraged the NVIDIA profiling tools for the measurements, which are not supported by rCUDA yet. As shown in this figure, both versions of MonteCarloMultiGPU spend almost all their time performing computations. In contrast, the matrix-matrix multiplication spends more time in transfers than with computations. Furthermore, the time spent in transfers increases because the number of GPUs involved in the execution is larger, thus reducing the scalability of the application. On the other hand, performing many more computations than transfers helps rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of rCUDA due to the transfers across the network. This explains the low overhead introduced by rCUDA in the MonteCarloMultiGPU example, as well as the higher overhead introduced in the matrix multiplication.

3.4.2 Support for CUDA Libraries

CUDA 5 introduced a significant change in the implementation of the CUDA libraries (CUBLAS or CUFFT, for example) and the way in which these libraries use the APIs offered by CUDA (Runtime and Driver APIs). These new features deeply affected rCUDA in terms of compatibility with CUDA. Before CUDA 5, the CUDA libraries used both the CUDA Runtime API and the CUDA Driver API. The latter was used indirectly by calling a function of the CUDA Runtime API (i.e., cudaGetExportTable) that simply bypassed the call to the CUDA Driver API function (i.e., cuGetExportTable). Since rCUDA does not offer support for the CUDA Driver API, the CUDA libraries were not supported. Nevertheless, in CUDA 5 these libraries have been modified, and now they use only the CUDA Runtime API. Since rCUDA already supports the CUDA Runtime API, the support for any library using the CUDA Runtime API is implicit. Figure 3.11 illustrates this aspect. The experiments presented in the previous section that used the `libflame` library also validate the support for CUDA libraries presented in this section, given that the functions of libflame internally make use of the CUBLAS library.

3.5 Summary

In this chapter we have shown the feasibility of creating a tool that automatically unextends CUDA source code, which leverages the CUDA extensions to the C language, to its plain C equivalent code. This tool overcomes the limitations imposed by the undocumented functions used by the NVIDIA `nvcc` compiler regarding remote GPU virtualization.

A thorough analysis of the performance of this tool shows that the time it requires to convert and compile a given CUDA code is very similar to the time required by the `nvcc` compiler, thus making it feasible to substitute the regular `nvcc`-based compilation flow by a `CU2rCU`-based one.

Regarding the program codes that can be used with the rCUDA framework, thanks to the enrichments presented in this chapter (the `CU2rCU` source-to-source converter and the multithread support), it is now possible to execute any CUDA program. As shown in the performance evaluation figures, CUDA programs that present high computation/transfer ratios will experience, in general, a much lower overhead than do programs that transfer large amounts of data to/from the GPUs. Notice, however, that even when using a local GPU with CUDA, a large amount of transfers to/from the GPU is not a good practice, given that such transfers also result into a performance reduction in the traditional local CUDA usage.

Finally, as previously commented at the beginning of this chapter, the research carried out for the development of the `CU2rCU` converter finally derived, in subsequent versions of rCUDA, in the binary compatibility with CUDA. This means that applications can be executed with rCUDA without requiring any conversion. Actually, applications can be executed with rCUDA even without being recompiled, provided that they were compiled using dynamic libraries. This new feature was, in practice, a tremendous step forward in the path to transfer rCUDA to industry. In following chapters, we will be using versions of rCUDA including this binary compatibility.

Chapter 4

Tuning rCUDA for InfiniBand Networks

Using a remote GPU introduces some overhead, mainly due to the virtualization framework and the network fabric. On the one hand, the GPU virtualization framework increases the latency to the real GPU, as requests must be forwarded to the remote GPU and responses delivered back to the application demanding GPU services. On the other hand, as GPUs are no longer located at the other end of a PCIe link within the host, but in a remote node, data have to traverse at least two PCIe links and two network interfaces, as well as the entire network fabric between the node requesting GPU services and the node where the actual GPU resides. Therefore, in addition to latency, bandwidth also suffers, given that PCIe bandwidth is usually noticeably larger than network bandwidth, thus increasing the performance gap between the local and remote uses of GPUs. In this regard, the performance of the interconnection network is key to achieve reasonable performance results by means of remote GPU virtualization.

Additionally, the ability of the remote GPU virtualization framework to squeeze as much performance as possible from the underlying interconnect is crucial for reducing overall overhead. In this chapter we present how rCUDA has been tuned for the high performance InfiniBand (IB) [43] network. We also introduce how the evolution of this interconnect influences the design of rCUDA as well as impacts the performance of this middleware. The work presented in this chapter has been published in [44–47].

Link with Industry

Once the rCUDA framework is binary compatible with CUDA, we are able to test its performance using real applications. Notice, however, that not only rCUDA is a live technology which evolves and improves over time, but also the underlying hardware and software make progress (see Figure 1.1). For this reason, it is also important to analyze the performance of a technology like rCUDA in the right industrial context, therefore using modern platforms. Although GPUs also become more powerful, the hardware which most influences the performance of rCUDA is the network, as we have already commented. In this regard, the InfiniBand interconnects experienced noticeable improvements during the implementation of this thesis. Thus, in 2011 FDR InfiniBand fabrics were introduced, attaining a bandwidth of 56Gbps. In 2013, dual-port FDR adapters appeared, and in 2015, EDR InfiniBand was introduced, both fabrics offering 100Gbps of bandwidth. This evolution of the InfiniBand network has also caused an evolution in the design and performance of the rCUDA middleware.

4.1 Introduction

The previous chapter has presented a first performance evaluation of the rCUDA remote GPU virtualization framework using mainly samples from the NVIDIA GPU Computing SDK, and using QDR InfiniBand as network fabric. In this chapter we evaluate the performance of rCUDA using production codes and more modern platforms. We specially focus on analyzing the impact of the newer InfiniBand interconnects on the performance of rCUDA.

On the other hand, it has been shown in Chapter 2, Figure 2.4, that the bandwidth attained by the rCUDA framework greatly depends on the direction of the memory copy (from host to GPU, or vice versa), as well as on the type of host memory used in the data transfer (pageable host memory or page-locked host memory). Therefore, it is necessary to properly design the rCUDA framework in order to get the best performance in every case.

This chapter details the performance analysis and optimizations carried out for InfiniBand networks in order to maximize the performance of rCUDA. First, in Section 4.2 we analyze the influence of FDR InfiniBand on the performance of the rCUDA remote GPU virtualization framework. Second, in Section 4.3 we present a new version of rCUDA supporting InfiniBand dual-port adapters, such as the InfiniBand Connect-IB (FDR x 2) ones, and also evaluate its performance. Next, in Section 4.4 we expose different optimizations to be used when developing applications using InfiniBand

Verbs and study their impact on rCUDA. Then, in Section 4.5 we analyze how the high bandwidth provided by EDR InfiniBand, along with the optimizations presented in previous sections, allows rCUDA not only to perform very similar to local GPUs, but also to improve overall performance for some applications. Finally, Section 4.6 summarizes the main contributions of this chapter.

4.2 Influence of FDR InfiniBand on the Performance of rCUDA

In order to minimize the impact of the network overhead, the latency and bandwidth of the interconnect should be comparable to those of PCIe. Although the performance gap between the internal PCIe interconnect and the external fabric was considerable in the past, there have appeared several networking technologies with throughput comparable to that of PCIe. As shown in Figure 4.1, the theoretical throughput of FDR InfiniBand is very close to that of PCIe 2 (PCIe version supported by GPUs used at this point of the rCUDA evolution). In this section we analyze the influence of FDR InfiniBand on the performance of rCUDA, comparing its impact on a variety of GPU-accelerated applications versus other networking technologies, such as QDR InfiniBand or Gigabit Ethernet. Results show that the FDR interconnect, featuring higher bandwidth than its predecessors, allows to reduce the overhead of using GPUs remotely, thus making this approach even more appealing.

The rest of the section is organized as follows. In Section 4.2.1 we compare the bandwidth and latency of PCIe with those of the networks considered in this work. In the next two sections we analyze the performance of several GPU applications using remote GPUs, again with different networks: we first use the NVIDIA CUDA Samples in Section 4.2.2;

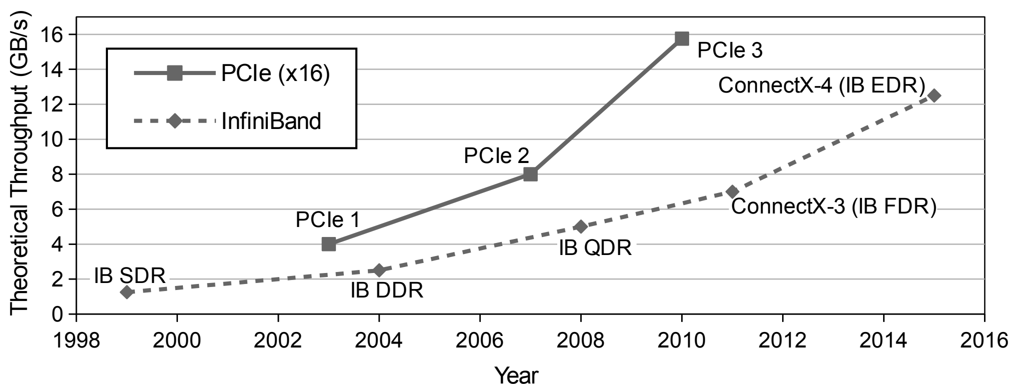


Figure 4.1: Comparison between the theoretical bandwidth of different versions of PCI Express x16 and those of commercialized InfiniBand (IB) fabrics.

and then we evaluate some GPU-accelerated production applications in Section 4.2.3. Finally, Section 4.2.4 summarizes the main conclusions of our Section 4.2.

4.2.1 Basic Performance Comparison of the Networks

The performance of data transfers to/from the remote GPU is mainly influenced by the bandwidth and latency of the communication path. When transferring small amounts of data, latency is the most important factor while, when transferring large blocks, bandwidth is crucial. In this section we start our analysis of the influence of a high bandwidth network such as FDR InfiniBand on the performance of remote GPU virtualization. To do so, we compare the bandwidth and latency of PCIe when using CUDA, with those observed for rCUDA over three different network technologies, namely FDR InfiniBand, QDR InfiniBand and also Gigabit Ethernet.

4.2.1.1 Testbed System

The setup employed for the experiments carried out in this section, which will also be the same for the rest of the tests presented along this section, consists of two servers, each with the following characteristics:

- Two Intel Xeon hexa-core processors E5-2620 (Sandy Bridge) operating at 2.00GHz
- 32 GB of DDR3 SDRAM memory at 1,333 MHz
- 1 Mellanox ConnectX-3 single-port InfiniBand Adapter
- CentOS Linux Distribution release 6.3, with Mellanox OFED 1.5.3 (InfiniBand drivers and administrative tools), CUDA 5.0 with NVIDIA driver 285.05, and rCUDA 4.0.1 (the latest stable release from February 2013)

Additionally, one of the nodes has an NVIDIA Tesla K20 GPU. On the other hand, both nodes are interconnected by a Gigabit Ethernet network with a Cisco SLM2014 switch, and also by an InfiniBand fabric. Two different Mellanox switches are leveraged for the InfiniBand fabric: an MTS3600 switch providing QDR compatibility, and an SX6025 for FDR compatibility. Depending on which of them is actually used, QDR or FDR features are leveraged.

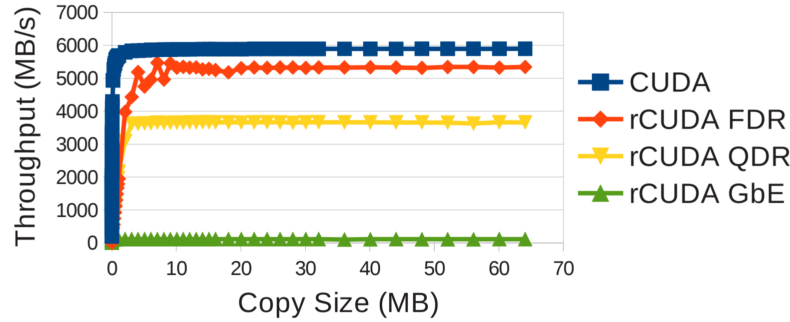


FIGURE 4.2: Bandwidth test for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over different networks.

4.2.1.2 Influence on Bandwidth

In this section we analyze the memory copy (`memcopy`) bandwidth between host memory and the device memory for several scenarios:

- Local GPU: `memcopy` bandwidth across PCIe (referred to as "CUDA" in the figures).
- Remote GPU: `memcopy` bandwidth for different networks: Gigabit Ethernet (rCUDA GbE), QDR InfiniBand (rCUDA QDR) and FDR InfiniBand (rCUDA FDR).
- Both the local and remote scenarios are evaluated using pageable host memory as well as pinned host memory.

In order to analyze the influence on bandwidth, we employ the `bandwidthTest` benchmark from the NVIDIA CUDA Samples, with the `shmoo` option, which performs memory copies of a large range of sizes. Figure 4.2 presents the results of this test using pinned host memory, while Figure 4.3 illustrates the results for pageable host memory. As expected, rCUDA over GbE provides the worst results, with its bandwidth reaching a maximum of 113.1MB/s in both cases (pinned and pageable). When using pinned host memory, rCUDA over FDR InfiniBand achieves a substantial gain (46.01%) with respect to QDR InfiniBand. Furthermore, its throughput is very close to that obtained by regular CUDA over a local GPU, with a difference of 0.5GB/s (9.4%). Regarding the use of pageable host memory, rCUDA over InfiniBand renders a higher bandwidth than native CUDA with a local GPU. This is due to the fact that memory copies between rCUDA clients and remote GPUs are pipelined using preallocated buffers of pinned memory, as explained in [15], exploiting thus the higher throughput of this type of memory.

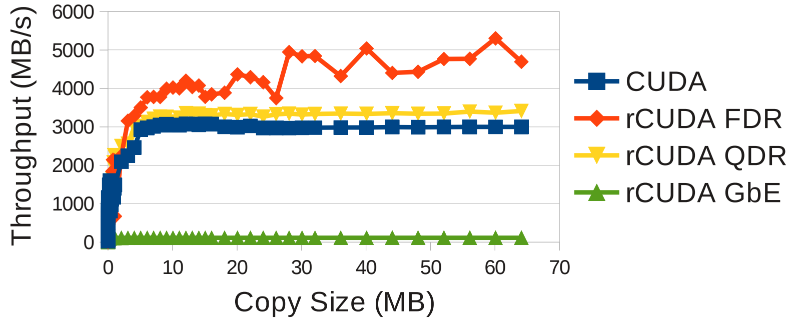


FIGURE 4.3: Bandwidth test for copies from host to device with pageable host memory, using CUDA and the rCUDA framework over different networks.

In summary, the higher bandwidth of FDR InfiniBand, in conjunction with a careful framework design, allows remote GPU virtualization frameworks to experience a bandwidth similar to that of PCIe all across the entire path between the local application demanding GPGPU services and the remote GPU.

4.2.1.3 Influence on Latency

In order to analyze the impact of the improvements on interconnect latency, we employ a synthetic test similar to the previous bandwidth test, but using negligible volumes of data (concretely, from 1 to 64 bytes). Table 4.1 shows the results for this experiment, obtained from the average of 100 repetitions of each scenario. Again, GbE presents the worst performance. On the other hand, Table 4.1 also reveals that the use of rCUDA over FDR InfiniBand does not improve latency with respect to QDR InfiniBand. However, standard deviation values exhibit a more constant behavior for FDR, demonstrating a higher stability.

To determine how latency to remote GPU affects application performance, we have implemented one additional synthetic benchmark which also copies a small dataset (64 bytes). However, in this case, it performs a varying number of copies, from 100 to

TABLE 4.1: Latency test using CUDA and the rCUDA framework over different networks.

	Time (μs)			
Copy size (bytes)	CUDA	rCUDA: FDR	QDR	GbE
1 (100-copy average)	11.62	50.73	50.34	130.63
2 (100-copy average)	11.56	50.53	50.49	130.05
4 (100-copy average)	11.59	50.32	50.32	130.59
8 (100-copy average)	11.55	50.69	50.26	130.68
16 (100-copy average)	11.56	50.71	50.06	130.50
32 (100-copy average)	11.67	50.64	50.22	133.03
64 (100-copy average)	11.71	50.87	50.12	135.18
Max. standard deviation	0.06	0.18	1.55	2.15

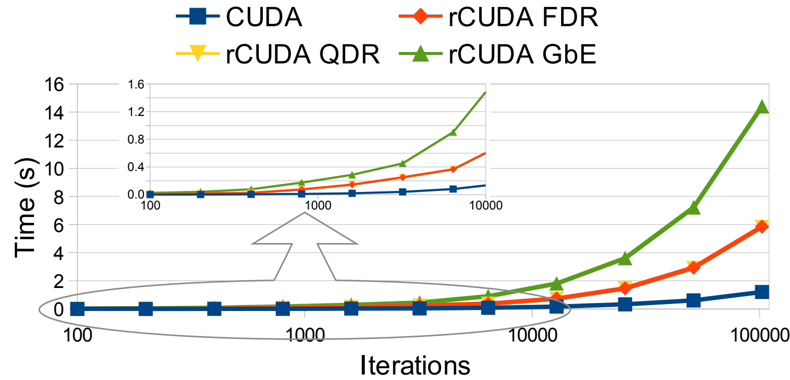


FIGURE 4.4: Latency test varying the number of iterations for CUDA and the rCUDA framework over different networks. X-axis in logarithmic scale.

102,400, doubling the number of copies in each iteration (i.e., 100, 200, 400, etc.). This resembles the behavior of applications, which typically will perform several sequential requests to the GPU along their execution. As Figure 4.4 shows, rCUDA over GbE performs very poorly starting from 25,000 iterations, whereas rCUDA over InfiniBand begins its degradation from 50,000 iterations. In general, the exact numbers show that QDR InfiniBand achieves slightly better results (an average of 5ms).

To sum up, the latency results show that, although the FDR InfiniBand version noticeably improves performance when the application can benefit from its superior bandwidth, this interconnect does not enhance performance, with respect to QDR InfiniBand, when the application is sensitive to latency.

4.2.2 NVIDIA CUDA Samples

In this section we leverage the entire NVIDIA CUDA Samples suite, also referred to as CUDA SDK samples, to analyze remote GPU virtualization performance in a comprehensive way. The NVIDIA CUDA Samples contain simple code programs covering a wide range of applications and techniques using CUDA, which we consider useful for an initial study.

Figure 4.5a presents the normalized sample execution time for CUDA and also for rCUDA over FDR InfiniBand, QDR InfiniBand, and GbE. Times are normalized to those obtained with local CUDA. The average of 10 repetitions is used, and the maximum Relative Standard Deviation (RSD) observed was 0.390 for sample `boxFilterNPP` (BN) when executed with CUDA, 0.147 for sample `transpose` (TA) in the case of rCUDA over FDR InfiniBand, 0.187 for sample `volumeRender` (VR) with rCUDA over QDR InfiniBand, and 0.563 for sample `simpleCUFFT` (FF) with rCUDA over GbE. In order to complete the data in Figure 4.5a, we have also measured the amount of bytes transferred

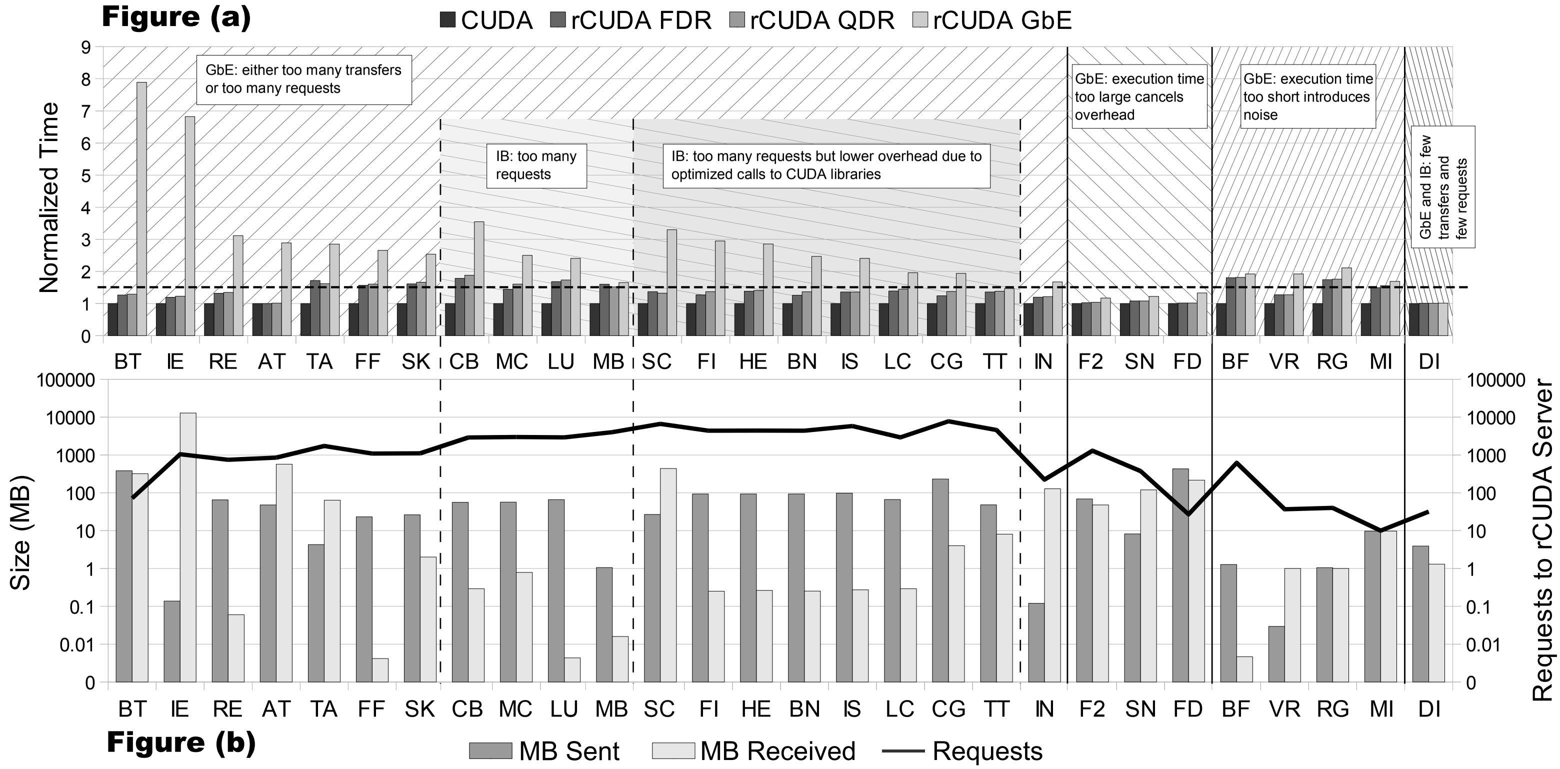


FIGURE 4.5: (a) CUDA SDK samples normalized execution time using CUDA and rCUDA over different networks. (b) rCUDA profiling measurements for CUDA SDK samples. Primary Y-axis shows MB sent/received by samples to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server. Both Y-axes in logarithmic scale.

between the rCUDA client and server (to take into account bandwidth), and the number of requests sent from the rCUDA client to the rCUDA server (to consider latency). The results for this experiment are displayed in Figure 4.5b.

Comparing rCUDA over GbE, we observe in these figures that most of the samples that exhibit a poor behavior (i.e., normalized time greater than or equal to 1.5) whether exceed 100MB of transfers (sent or received) or 1,000 requests to the rCUDA server. This is the case of 19 out of the 28 samples, namely `bandwidthTest` (BT), `simpleStreams` (IE), `alignedTypes` (AT), TA, FF, `smokeParticles` (SK), `simpleCUBLAS` (CB), `matrixMulCUBLAS` (MC), `cdpLUDecomposition` (LU), `Mandelbrot` (MB), `scan` (SC), `freeImageInteropNPP` (FI), `histEqualizationNPP` (HE), BN, `imageSegmentationNPP` (IS), `simpleDevLibCUBLAS` (LC), `conjugateGradient` (CG), `segmentationTreeThrust` (TT), and `interval` (IN).

Although sample `reduction` (RE) does not exceed these thresholds, it is close to both of them, 65MB of data and 749 requests sent to the rCUDA server, which explains the overhead. Samples `sortingNetworks` (SN) and `FDTD3d` (FD) transfer more than 100MB, but in these cases the overhead of using GbE is canceled by the long execution time of the samples, 9.93 and 17.47 seconds, respectively. For the same reason, sample `convolutionFFT2D` (F2), which takes almost 6 seconds, is not much affected by its large number of requests (over 1,300). On the contrary, samples `bilateralFilter` (BF), VR, `recursiveGaussian` (RG), and `simpleMPI` (MI) are too short, less than 1 second, which distort their overheads, in spite of performing neither more than 100MB of transfers nor 1,000 requests.

Concerning rCUDA over InfiniBand, the figures reveal that the samples performing worst are TA, FF, SK, CB, MC, LU, MB, BF, RG, and MI. The reason for some of them (CB, MC, LU, and MB) is a large amount of requests to the rCUDA server (over 2,800). The execution time of the rest of samples is too short, less than 1 second, which distort their overheads, despite not going beyond this limit. Samples SC, FI, HE, BN, IS, LC, CG, and TT also have a large amount of requests, but these samples use CUDA libraries [48] and the vast majority of the requests are originated by the load of these libraries, which are optimized by rCUDA, resulting in a lower overhead. Considering the amount of bytes transferred in these samples, they are insufficient to penalize rCUDA over InfiniBand. Sample IE is the only one that transfers a considerable amount of data, over 12GB, but the network overhead is hidden by the long duration of the sample, nearly 20 seconds.

When comparing the different networking technologies, almost all the samples perform as expected: rCUDA over InfiniBand always runs faster than rCUDA over GbE. Sample `stereoDisparity` (DI) takes similar time to rCUDA over the different networks because it performs very few transfers and very few requests to the remote server. These two

factors, added to its long duration, nearly 50 seconds, hide the network overhead. Regarding rCUDA over InfiniBand, the FDR InfiniBand executions are, in general, faster than the QDR InfiniBand runs, except for samples TA, MB, SC, and IS. These cases perform few transfers and a large number of requests to the rCUDA server, 1,729, 4,020, 6,669, and 6,048, respectively. This allows them to benefit from the slightly lower latency of QDR.

In conclusion, although the samples analyzed in this section are simple and usually involve few transfers and requests to the rCUDA server, they reveal the following insights:

- For rCUDA over GbE, transfers over 100MB and requests from 1,000 up drive to bad performance, because of the low bandwidth and high latency of this interconnect.
- For rCUDA over InfiniBand, these samples do not present enough transfers to arise InfiniBand throughput constraints. However, in terms of latency, samples with 2,800 or more requests start showing higher overheads.
- Compared to FDR InfiniBand, rCUDA over QDR InfiniBand benefits from its lower latency when samples imply thousands of requests to the rCUDA server. Nevertheless, in general, FDR InfiniBand takes advantage of its higher bandwidth, overcoming QDR InfiniBand in 87.5% of the studied samples.
- The actual execution time of the samples introduce a considerable noise in this study and modify the thresholds mentioned above concerning transfers and requests. Thus, longer samples minimize the impact of these limits, while shorter ones maximize it.

4.2.3 Influence of FDR InfiniBand on Production Applications

In order to study more thoroughly the influence of the network on remote GPU virtualization, in this section we analyze some production codes selected from the NVIDIA Popular GPU-Accelerated Applications [49].

4.2.3.1 CUDASW++

CUDASW++ [50] 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. In particular, we have used its last release,

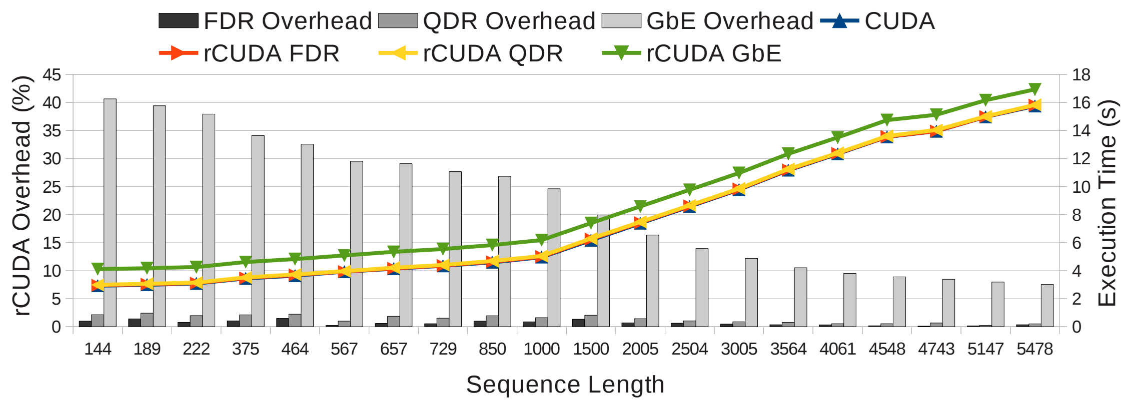


FIGURE 4.6: CUDASW++ execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA's overhead and secondary Y-axis execution time.

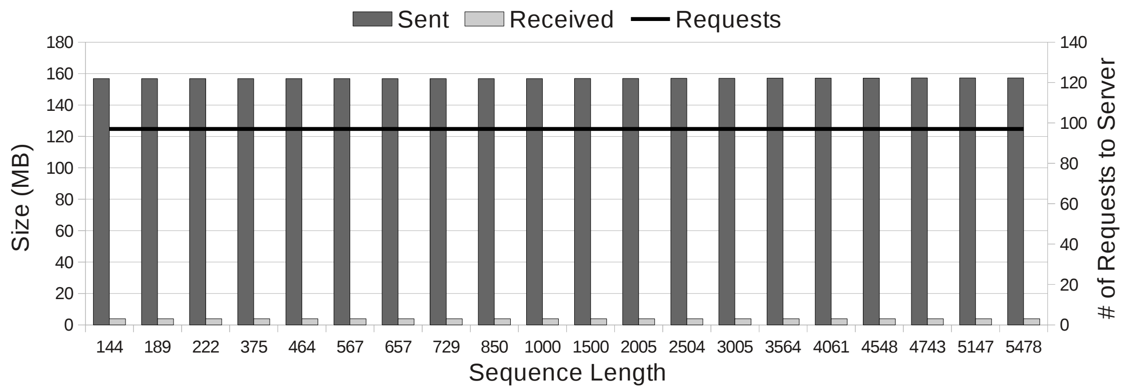


FIGURE 4.7: rCUDA profiling measurements for CUDASW++ executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by CUDASW++ to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

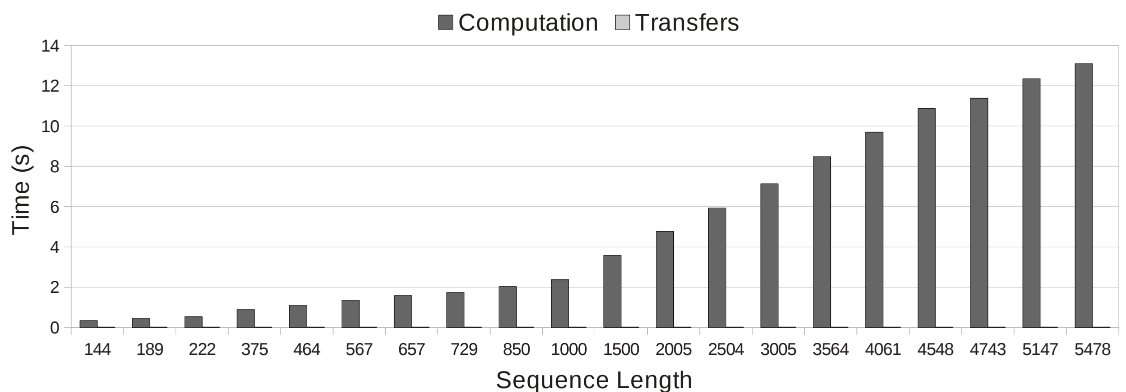


FIGURE 4.8: NVIDIA profiling result for CUDASW++ executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy).

version 3.0, for our study, along with the *Latest Swiss-Prot database* and the example query sequences available in the application's website: `http://cudasw.sourceforge.net`.

Figure 4.6 shows CUDASW++ execution time for queries of different sequence lengths using CUDA and rCUDA over different networks. The average of 10 repetitions is presented, and the maximum RSD observed was 0.019 for a sequence of length 222 when executed with CUDA, 0.010 for a sequence of length 464 in the case of rCUDA over FDR InfiniBand, 0.010 for a sequence of length 144 with rCUDA over QDR InfiniBand, and 0.009 for a sequence of length 657 with rCUDA over GbE. The figure also presents the overhead of using rCUDA: over InfiniBand the execution time is very close to that of CUDA. FDR InfiniBand and QDR InfiniBand introduce average overheads of 0.67% and 1.37%, respectively. For rCUDA over GbE, the average overhead is significantly higher though (21.88%).

The reason for rCUDA over InfiniBand performing only slightly worse than the native CUDA is the small number of transfers and the reduced number of requests done by the application to the rCUDA server (see Figure 4.7). For GbE, this small transfer size (around 160MB) is enough to penalize rCUDA because of the low bandwidth of this technology. With respect to the different InfiniBand networks, QDR InfiniBand presents an average overhead 0.7% higher than FDR InfiniBand.

We can also observe that longer query sequences reduce the overhead introduced by rCUDA. Figure 4.8 reveals that this is due to the fact that the time spent in transfers (i.e., time spent in memory copies between host memory and the device memory, also referred to as *CUDA memcopy*) remains always the same for all the query lengths, but the time employed by computations (i.e., time employed by CUDA kernels) increases with the query sequence length. Performing more computations helps rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of rCUDA due to the transfers across the network.

4.2.3.2 GPU-BLAST

GPU-BLAST [51] 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. It is integrated into the NCBI-BLAST code and produces identical results. We utilize release 1.1 in the next experiments, where we have followed the installation instructions for (1) sorting a database, and (2) creating a GPU database.

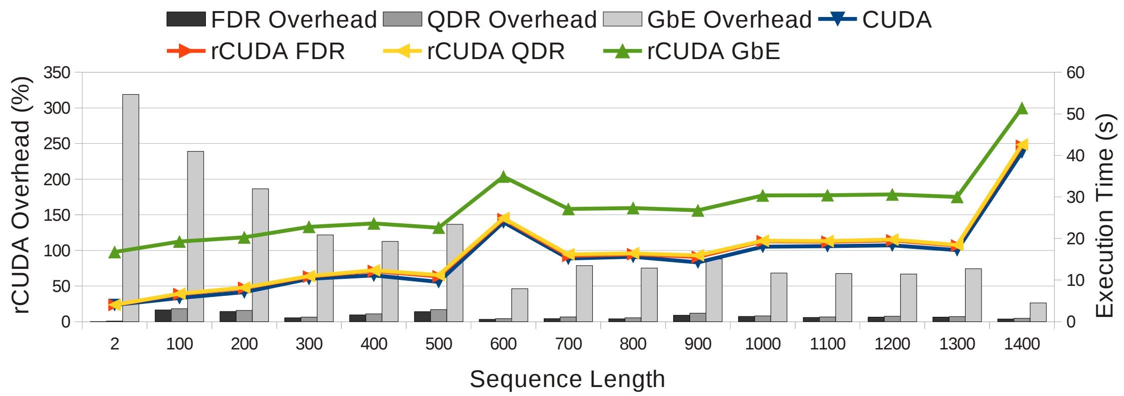


Figure 4.9: GPU-BLAST execution time for queries of different sequence lengths, using CUDA and rCUDA over different networks. Primary Y-axis shows rCUDA's overhead, while secondary Y-axis represents execution time.

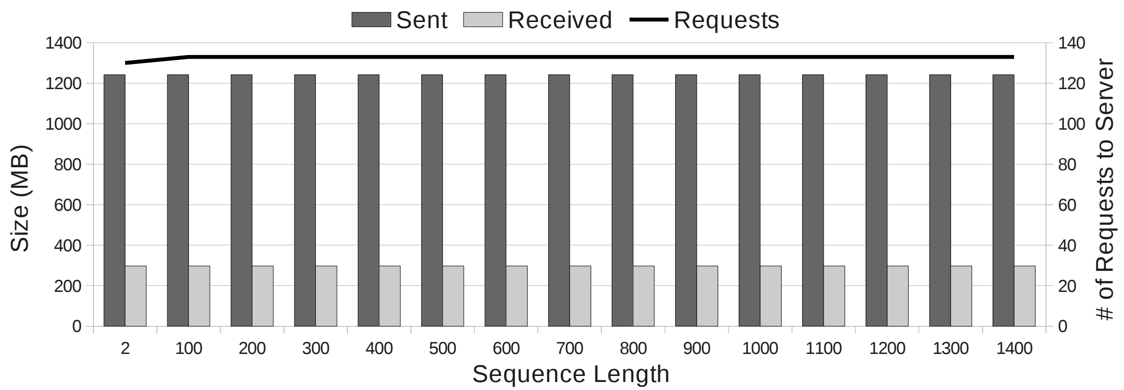


Figure 4.10: rCUDA profiling measurements for GPU-BLAST executing queries of different sequence lengths. Primary Y-axis shows MB sent/received by GPU-BLAST to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

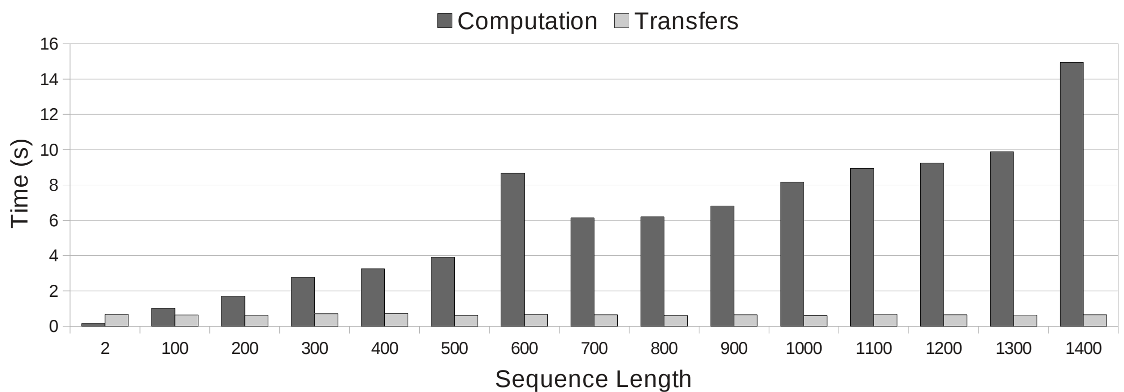


Figure 4.11: NVIDIA profiling result for GPU-BLAST executing queries of different sequence lengths. In particular, time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy).

We then use the query sequences which come with the application package to search the database.

Figure 4.9 depicts the GPU-BLAST execution time for queries of different sequence lengths using CUDA and rCUDA over the three networks. The average of 10 repetitions is presented, and the maximum RSD observed was 0.207 for a sequence of length 100 when executed with CUDA, 0.031 for a sequence of length 200 in the case of rCUDA over FDR InfiniBand, 0.051 for a sequence of length 700 with rCUDA over QDR InfiniBand, and 0.012 for a sequence of length 1400 with rCUDA over GbE. From our results we also extract that the average overhead of using rCUDA is 7.07%, 8.63%, and 113.71% for FDR InfiniBand, QDR InfiniBand, and GbE, respectively. Data transfers over 1.2GB (see Figure 4.10) hurt performance for rCUDA over GbE. Concerning rCUDA over InfiniBand, QDR InfiniBand presents an average overhead 1.56% higher than FDR InfiniBand.

As it was the case for CUDASW++, Figure 4.11 illustrates that the time spent in transfers is constant for all the queries with GPU-BLAST and only the time required by computations varies. Again, we can observe that rCUDA's overhead decreases as computation time increases. Figure 4.11 also reveals a peak in time spent in GPU computations when running GPU-BLAST with a sequence of length 600, which explains a similar peak in Figure 4.9 for this sequence length.

4.2.3.3 LAMMPS

LAMMPS [40] is a classic molecular dynamics simulator which can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For the tests below, we use the release from Feb. 19, 2013, and benchmarks `in.eam` and `in.lj` installed with the application. We run the benchmarks with one processor and scaling by a factor of 5 in all three dimensions (i.e., a problem size of 4 million atoms).

Table 4.2 reports the execution time for these benchmarks using CUDA and rCUDA over the three networks, and Table 4.3 shows rCUDA's overhead for the same tests. The average of 10 executions is presented, and the maximum RSD observed was 0.013 for benchmark `in.lj`, when executed with rCUDA over GbE. 0.005 for benchmark `in.eam` in the case of rCUDA over FDR InfiniBand, 0.009 for benchmark `in.eam` with rCUDA over QDR InfiniBand, and 0.004 for benchmark `in.lj` with CUDA. Once again, rCUDA over GbE exhibits a poor performance due to the large transfers involved and the huge number of requests sent to the rCUDA server, as shown in Figure 4.12. For rCUDA over InfiniBand, QDR InfiniBand has an average overhead 2.39% higher than FDR

Table 4.2: LAMMPS execution time for benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions.

LAMMPS	Execution time (s)			
benchmark	CUDA	rCUDA FDR	rCUDA QDR	rCUDA GbE
in.eam	52.33	56.36	57.60	102.09
in.lj	36.39	38.02	38.90	79.37

Table 4.3: rCUDA overhead for the benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions.

LAMMPS	Overhead (%)		
benchmark	rCUDA FDR	rCUDA QDR	rCUDA GbE
in.eam	7.71	10.07	95.10
in.lj	4.50	6.90	118.12

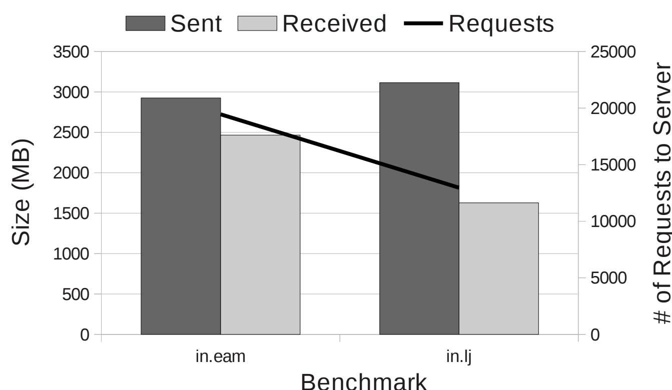


Figure 4.12: rCUDA profiling measurements for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Primary Y-axis shows MB sent/received by LAMMPS to/from server when using the rCUDA framework. Secondary Y-axis presents requests sent to the rCUDA server.

InfiniBand. Despite the large amount of requests, which could help QDR InfiniBand in terms of latency, these experiments reveal that FDR InfiniBand better bandwidth has more influence than its worse latency when data transfers are of a considerable size, as also pointed out in previous sections.

In this application, the benchmark presenting greater overhead, `in.eam`, spends also significantly more time in computations (see Figure 4.13) than benchmark `in.lj`, which shows a lower overhead. Apparently, this behavior does not obey the conclusions obtained in previous sections with respect to the time employed by computations and rCUDA's overhead. The explanation lies in the fact that here, unlike in preceding experiments, the number of bytes sent is almost the same for both benchmarks, but the number of bytes received and requests to the server are significantly higher for `in.eam`, thus making a higher use of the network fabric than `in.lj`.

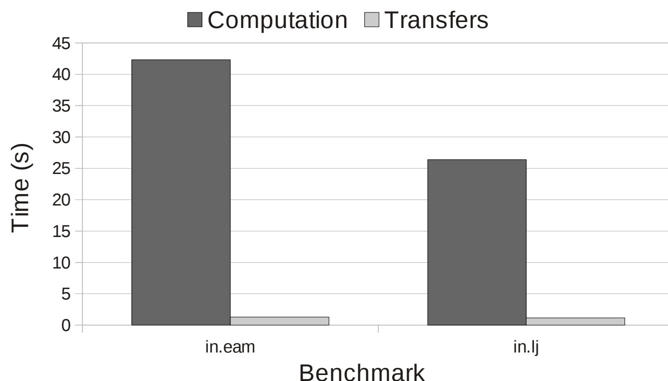


FIGURE 4.13: NVIDIA profiling result for LAMMPS executing benchmarks in.eam and in.lj, scaled by a factor of 5 in all three dimensions. Time employed by computations (i.e., CUDA kernels) and by memory transfers (i.e., CUDA memcopy) is shown.

4.2.4 Summary

In this section we have analyzed how the bandwidth matching between PCIe 2.0 and FDR InfiniBand influences the performance of remote GPU virtualization, using both synthetic tests and real GPU-accelerated applications. Synthetic tests have revealed that FDR InfiniBand achieves a substantial gain (over 40% with respect to the previous QDR InfiniBand version) in terms of bandwidth to/from the remote GPU. This bandwidth gain, when incorporated into the context of a GPU-accelerated application, which performs computations in addition to transfers between main memory and GPU memory, reduces overhead up to 2.39% with respect to QDR InfiniBand, clearly showing that the new interconnect not only serves traditional applications running in the cluster but also remotely accelerated ones. This can be seen in Table 4.4, which summarizes the most important results for the analyzed applications. We have analyzed applications with low, medium, and high volumes of data transfers. This analysis reveals that rCUDA over GbE has a high overhead independently of the amount of transfers. In contrast, rCUDA over InfiniBand exploits the much higher throughput than GbE, exposing a very small overhead for applications with a low amount of transfers. For applications that involve a moderate to high volume of transfers, the overhead of using rCUDA over InfiniBand depends on the time spent in computations. Thus, if the amount of computations is enough to compensate for the extra time spent transferring data across the network, then the overhead of rCUDA over InfiniBand is very low. Otherwise, the overhead becomes significant.

Concerning the two different InfiniBand versions examined in these tests, QDR InfiniBand and FDR InfiniBand, it appears that as the size of transfers increases, FDR InfiniBand higher throughput is more important. In this way, for the applications considered in this study, QDR InfiniBand average overhead in comparison to FDR

Table 4.4: Summary of rCUDA overhead using different networks, related with average number of rCUDA transfers and requests to rCUDA server, for the studied applications.

Application	rCUDA transfers	rCUDA requests	rCUDA overhead (%) FDR	QDR	GbE
CUDASW++	~160MB	~100	0.67	1.37	21.88
GPU-BLAST	~1200MB	~130	7.07	8.63	113.71
LAMMPS	~3000MB	~16000	6.10	8.49	106.61

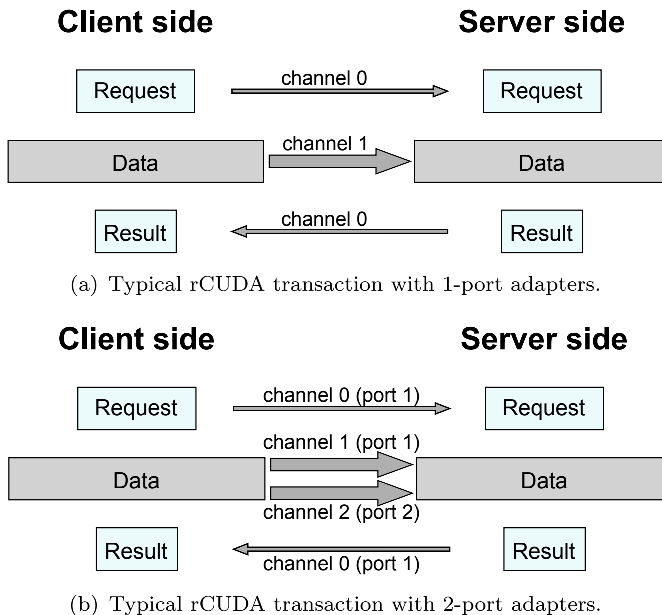
InfiniBand grows together with the level of transfers: 0.7%, 1.56%, and 2.39%, for low, medium, and high volumes of transfers, respectively.

4.3 Enhancing rCUDA with Support for InfiniBand Dual-port Adapters

As previously commented, in order to minimize the performance difference between the traditional local use of GPUs and the remote virtualized access, the external network fabric should offer a throughput similar to that of the internal PCIe link. Although the differences in performance of these interconnects were notable in the past, recent advances in networking technologies have considerably reduced the performance gap. For example, the theoretical throughput of the latest available versions of PCIe and InfiniBand are relatively close: 15.75 GB/s for PCIe 3.0 x16 vs. 12.5 GB/s for FDR InfiniBand using Mellanox Connect-IB dual-port adapters [52]. Furthermore, the effective bandwidth figures of these technologies also need to be considered, as shown later in this section. The small difference in performance between the InfiniBand Connect-IB and PCIe 3.0 technologies motivates the work presented in this section, where we analyze how their combined use reduces in practice the performance penalty of remote GPU virtualization solutions.

In this section we present the version 15.07 of rCUDA (see Table 2.2 for more details). This version is enhanced with support for InfiniBand dual-port adapters, such as the InfiniBand Connect-IB (FDR x 2) ones. We also analyze how the use of these network adapters (with performance similar to that of PCIe 3.0) influence the overhead of rCUDA.

The rest of the section is organized as follows. Section 4.3.1 details how rCUDA has been extended with support for dual-port network cards. Section 4.3.2 studies the impact of Connect-IB network adapters on the bandwidth and latency figures attained by rCUDA. Finally, Section 4.3.3 summarizes the main conclusions of this section.



(a) Typical rCUDA transaction with 1-port adapters.

(b) Typical rCUDA transaction with 2-port adapters.

FIGURE 4.14: Scheme of the use of dual-port network adapters. (a) CUDA requests and responses, as well as data transfers, make use of the single available network port. (b) CUDA requests and responses to/from the remote server employ a single logic channel across one of the ports of the network card. Data transfers use two different logic channels across both ports, thus doubling the attained bandwidth for bulk transfers.

4.3.1 Adding Dual-port Support to rCUDA

For the work presented in this section, we have improved rCUDA with support for dual-port network cards. This allows a single data transfer to exploit the aggregated bandwidth provided by both ports in order to achieve an effective bandwidth close to 100 Gbps. This high bandwidth is attained by appropriately splitting, inside the InfiniBand communication module of rCUDA, the transmission of bulk transfers into smaller data chunks that are concurrently forwarded across both ports of the network adapter.

Figure 4.14 depicts an scheme of the use of dual-port adapters. Notice that including a second port in the data transfers is not just a matter of duplicating the original pipeline, given that both pipelines must share work and also remain synchronized, thus turning the adaptation process to dual-port adapters noticeably more complex.

4.3.2 Impact of Connect-IB on Remote GPU Usage

When making use of remote GPU virtualization, the latency and bandwidth properties of the communication path between main memory, in the client node, and GPU memory, in the remote server, greatly influence the performance of data transfers between them.

In this regard, latency is crucial when the size of the data being copied is small whereas for large data copies bandwidth is the most important concern. In this section we present how the improved features of the Connect-IB network adapter, as well as the use of PCIe 3.0 enabled GPUs, influence the basic performance characteristics of remote GPU virtualization techniques.

In order to do so, in next subsection we compare the exact bandwidth and latency of Connect-IB and PCIe 3.0. We then study how these bandwidth and latency features affect the performance and internal design of remote GPU virtualization solutions. At the end of this section we present how rCUDA has been internally improved to extract as much bandwidth as possible from the underlying interconnect.

4.3.2.1 From Theoretical to Real Bandwidth and Latency Figures

The bandwidth and latency of the interconnects involved in remote GPU virtualization greatly affect the performance of these solutions. Basically, the interconnects to consider in this context are, on the one hand, the network fabric connecting the client and server computers and, on the other, the PCIe links that network interfaces and GPUs are attached to. In general, the closer the performance of these interconnects are, the lower the overhead that will be experienced when making use of remote GPUs.

As mentioned in Section 4.3, the theoretical bandwidth of Connect-IB and PCIe 3.0 are relatively close. Therefore, one may expect that the performance penalty of remote GPU virtualization mechanisms such as rCUDA will be quite reduced with respect to the traditional (i.e., local) use of GPUs. It is well known, however, that the actual performance of any interconnection technology largely depends on the ability of the upper software layers using such interconnect to obtain as much bandwidth as possible from it. Thus, the effective bandwidth and latency features of a given interconnect are usually perceptibly lower than the theoretical ones. The actual concern, therefore, is not about theoretical performance but about real numbers, that is, whether real bandwidth numbers help to reduce the performance difference among the external InfiniBand fabric and the internal PCIe or, on the contrary, make that difference larger, thus increasing the penalty of remote GPU virtualization techniques.

To address this concern, in this subsection we compare the FDR InfiniBand fabric using the dual-port Connect-IB network adapter against the PCIe 3.0 link, when used by their respective upper software layers: the network driver in the case of InfiniBand and the CUDA driver, along with the NVIDIA Tesla K40 GPU, in the case of the PCIe

3.0[1]. This analysis will expose the maximum performance that might be achieved by a remote GPU virtualization solution and, additionally, will put the performance of such a virtualization solution into the right perspective.

In order to perform such a comparison, a testbed was configured using two 1027GR-TRF Supermicro servers, each with the following characteristics:

- Two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz.
- 32 GB of DDR3 SDRAM memory at 1,600 MHz.
- 1 Mellanox ConnectX-3 single-port InfiniBand adapter (PCIe 3.0 x8).
- 1 Mellanox Connect-IB dual-port InfiniBand adapter (PCIe 3.0 x16).
- CentOS 6.4 with Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools) and CUDA 7.0 with NVIDIA driver 340.46.

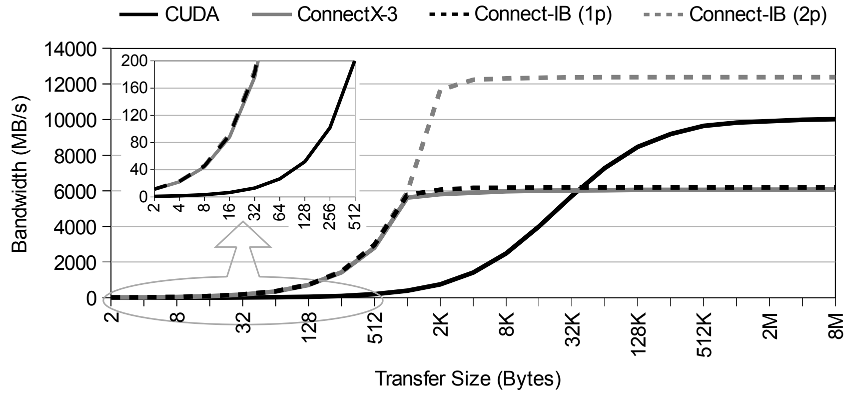
Additionally, one of the nodes had an NVIDIA Tesla K40 GPU (which makes use of a PCIe 3.0 x16 link). Furthermore, both nodes were interconnected by a Mellanox Switch SX6025 (FDR compatible).

The test servers are NUMA machines and NUMA effects matter for rCUDA. For this reason, the InfiniBand adapters and the NVIDIA GPU were connected to the same NUMA node (i.e., node 0). For all the experiments shown in this section, processes and memory buffers are bound to this node.

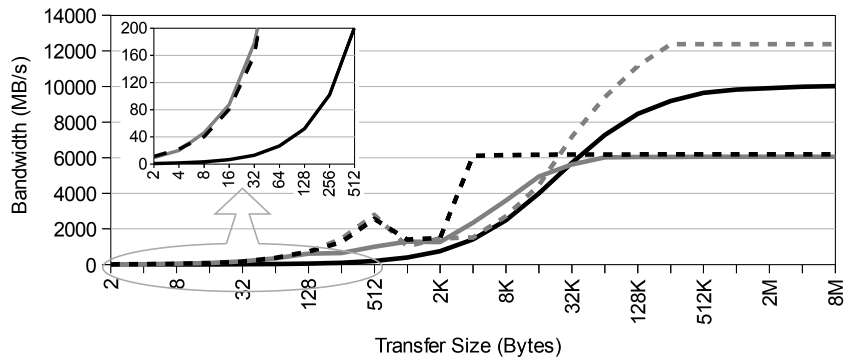
Figure 4.15 shows a comparison of the actual bandwidth achieved by PCIe 3.0 and InfiniBand. For PCIe 3.0, the `bandwidthTest` benchmark from the NVIDIA CUDA Samples [53] was used to transfer data from main memory to the Tesla K40 GPU located in the same server. Pinned memory was used. For InfiniBand, the `ib_write_bw`, `ib_read_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED software distribution were used. Figure 4.15(a) presents results when messages are sent using the memory semantics with RDMA write whereas Figure 4.15(b) shows results for the channel semantics (i.e., send/receive not using RDMA). Results for the RDMA read operation are omitted given their similarity to the ones provided by the `ib_write_bw` benchmark.

From the data in Figure 4.15 we can do two different sets of comparisons. First, we can see that the Connect-IB technology, when used with a single port—labeled "Connect-IB (1p)"—achieves a slightly higher bandwidth than does the previous ConnectX-3 version

[1]Notice that the NVIDIA Tesla K80 GPU was already available when this study was carried out. However, for a single bulk transfer the new GPU attains lower bandwidth than the K40 GPU, what has motivated to use the K40 GPU in this study.



(a) InfiniBand RDMA write bandwidth.



(b) InfiniBand send bandwidth.

FIGURE 4.15: InfiniBand bandwidth test using FDR over different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). The CUDA line shows bandwidth attained when copying data from main memory to the memory of the K40 GPU, located within the same node.

[54] of InfiniBand (6,191 MB/s vs. 6,041 MB/s). One may believe that the reason for this higher bandwidth rests with the PCIe connector of the Connect-IB network adapter, which has extended the previous PCIe 3.0 x8 connector, used by ConnectX-3, to a wider PCIe 3.0 x16 one (used by the Connect-IB adapter), thus moving from a theoretical bandwidth of 7.87 GB/s (PCIe 3.0 x8) to 15.75 GB/s (PCIe 3.0 x16) and, therefore, allowing the Connect-IB card to be fed with data from main memory at full speed. However, if this were the reason, then when making use of both ports of the Connect-IB card, the achieved bandwidth (12,382 MB/s) would not double the bandwidth with one port but would be, at most, just twice the bandwidth attained by the previous single port ConnectX-3 card, thus being lower than the actual value achieved in our experiments. Therefore, the reason for the improved bandwidth may not be the wider PCIe 3.0 connector but the enhanced internal architecture that the Connect-IB adapter features.

The second comparison that can be made from the data in Figure 4.15 is between the performance of InfiniBand and that of PCIe 3.0 when the latter is being used by CUDA. In this case, the theoretical 15.75 GB/s of PCIe 3.0 x16 is reduced to 10 GB/s because

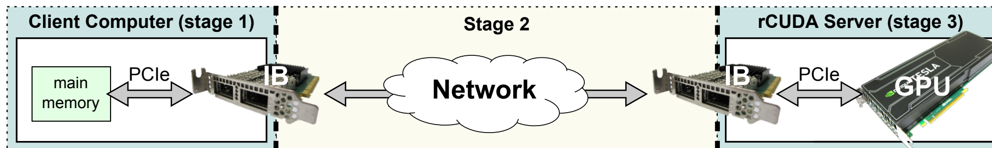


FIGURE 4.16: Three-stage pipeline modeling the client-server data transfer process within a remote GPU virtualization framework.

of the overhead introduced by the software layers, as well as the unavoidable bandwidth losses of the hardware itself. This large performance drop was not expected in this study, given that the previous version of the NVIDIA GPU, the Tesla K20, that made use of a PCIe 2.0 x16 link, achieved a maximum bandwidth of approximately 6 GB/s[2] out of the theoretical 8 GB/s bandwidth of PCIe 2.0, thus introducing a loss of 25% of the available bandwidth, in contrast to the 36.5% loss introduced by the NVIDIA Tesla K40 GPU. Nevertheless, the unexpectedly large performance loss introduced by the CUDA software provides a surprising conclusion: for the first time, the effective bandwidth of the external interconnect (FDR InfiniBand + dual-port Connect-IB adapters) is noticeably larger than the maximum bandwidth attained by the internal interconnect (PCI 3.0 x16 + CUDA). In particular, the external bandwidth is now 23% higher than the internal one. This fact, along with the earlier bandwidth slope of InfiniBand, introduces new possibilities for remote GPU virtualization solutions, which so far were strongly limited by the lower performance of the external interconnect.

However, bandwidth is not the only issue to consider when leveraging remote GPUs: latency also matters. Actually, the process of transferring data from main memory in the source computer to the GPU memory in the target node can be seen as a pipeline composed of three stages (see Figure 4.16): the PCIe link from main memory to the source network interface (stage 1); the network fabric connecting the source and the destination network cards (stage 2); and the PCIe link at the target node connecting both the network adapter and the GPU (stage 3). Since bandwidth in the intermediate stage is no longer the bottleneck and, therefore, all the stages in the pipeline are equalized, the factor that will contribute most to reduce performance when using remote GPUs, with respect to their local use, is the increased latency to reach the GPU. While in the local case only one stage must be traversed, in the remote case data must travel across three stages, with the first and last ones being similar to the one in the local case (the PCIe link), whereas the intermediate one (the network fabric) can be expected to present a noticeably larger latency. Nevertheless, the software layers are also expected to play an important role in the case of latency. In this case, not only are the network and GPU software layers involved, but also the remote GPU virtualization framework should be taken into consideration. For the sake of simplicity, in this section we study only the

[2] This bandwidth was attained for data copies from host to device using pinned host memory, using a similar testbed system.

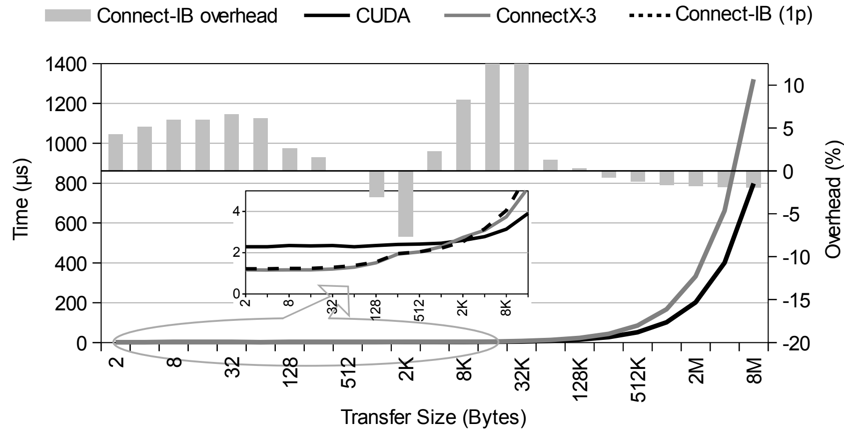


FIGURE 4.17: InfiniBand RDMA write latency test using FDR over ConnectX-3 and Connect-IB with 1 port. The CUDA line shows latency for copying data from main memory to the memory of a local Tesla K40 GPU (i.e. just using the local PCIe link).

latency for InfiniBand and CUDA; we postpone the analysis of the complete picture, taking into account the remote virtualization software layer, until Section 4.3.2.2.

To assess the latency of the Connect-IB and PCIe 3.0 + CUDA interconnects, in Figure 4.17 we compare the latency of the previous ConnectX-3 adapter and the Connect-IB when one port is leveraged (notice that, unlike the tools used to measure bandwidth, the tools to measure latency within the OFED make use only of a single port). Numbers for the latter are presented as overhead (positive or negative) with respect to the ConnectX-3 version. Latency data for CUDA (Tesla K40) are also depicted. Figure 4.17 shows latency numbers for the RDMA write case. Similar numbers were obtained for the RDMA read and non-RDMA send operations.

In general, Figure 4.17 reveals that latency for the Connect-IB card is slightly increased: latency for small data transfers increases about 5% with respect to the latency reported for the previous ConnectX-3 version. Furthermore, latency for InfiniBand is lower than for CUDA for transfer sizes up to 4 KB, where the higher bandwidth of CUDA (remember that Figure 4.17 depicts latency numbers for Connect-IB with one port) causes that latency for larger transfers with CUDA becomes lower than with Connect-IB with 1 port.

4.3.2.2 Influence on the Bandwidth of rCUDA

Next we investigate how the improved characteristics of Connect-IB and PCIe 3.0 + CUDA influence the performance of the rCUDA. As the baseline reference, we use the performance of CUDA, given that minimizing the overhead over CUDA performance is the goal of any remote GPU virtualization framework. Furthermore, we consider the

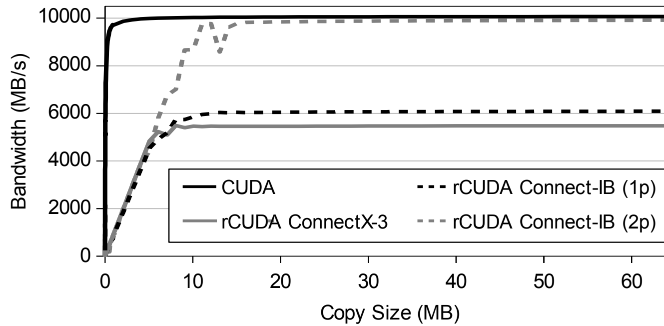


FIGURE 4.18: Bandwidth test for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports).

performance of the previous InfiniBand generation, the ConnectX-3 network adapter, so that the benefits of the Connect-IB card are clearly exposed.

Figure 4.18 shows that the Connect-IB card, when using a single port, improves rCUDA performance over the previous ConnectX-3 adapter for copies from host to device with pinned host memory. Furthermore, when two ports are used, the maximum bandwidth attained by rCUDA is almost the same as that achieved by CUDA (98.5% of CUDA's bandwidth). However, despite the faster slope of InfiniBand with respect to CUDA reported in Figure 4.15, rCUDA presents a noticeably lower bandwidth than does CUDA for small/medium copy sizes (up to 10 MB). This is interesting because the presence of this delayed slope was much less noticeable in rCUDA for the slower ConnectX-3 and PCIe 2.0 + CUDA, but it is now exacerbated by the increased bandwidth of the Connect-IB adapter. Actually, notice that rCUDA with ConnectX-3 presents the same slope as rCUDA with Connect-IB. In this regard, when using ConnectX-3, rCUDA reaches 80% of the maximum rCUDA bandwidth for a transfer size equal to 6 MB, whereas in the case for Connect-IB, reaching 80% of the maximum rCUDA bandwidth is delayed until transfer size equals 9.5 MB. Similar conclusions were obtained for the device-to-host direction. Hence, given that the performance of InfiniBand is not causing this slower slope, the internal design of rCUDA must be improved in order to increase the performance for small/medium transfer sizes. This issue will be further analyzed in Section 4.3.2.3. Before that, we complete the analysis of the performance of rCUDA taking a look at latency.

Regarding latency, Figure 4.19 depicts the latency attained by rCUDA for the different network adapters. Again, one can conclude from the figure that rCUDA presents an overhead that needs to be further investigated. This concern will also appear in later sections.

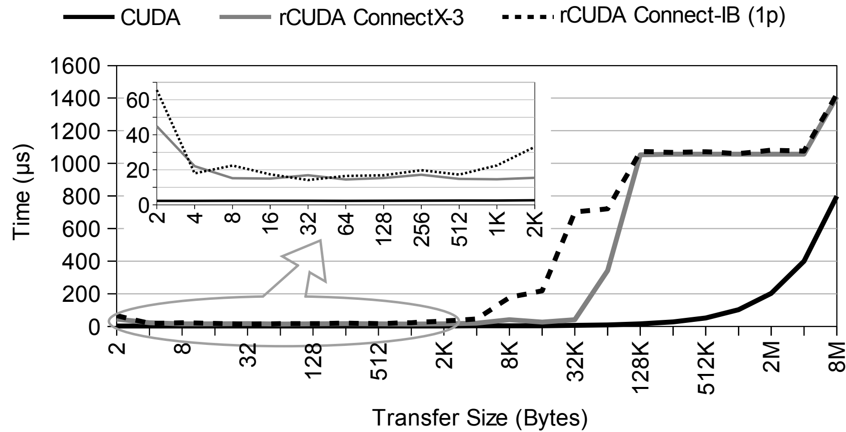
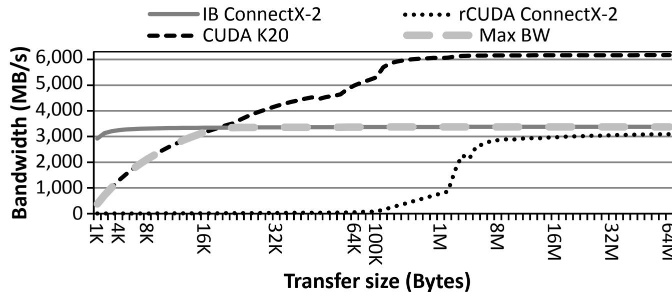


FIGURE 4.19: Latency for copies from host to device with pinned host memory, using CUDA and the rCUDA framework over InfiniBand employing ConnectX-3 and Connect-IB with 1 port.

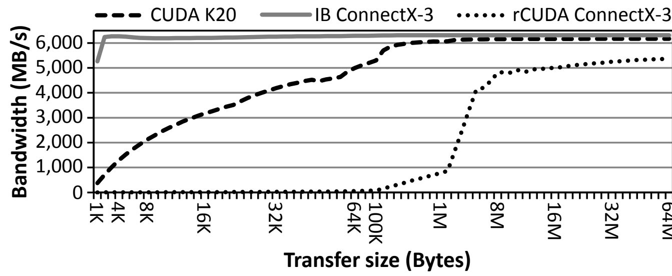
4.3.2.3 Analyzing the Bandwidth of rCUDA

It has been shown in previous section that rCUDA presents low performance for short and medium sized messages up to, approximately, 10 MB. In order to further investigate the reasons for this poor bandwidth, we have comparatively studied the bandwidth attained by rCUDA for three different hardware generations: InfiniBand ConnectX-2 (QDR, 40 Gbps), InfiniBand ConnectX-3 (FDR, 56 Gbps), and InfiniBand Connect-IB (more than 100 Gbps). Figure 4.20 presents such analysis. Figure 4.20(a) depicts the case of ConnectX-2, showing that the curves for the performance of InfiniBand and for CUDA cross each other at 18 KB. In this way, for small messages up to 18 KB, the maximum performance attained by rCUDA will be limited by the bandwidth achieved by the Tesla K20 GPU whereas for larger transfer sizes maximum performance will be limited by the bandwidth of InfiniBand. This mixed boundary is depicted in Figure 4.20(a) by the curve *"Max BW"*. Anyway, the figure shows that the bandwidth achieved by rCUDA is very small up to 100 KB, where it quickly grows up to 3 GB/s, reaching 80% of this bandwidth for a transfer size equal to 4.9 MB.

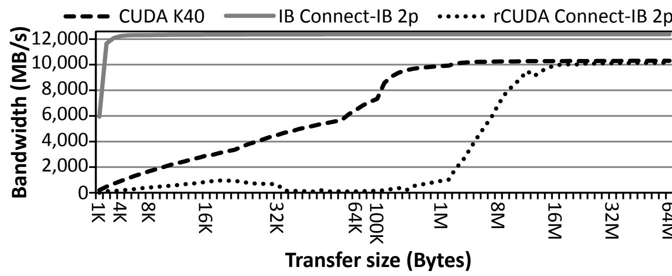
Figures 4.20(b) and 4.20(c) present similar conclusions, although in these cases the bandwidth attained by rCUDA is limited by the performance of the Tesla K20 and K40 GPUs, respectively, which achieve lower performance than their associated InfiniBand counterpart (InfiniBand ConnectX-3 and Connect-IB, respectively). In the same way as for InfiniBand ConnectX-2, it can be seen in Figures 4.20(b) and 4.20(c) that 80% of maximum rCUDA bandwidth is obtained for a transfer size equal to 6 MB (ConnectX-3 + K20) and 9.5 MB (Connect-IB + K40), as mentioned before. One can see how the 80% threshold is increasingly reached at larger transfer sizes as the InfiniBand



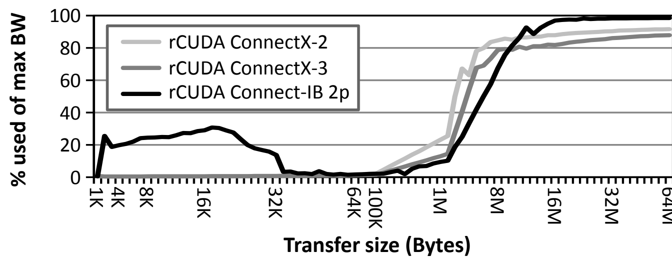
(a) InfiniBand ConnectX-2 and NVIDIA Tesla K20.



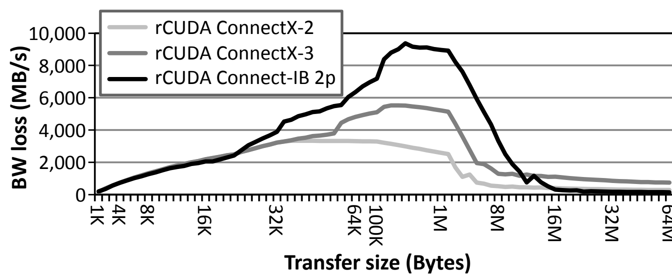
(b) InfiniBand ConnectX-3 and NVIDIA Tesla K20.



(c) InfiniBand Connect-IB dual port and NVIDIA Tesla K40.



(d) Percentage used of maximum available bandwidth.



(e) Bandwidth lost for each rCUDA generation.

Figure 4.20: Analysis of rCUDA with different hardware generations.

technology improves. This can also be seen in Figure 4.20(d), which compares the percentage used from the total available bandwidth for the three hardware generations. This figure clearly shows that reaching maximum performance is progressively delayed as the network fabric improves bandwidth. Finally, Figure 4.20(e) depicts the bandwidth loss for each hardware generation (from the maximum that could be achieved). Again, as new InfiniBand generations appear, the loss increases for transfers sizes below 10 MB, approximately. In summary, the internal implementation of communications within rCUDA needs to be fixed, given that rCUDA is not able to efficiently cope with the increasing bandwidth that InfiniBand + CUDA provide with each hardware generation. In order to address this important concern, in Section 4.4 we carry out a thorough search for the causes of the poor bandwidth attained by rCUDA for small and medium transfer sizes.

4.3.3 Summary

In this section we have explored how the bandwidth matching between the InfiniBand Connect-IB network adapter and the PCIe 3.0 influences the performance of the rCUDA remote GPU virtualization framework. We have concluded that the internal implementation of communications within rCUDA needs to be reviewed, in order to efficiently benefit from the increasing bandwidth that InfiniBand + CUDA provide with each new hardware generation.

4.4 InfiniBand Verbs Optimizations for Remote GPU Virtualization

InfiniBand is an interconnect providing high bandwidth and low latency, being commonly used in HPC. The high performance attained by InfiniBand makes that its use in supercomputers has considerably increased during the last years [55], as shown in Figure 4.21. This figure presents the amount of supercomputers in the TOP500 list [56] using the InfiniBand network as well as different versions of the Ethernet one. Actually, as it can be seen in the figure, the presence of the InfiniBand technology in current supercomputers is even higher than that of Ethernet, having the former a share of 44.8% whereas the latter presents a share of 37.6%. Furthermore, we can observe that the total sum represented by systems based on any of these two interconnect technologies accounts for more than 80% of the systems in this list, what reveals that the InfiniBand technology is the most widely one used in the HPC domain.

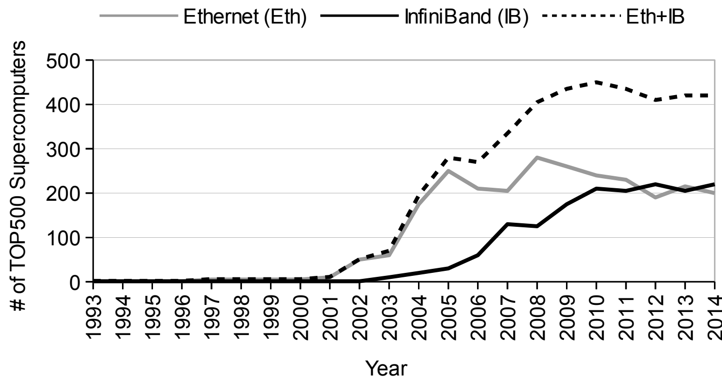


FIGURE 4.21: Presence of Ethernet and InfiniBand in the TOP500 list.

However, a major disadvantage of the InfiniBand network lies in the fact that its specification [57] does not clearly define an API that can be easily learned without attending specific courses. Indeed, it only describes a set of functions, usually referred to as *verbs* (i.e., the InfiniBand Verbs–IBV), which must be available in any commercial product adhering the specification. As a consequence, the lack of such explicit API in conjunction with the complexity of the IBV semantics make it difficult to develop even a simple program. This is evidenced by the publication of papers with the sole purpose of clarifying how to interact with the InfiniBand Verbs, such as [58], [59] or [60], to name only a few.

The net result is that InfiniBand is the most widely used interconnect in the supercomputers included in the TOP500 list, but the lack of recent documentation makes it difficult to get all the benefits from this network fabric. In this section we explore two general code optimizations that may be helpful when developing applications using InfiniBand Verbs. These optimizations achieve an average increase of up to 43% in the attained bandwidth, what is later translated into a reduction of up to 35% in execution time of applications using remote GPU virtualization frameworks.

The rest of the section is organized as follows. In Section 4.4.1, we discuss the work related to our study. Next, in Section 4.4.2, we present and analyze the code optimizations proposed in this section. Section 4.4.3 analyzes the benefits that these optimizations bring to remote GPU virtualization frameworks. Finally, in Section 4.4.4, the main conclusions of this work are presented.

4.4.1 Related Work

As commented before, the lack of an easy to understand programming API for InfiniBand is one of the major concerns when developing applications that use this network fabric.

Actually, this was what motivated G. Kerr to dissect in [58] a simple *pingpong* program, in an attempt to make clear how to interact with the InfiniBand Verbs API.

In view of this, exploring optimizations for InfiniBand applications has typically remained a big challenge. Several researchers have attempted to present recommendations for achieving optimal performance. One such example is the work by Liu et al. in [61], which presents a recent in-depth analysis of the FDR InfiniBand network, proposing several interesting optimizations. However, the study is limited to the memory semantics (i.e., Remote Direct Memory Access–RDMA), not addressing the channel semantics (i.e., send/receive verbs not using RDMA). Additionally, the tests are done using Sandy Bridge processors, while results over later generation processors (i.e., Ivy Bridge) could lead to different conclusions.

Other researchers have also presented improvements in recent studies. For instance, Subramoni et al. in [62] study the benefits of using the new Dynamically Connected (DC) InfiniBand transport protocol, showing great improvements for both synthetic benchmarks and production applications. Another such example can be found in the work by Wang et al. in [63], where it is proposed an optimized GPU to GPU communication design for InfiniBand clusters. Nevertheless, although these proposals improve performance, both of them are focused on optimizing MPI libraries, whose requirements differ from general applications.

From our point of view, all these previous efforts to optimize InfiniBand environments will benefit from the work presented in this section, as the improvements here exposed can be applied to all those fields.

4.4.2 Bandwidth Optimizations

In this section we introduce and analyze the optimizations proposed in this work. The setup used for the experiments reported in this section consists of two 1027GR-TRF Supermicro servers connected by an SX6025 InfiniBand Switch (FDR), each of the servers with the following characteristics:

- Two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz.
- 32 GB of DDR3 SDRAM memory at 1,600 MHz.
- 1 Mellanox ConnectX-3 single-port InfiniBand adapter.
- 1 NVIDIA Tesla K20m GPU.

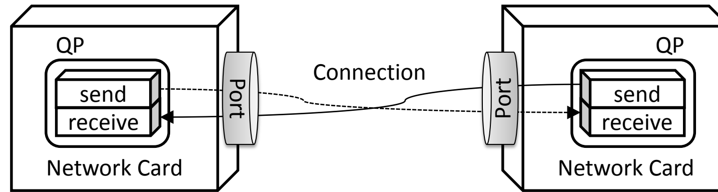


FIGURE 4.22: InfiniBand Queue Pair (QP) scheme.

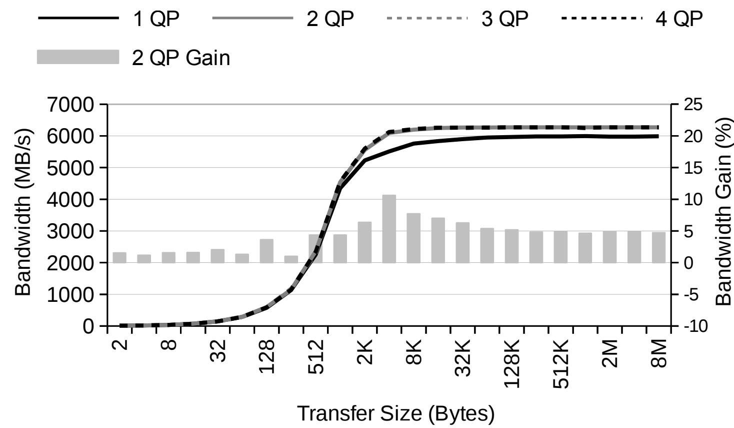
- CentOS 6.4 operating system with Mellanox OFED 2.3-2.0.0 (InfiniBand drivers and administrative tools) and CUDA 6.5 with NVIDIA driver 340.29.

The testbed servers are NUMA machines and therefore NUMA effects matter for the experiments shown in this section. For this reason, the InfiniBand adapter and the NVIDIA GPU are attached to the same NUMA node (i.e., processor 0), and processes and memory buffers are bound to this processor in the experiments.

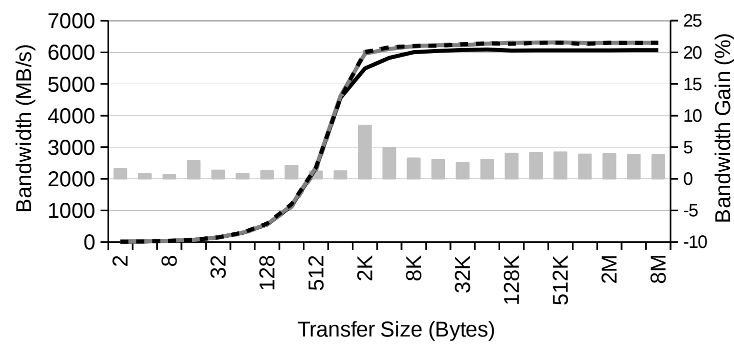
4.4.2.1 Number of Queue Pairs per Port

As depicted in Figure 4.22, in order for an application in one computer to communicate over an InfiniBand network with another application in a different cluster node, it must first create a connection that consists of a queue pair (QP) at each end: one queue for sending data and another queue for receiving them. Interestingly, a QP does not store data but work requests submitted by the application. A work request can be seen as a descriptor of the transfer operation to be performed. A given QP is assigned to one port and a process may create one or more QPs associated to the same network adapter port for communicating purposes with an application in another computer. Obviously, the use of several QPs increases the complexity of maintaining all of them coordinated and synchronized. In this subsection we analyze the impact on performance of using several QPs per port, trying to determine the optimal number of QPs per port that a programmer should use. To do so, we base our analysis in the maximum bandwidth achieved when varying the number of QPs associated to a network adapter port. We make use of the bandwidth benchmarks included in the Mellanox OFED software distribution. These tests measure the bandwidth when copying different data sizes using the channel semantics (i.e., send/receive verbs not using RDMA, `ib_send_bw` benchmark in Figure 4.23(a)), and the memory semantics (i.e., RDMA read and write, `ib_read_bw` and `ib_write_bw` benchmarks, in Figure 4.23(b) and Figure 4.23(c), respectively).

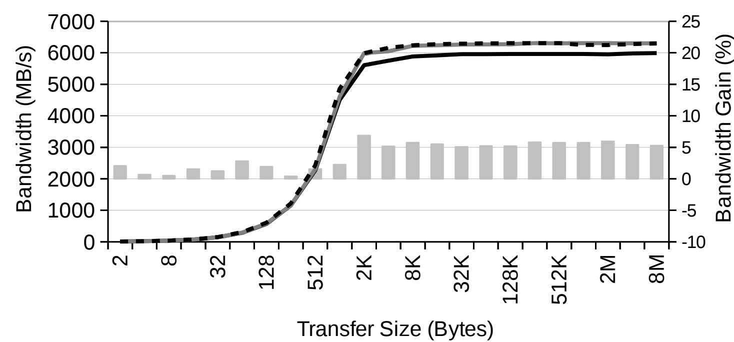
Figure 4.23 shows the results of the mentioned benchmarks, which were run varying the number of QPs per port. Notice that when using more than one QP, the transferred data are split among the available QPs. For instance, when transferring 2KB using 2 QPs,



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

Figure 4.23: InfiniBand bandwidth tests varying the number of queue pairs (QP) per port. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs over using only 1 QP.

1KB is sent using QP1 and 1KB is sent using QP2. Notice also that this division of labor between the several QPs is not automatically performed but the programmer must take care of distributing the work at the same time that all the QPs remain synchronized and balanced. The figure shows the average bandwidth of 100 repetitions for each test. The maximum Relative Standard Deviation (RSD) observed was 0.391 for 16B of transfer size when using 3 QPs in the `ib_write_bw` benchmark.

From the results in Figure 4.23 two main conclusions can be derived. First, there exists a performance difference between using one or several QPs. However, when more than one QP are used, performance remains the same independently of the amount of QPs. Second, results in Figure 4.23 can be divided, from the point of view of performance, into three groups, depending on the size of the transferred data:

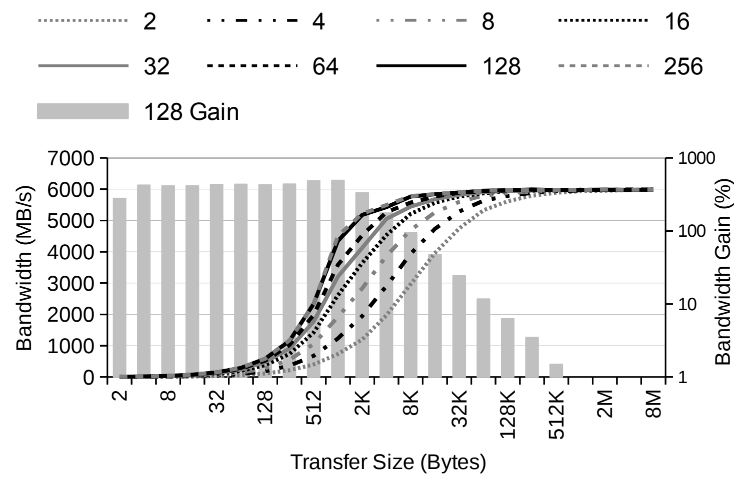
- Less than 2KB: using more than 1 QP translates into an average bandwidth gain of approximately 2%.
- 2KB: the maximum peak bandwidth is achieved because the maximum transfer unit (MTU) is 2KB, and the benefits of using more than 1 QP are more evident.
- 4KB or more: the gain of using more than 1 QP stabilizes, resulting in an average bandwidth improvement of approximately 5%.

Therefore, based on these results, we can conclude that using more than one queue pair per port turns into an average gain of 3.68%. Given that using 2 or more QPs per port provides the same performance, we consider that 2 QPs per port is the optimal value, because the more QPs per port we use, the more the programming complexity increases.

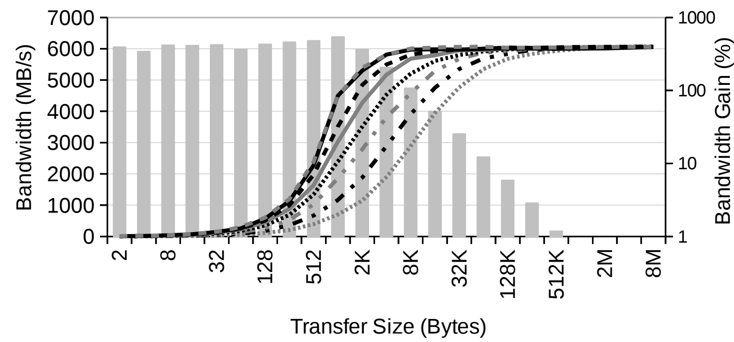
4.4.2.2 Capacity of Send/Receive Queues

As commented previously, applications communicating over InfiniBand must create queue pairs for sending and receiving data. Choosing the length of these queues (i.e., number of work requests they can store) is not a trivial task: the queue should have enough space to allocate all incoming requests from the application in order to not lose performance, but larger queue sizes imply also higher resource consumption. This is especially noticeable in the case of work requests involving RDMA operations, which have associated page-aligned memory regions that must be allocated before submitting the work request to the QP. In this subsection we study the influence of the length of these queues in performance using the attained bandwidth as the metric.

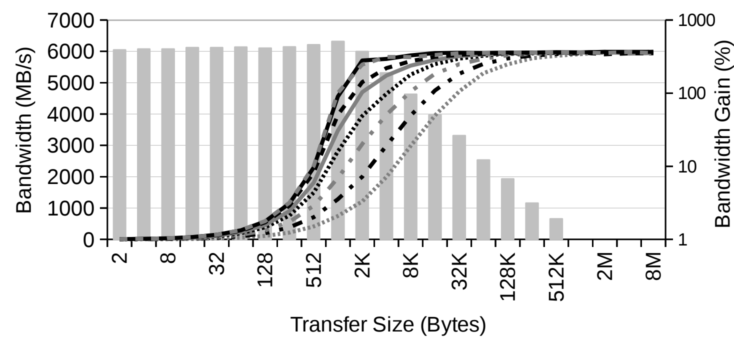
Figure 4.24 shows the results of the bandwidth benchmarks from the Mellanox OFED mentioned before. As in the previous section, we use the `ib_send_bw` benchmark (no



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

FIGURE 4.24: InfiniBand bandwidth tests varying the capacity of the send/receive queues (i.e., number of work requests that can be allocated) from 2 requests to 256. Primary Y-axis shows attained bandwidth, while secondary Y-axis presents the bandwidth gain of using 128 queues over using only 2 queues. Notice the logarithmic scale of the secondary Y-axis.

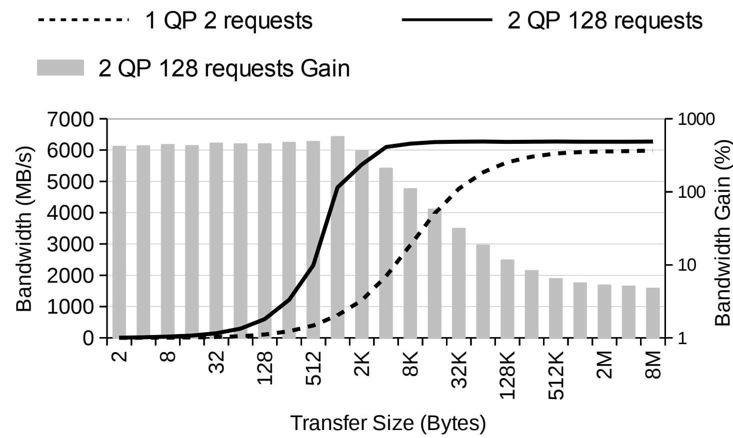
RDMA, Figure 4.24(a)), the `ib_read_bw` benchmark (RDMA read, Figure 4.24(b)), and the `ib_write_bw` benchmark (RDMA write, Figure 4.24(a)). The benchmarks were run varying the length of the send/receive queues. The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.587 for 8B of transfer size when using a queue capacity of 128 requests in the `ib_read_bw` benchmark. As can be observed in Figure 4.24, the queue length is particularly important for small transfer sizes (up to 2KB), where the use of a buffer with space for 128 requests increases the bandwidth an average of 418.69% in comparison to a buffer with capacity for 2 requests. For transfer sizes over 2KB, the bandwidth improvement decreases in the range of 4KB to 512KB, with an average gain of 48.46%. With regard to sizes over 512KB, the gain of increasing the number of queues is almost null (0.22%, on average). Additionally, from these experiments we also extract than using a queue length of more than 128 requests results in no gain.

In summary, averaging the results in Figure 4.24 for all transfer sizes, using a send/receive queue capacity of 128 requests provides a bandwidth gain of 217.14% when compared to a 2-request queue capacity.

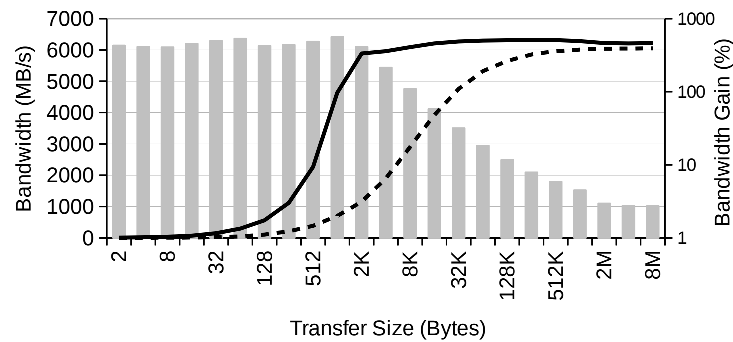
4.4.2.3 Combining both Optimizations

The optimizations presented in the previous subsections complement each other: the first optimization increases performance for large data transfers starting from 2KB, whereas the second optimization boosts performance for small message transfers up to 2KB, point where the increment in performance starts diminishing. Therefore, the obvious question arises: which would be the performance when both optimizations are combined and applied at the same time?

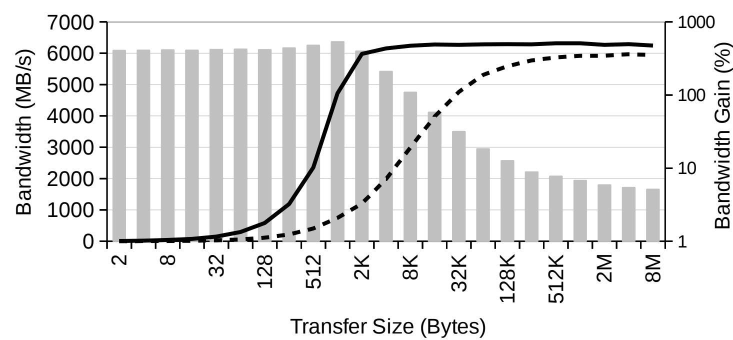
Figure 4.25 presents results for the combination of both optimizations. The results shown are the average bandwidth of 100 repetitions, the maximum RSD being 0.423 for 2B of transfer size when running the `ib_send_bw` benchmark. It can be seen in this figure that bandwidth for transfer sizes up to 2KB is increased, in average, more than 450%. From this point, more modest improvements are achieved, although they are still significant. In this regard, from 4KB up to 512KB, bandwidth is increased, in average, 37.68%, whereas for larger transfer sizes starting from 512KB bandwidth only increases an average of 4.73%. In average, considering all the transfer sizes analyzed, bandwidth is increased 43.29%.



(a) InfiniBand send bandwidth (no RDMA).



(b) InfiniBand RDMA read bandwidth.



(c) InfiniBand RDMA write bandwidth.

FIGURE 4.25: InfiniBand bandwidth tests varying the capacity of the send/receive queues from 2 requests to 128, and number of queue pairs per port from 1 QP to 2. Primary Y-axis shows the benchmark bandwidth, while secondary Y-axis presents the bandwidth gain of using 2 QPs and 128 queues over using only 1 QP and 2 queues. Notice that secondary Y-axis is in logarithmic scale.

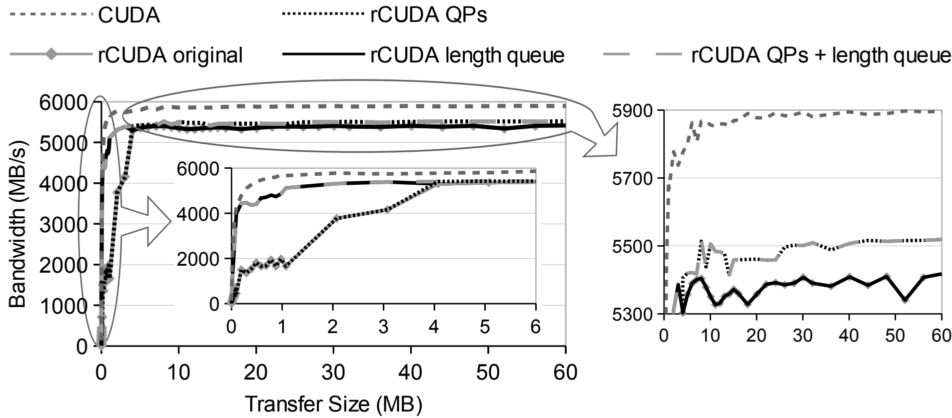


Figure 4.26: Bandwidth test for regular CUDA (using the GPU within the host executing the benchmark) and also for rCUDA (using a remote GPU). Four different versions of rCUDA are considered: the original rCUDA, rCUDA optimized by increasing the capacity of send/receive queues, rCUDA optimized by using two QPs, and rCUDA optimized combining both optimizations.

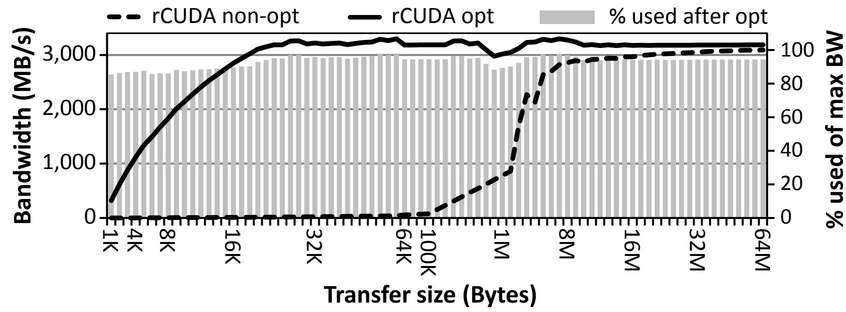
4.4.3 Experiments

In this section we analyze how the optimizations presented in Section 4.4.2 influence the performance of upper software layers. For doing so we use a two level approach: first we analyze these optimizations in the context of the rCUDA framework and later we study the benefits provided to applications that use this framework.

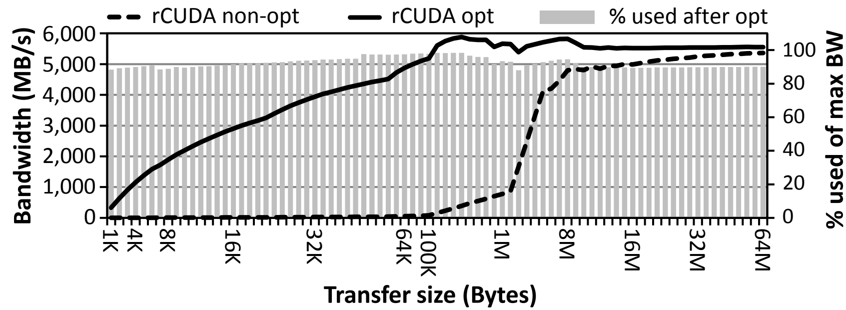
4.4.3.1 Impact of the Optimizations on rCUDA

Figure 4.26 presents the results for a CUDA bandwidth test, available in the NVIDIA CUDA Samples [64]. This test measures the bandwidth when copying data from page-locked system memory to GPU memory. Figure 4.26 presents results when using CUDA and different versions of rCUDA:

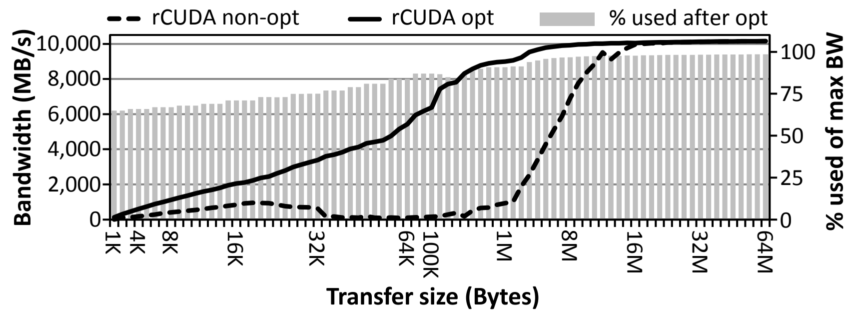
- rCUDA original: this is the current version of rCUDA, which already implements an efficient communication layer based on the use of pipelined transfers [15]. We have included these results for reference.
- rCUDA length queue: this is an enhanced version of rCUDA where, in addition to the already existing pipelined communications, the capacity of send/receive queues has been increased to 128 requests.
- rCUDA QPs: this version of rCUDA uses two QPs in addition to the initial pipelined communication data transfer.
- rCUDA QPs + length queue: this version of rCUDA combines all the optimizations.



(a) InfiniBand ConnectX-2 and NVIDIA Tesla K20.



(b) InfiniBand ConnectX-3 and NVIDIA Tesla K20.



(c) InfiniBand Connect-IB dual port and NVIDIA Tesla K40.

FIGURE 4.27: Performance of rCUDA before and after optimization.

Results shown in Figure 4.26 are the average bandwidth of 100 repetitions, and the maximum RSD observed was 1.319 for 14KB of transfer size when using the initial rCUDA version. However, this high RSD tends to decrease for larger sizes, reaching a maximum of 0.461 for the biggest ones. It can be seen in the figure that when increasing the capacity of send/receive queues to 128 requests there is a noticeable increase in bandwidth (over 4 times more than the bandwidth obtained by the original rCUDA software) for small/medium transfer sizes (up to 4MB). Furthermore, when using two QPs, bandwidth is only increased by 1% with respect to the original rCUDA version. However, in Section 4.23 we have determined an average gain of 3.68% when using multiple QPs. Thus, rCUDA internal paths are limiting the gain. We will investigate further during future work on this matter.

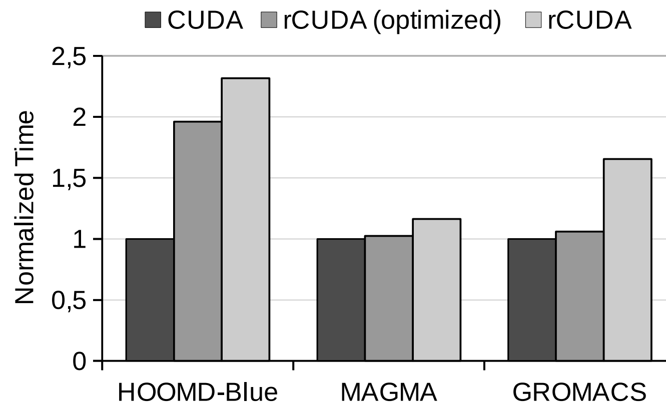
In Figure 4.20, Section 4.3.2.3, we analyzed the bandwidth attained by rCUDA for three different hardware generations: InfiniBand ConnectX-2 (QDR, 40 Gbps), InfiniBand ConnectX-3 (FDR, 56 Gbps), and InfiniBand Connect-IB (more than 100 Gbps). The conclusions of that analysis were that the internal implementation of communications within rCUDA needed to be reviewed, in order to efficiently benefit from the increasing bandwidth that InfiniBand + CUDA provide with each new hardware generation.

In the present section we have addressed the concerns discovered in Section 4.3.2.3, and conveniently optimized rCUDA, as previously shown in Figure 4.26. Figure 4.27 presents the same performance analysis shown in previous Figure 4.20, but this time using the optimized version of rCUDA (using 2 QPs and a queue capacity of 128 requests). It can be seen that for InfiniBand ConnectX-2 and ConnectX-3 the performance of rCUDA has been noticeably improved, reaching almost 100% of available bandwidth for the entire range of transfer sizes. In the case for the dual-port InfiniBand Connect-IB adapter, results for transfer sizes up to 10 MB have been also improved, but they are not so close to 100% of the available bandwidth, achieving almost the 80%, on average. Notice that in this case a network card with two ports is used, whereas for the previous hardware generations the network adapters featured only one port. Thus, this reduction in bandwidth is due to the fact that the two ports of the Connect-IB card are not automatically managed by the network adapter, as explained in Section 4.3, but the application has to deal with all the synchronization issues among them, introducing some overhead for small/medium sizes.

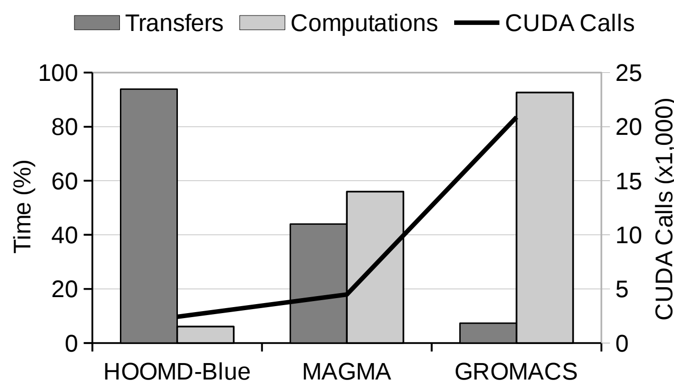
4.4.3.2 Impact of the Optimizations on Applications using rCUDA with Single-port Adapters

Next we evaluate the benefits that the optimized version of rCUDA (using 2 QPs and a queue capacity of 128 requests) provides to applications (the software layer immediately on top of it) using InfiniBand ConnectX-3 single-port network adapters. For that purpose, we use the HOOMD-Blue, MAGMA, and GROMACS production codes:

- HOOMD-Blue [65, 66]: it is a general-purpose particle simulation toolkit. In particular, we have used its version 1.0.1 for our study, running a classic MD simulation, the Lennard-Jones liquid, with 10 random particles and 10 time steps.
- MAGMA [67, 68]: it is a dense linear algebra library similar to LAPACK but for heterogeneous architectures. We utilize release 1.6.0 along with the `dpotrf_gpu` benchmark, which computes the Cholesky factorization for different matrix sizes (from 1K to 10K elements per dimension, in 1K increments).



(a) Normalized execution time of the applications when using regular CUDA, rCUDA and rCUDA optimized.



(b) NVIDIA profiling results: time spent in transfers (i.e., copies to/from GPU memory), time employed by computations (i.e., CUDA kernels), and total number of calls to the CUDA API.

Figure 4.28: Performance evaluation of HOOMD-Blue, MAGMA, and GROMACS.

- GROMACS [69, 70]: it is a versatile package to perform molecular dynamics, i.e., simulate the Newtonian equations of motion for systems with hundreds to millions of particles. We use version 4.6.5 and the ion channel system benchmark with 1K steps.

Figure 4.28 presents the results of this evaluation. Figure 4.28(a), shows the normalized execution time when running these applications with regular CUDA, original rCUDA, and optimized rCUDA (the latter is referred to as *rCUDA (optimized)*, whereas the use of the original version of rCUDA is denoted by the *rCUDA* label). In order to better analyze these results, Figure 4.28(b) shows some profiling results: time spent in transfers (i.e., copies to/from GPU memory, also referred to as CUDA memcopy), time employed by computations (i.e., time employed by CUDA kernels), and total number of calls to the CUDA API. The results shown are the average of 10 repetitions, and the maximum RSD observed was 0.671 when running the HOOMD-Blue simulation with the original version of rCUDA.

As we can observe in Figure 4.28(b), each application presents a different behavior. Firstly, the HOOMD-Blue test has been selected because it represents a scenario where there are much more transfers than computations: over 90% of the test execution time is devoted to transfer data to/from the GPU memory. As expected, this is the worst possible scenario for rCUDA, because the overhead due to the transfers across the network is more evident. Thus, rCUDA needs over 2 times more than CUDA to complete the test. However, it is also a good scenario to show the benefits of the analyzed optimizations in terms of bandwidth gain. In this manner, the optimized version of rCUDA presents an improvement of over the 15% with regard to the initial version of rCUDA.

Next, GROMACS shows the opposite scenario: over 90% of the execution time is devoted to computations in the GPU. Performing much more computations than transfers benefits rCUDA in the sense that the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of rCUDA due to transfers across the network. Notice that this application presents a huge number of calls to the CUDA API. Each CUDA call is forwarded by rCUDA over the network to the remote node owning the real GPU. From the InfiniBand perspective, CUDA calls can be seen as transfers of small size (a header of 12 bytes + a variable size of data depending on the arguments of each CUDA call). As previously shown in Figure 4.26, the optimization consisting in increasing the send/receive queue capacity improved performance for small/medium transfer sizes (up to 4MB). This explains why the rCUDA optimized version needs 35% less time to complete this test, as shown in Figure 4.28(a),

Finally, we have used the MAGMA application experiment to show an scenario where the time spent in transfers and computations is equilibrated (44% of time spent in transfers, 56% in computations). The number of CUDA calls is also an intermediate amount with respect to the previous experiments. In this case, we can attribute the gain when using rCUDA optimized (an 11% when compared to the initial version of rCUDA) to both the increment of the maximum bandwidth because of using two QPs, and the reduction of the time spent in sending small/medium messages due to the increased send/receive queue capacity.

4.4.3.3 Impact of the Optimizations on Applications using rCUDA with Dual-port Adapters

In Section 4.3, we explored how the InfiniBand Connect-IB dual-port network adapter influenced the performance of rCUDA. After analyzing the bandwidth and the latency,

Sequences using DNA alphabet	Sites for each motif
500	0.5 million
1000	1 million
2000	2 millions

Table 4.5: Parameters used for CUDA-MEME application.

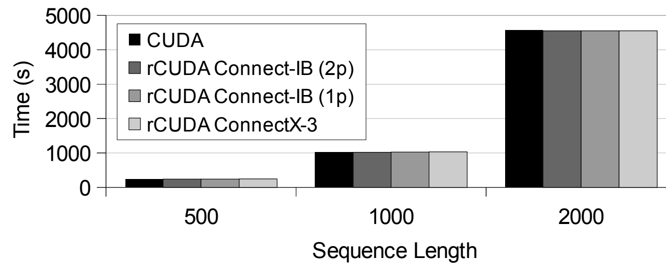
we realized that the internal implementation of communications within rCUDA presented some performance issues. Now that these issues have been addressed, in this section we study how the characteristics of Connect-IB influence the performance of real applications using the optimized version of rCUDA. To that end, we use two applications as study cases: CUDA-MEME and CTA.

CUDA-MEME: Motif Discovery in Biological Sequences

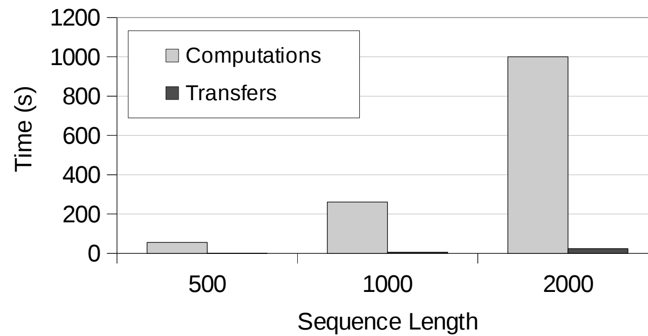
CUDA-MEME [71], is a CUDA-based parallel formulation and implementation of the MEME motif discovery algorithm. Version 3.0.15 was used in our study along with the test cases available in the application website [72], choosing OOPS model for motif distribution with the parameters shown in Table 4.5.

Figure 4.29(a) presents the execution time of CUDA-MEME using CUDA and rCUDA. The overhead of using rCUDA in these experiments is, on average, 2.43%, 1.03%, and 0.51% when using ConnectX-3, Connect-IB 1 port, and Connect-IB 2 ports, respectively. Moreover, the overhead decreases as problem size increases. Thus, for a sequence length equal to 2000, the overhead is 0.24%, 0.22%, and 0.17% when using ConnectX-3, Connect-IB 1 port, and Connect-IB 2 ports, respectively. The reason for the overhead reduction introduced by rCUDA as sequence length increases is the increased difference between the time employed in transfers and computations, as shown in Figure 4.29(b). In general, the more computations an application performs, the less overhead is introduced by rCUDA because the time spent in computations in the GPU is the same for CUDA and rCUDA, thus compensating the overhead of transferring data across the network.

These results are aligned with the ones in previous sections and reinforce the conclusion that the overhead of remote GPU virtualization is reduced when using the InfiniBand Connect-IB adapter. Nevertheless, the improvement obtained when using this card is not as noticeably as it could be expected after the bandwidth increase observed in Figure 4.18. The reason is that this is an application computation bound, in which time spent doing computations in the GPU is clearly larger than time spent transferring data to/from the GPU memory, as depicted in Figure 4.29(b). Therefore, given that data transfers represent a small fraction of total execution time, a reduction in the time



(a) CUDA-MEME execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports).



(b) NVIDIA profiling results. Time employed by computations (i.e., CUDA kernels) and memory transfers (i.e., CUDA memcopy) is shown.

FIGURE 4.29: CUDA-MEME analysis.

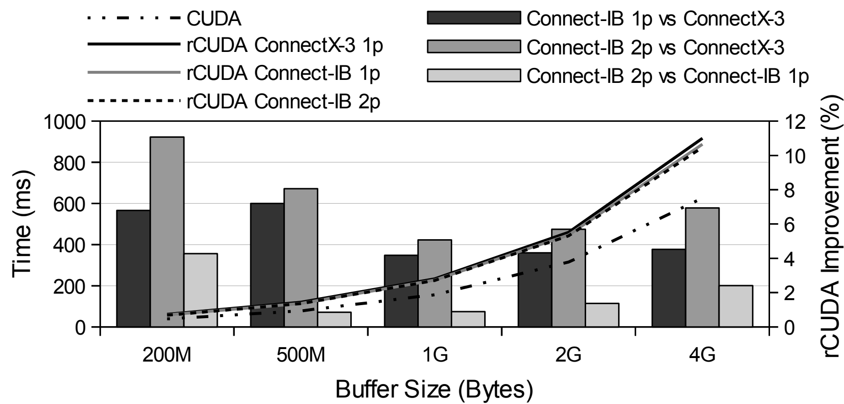
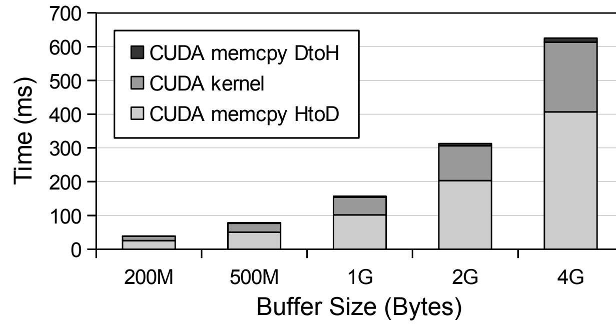
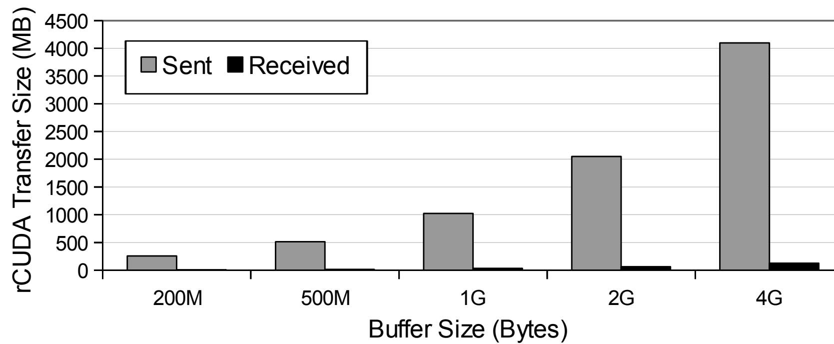


FIGURE 4.30: Primary y-axis shows CTA execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). Secondary y-axis presents the rCUDA improvement of using Connect-IB (with 1 and 2 ports) with respect to ConnectX-3.

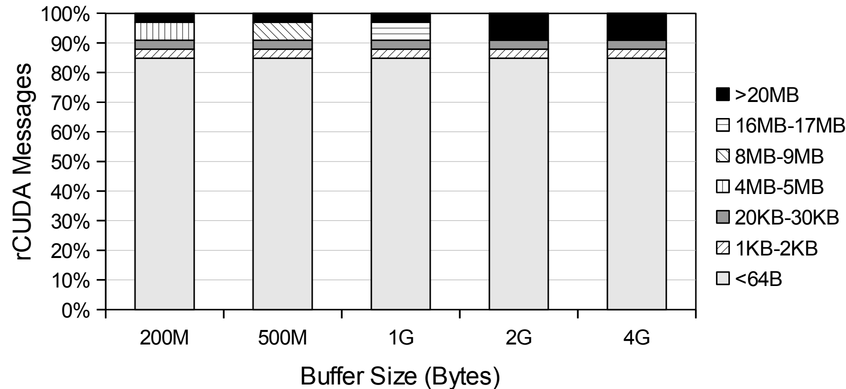
required for transferring data translates into a much smaller reduction in total execution time. In the next section, we show an application bandwidth bound in order to better reflect the overhead reduction.



(a) NVIDIA profiling results. Time spent copying data to the GPU (*CUDA memcopy HtoD*), performing computations (*CUDA kernel*), and transferring data back to host memory (*CUDA memcopy DtoH*).



(b) rCUDA profiling results. Total amount of bytes exchanged between rCUDA client and server.



(c) Histogram of rCUDA messages attending to their size.

Figure 4.31: Profiling of the CTA application.

CTA: Signal Extraction

CTA [73] is a signal extraction algorithm pertaining to the entry stage of the Cherenkov Telescope Array's Real Time Analysis pipeline, implemented using CUDA. The algorithm receives waveforms as input. In these experiments we have measured the processing time of different input buffer sizes: 200MB, 500MB, 1GB, 2GB and 4GB.

Figure 4.30 shows CTA execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports).

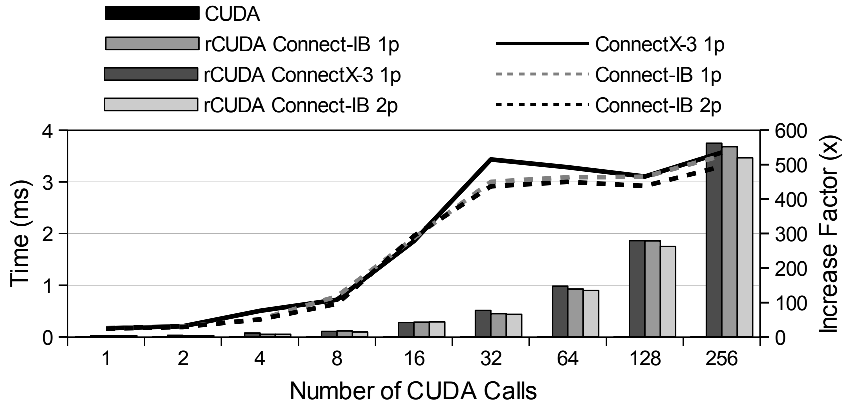


FIGURE 4.32: Test varying the number of CUDA calls. Each CUDA call translates into one rCUDA small message (shorter than 64 bytes). Primary y-axis shows execution time using CUDA and the rCUDA framework over InfiniBand employing different cards: ConnectX-3 and Connect-IB (with 1 and 2 ports). Secondary y-axis shows the increased factor of rCUDA overhead with respect to CUDA.

The right axis of this figure also presents the improvement of using Connect-IB (with 1 and 2 ports) with respect to ConnectX-3. As it can be seen, the use of Connect-IB with 2 ports reduces rCUDA overhead up to 11% for a 200MB buffer size, and over 7%, on average, for all the buffer sizes tested, with respect to ConnectX-3. This improvement was expected because this application is bandwidth bound, as shown in Figure 4.31(a). In that figure one can see that this application spends most of the time in copying data to GPU memory (bar segment labeled as *"CUDA memcopy HtoD"*).

However, although in this application the overhead of rCUDA is clearly reduced when using Connect-IB cards, the overhead of rCUDA with regard to CUDA is larger than in the case of CUDA-MEME, as can be seen in Figure 4.30. One may think that this higher overhead was in part expected, given that this application spends much more time in transferring data than CUDA-MEME and, given that rCUDA does not use 100% of the available bandwidth, then the more data is transferred, the higher the overhead is. In this regard, Figure 4.31(b) shows that this application transfers, for the largest input buffer used, more than 4 GB to the rCUDA server. However, a simple calculation shows that the small difference in the maximum attained bandwidth between CUDA and rCUDA does not completely explain the execution time overhead shown in Figure 4.30. This calculation is the following: sending 4 GB of data to the remote server at maximum speed (10 GB/s) takes 0.4 seconds. Thus, given that the difference in maximum bandwidth achieved between CUDA and rCUDA is only 1.5%, then the rCUDA overhead of transferring these 4 GB of data would be 6 milliseconds, what is much less than the overhead shown in Figure 4.30. Thus, although the lower bandwidth of rCUDA increases execution time, it is not the main reason for the overhead.

In order to find a more contributing reason for the overhead, notice that Figure 4.31(b) shows the total amount of data transferred, but this data does not necessarily have to

be transferred in one single message. Furthermore, smaller transfer sizes attain lower bandwidth. Thus, given that the 4 GB of input data will not probably be transferred in a single chunk, it is necessary to analyze the different message sizes actually used. In this regard, as presented in Figure 4.31(c), over 80% of the messages between rCUDA client and server are smaller than 64 bytes. Moreover, an additional analysis of the messages exchanged between the rCUDA client and server, attending to message type, revealed that these small messages are not data copies to/from GPU memory, but CUDA calls such as CUDA synchronization points, for example, which do not carry data. Furthermore, notice that these small messages present a non-negligible latency given that they are CUDA commands forwarded to the GPU which returns a short response. Hence, these small messages might be seen as ping-pong messages. Therefore, it could be possible that this large percentage of small messages is the cause for the execution time overhead shown in Figure 4.30.

In order to analyze the impact of these small messages, we have implemented a test which performs a varying amount of CUDA calls. Each of these CUDA calls later translates into one small rCUDA message (shorter than 64 bytes) and its associated response from the GPU. Results for this test are presented in Figure 4.32. As we can see, rCUDA performance is clearly impaired by this kind of messages, thus explaining the large overhead of rCUDA when remotely executing the CTA application.

4.4.4 Summary

The use of InfiniBand networks to interconnect high performance computing clusters has considerably increased during the last years. However, due to the programming complexity of the InfiniBand API and the lack of documentation, there are not enough recent available studies explaining how to optimize applications to get the maximum performance of this fabric.

In this section we have exposed two general optimizations to be used when developing applications using InfiniBand Verbs. Based on our experiments we can conclude that (1) using more than one queue pair per port clearly improves bandwidth (an average gain of 3.68% in our experiments), (2) increasing the capacity of the send/receive queues turns into an average bandwidth improvement of over 200%, being especially noteworthy for small/medium message sizes (over 400% more bandwidth in our experiments), and (3) both optimizations complement each other. In this regard, when combining both optimizations, the average bandwidth gain is 43.29%. This bandwidth increment is key for remote GPU virtualization frameworks. Actually, this noticeable gain translates into a reduction of up to 35% in execution time of applications using remote GPU

virtualization frameworks. These optimizations were released in versions 15.07, 15.10 and 16.06 of the rCUDA framework (see Table 2.2 for more details).

4.5 Influence of EDR InfiniBand on the Bandwidth of rCUDA

During the time frame used to carry out this thesis, several generations of InfiniBand adapters appeared, as previously shown in Figure 1.1. In this regard, by the end of this thesis EDR 100G InfiniBand adapters came on the market. In this section we analyze how the high bandwidth provided by this InfiniBand fabric allows remote GPU virtualization middleware systems not only to perform very similar to local GPUs, but also to improve overall performance for some applications.

4.5.1 Background

As commented in previous sections, remote GPU virtualization may experience a noticeable reduction in their overhead with respect to the use of local GPUs when high performance interconnects are used. In this regard, recent advances in networking technologies have considerably reduced the performance gap between the intra-node PCIe link and the inter-node fabric. For example, as depicted in Figure 4.1, the theoretical throughput of the latest available versions of PCIe and InfiniBand are relatively close: 15.75 GB/s for PCIe 3.0 x16 vs. 12.5 GB/s for EDR 100G InfiniBand using Mellanox adapters. Furthermore, when considering the effective bandwidth of these technologies instead of the theoretical numbers, EDR InfiniBand presents better results, as shown later.

In this manner, the main contribution of this section is showing how this high effective bandwidth provided by EDR InfiniBand allows remote GPU virtualization middleware systems not only to perform very similar to local GPU-accelerators, but also to improve the performance for some applications. The optimizations described in the previous sections are also leveraged in this section.

The rest of the section is organized as follows. Section 4.5.2 further investigates the effective bandwidth of EDR InfiniBand networks and PCIe 3.0 when used by the latest NVIDIA GPUs at the time of performing this analysis. Section 4.5.3 provides a performance evaluation of the rCUDA middleware using both popular benchmarks and applications. Finally, Section 4.5.4 summarizes the main conclusions of this work.

4.5.2 Motivation

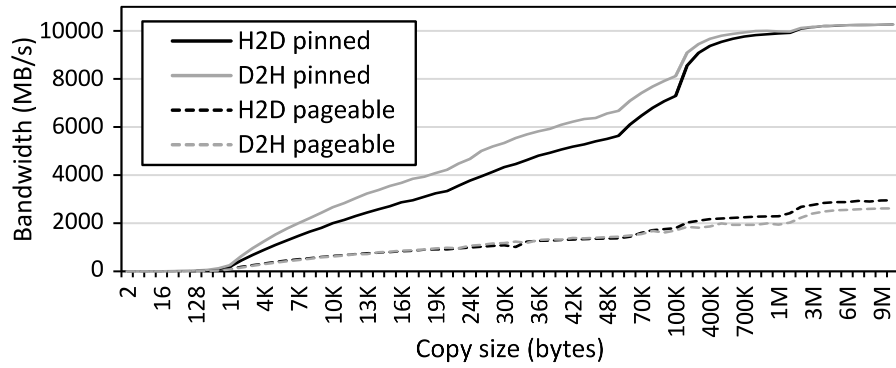
As shown in Figure 4.1, the theoretical bandwidth of EDR InfiniBand and that of PCIe 3.0 are relatively close. However, it is well known that the actual performance of any interconnect largely depends on the ability of the upper software layers to obtain as much bandwidth as possible from it. Thus, the effective performance of an interconnect is usually lower than the theoretical one. The real concern, therefore, is not about the theoretical numbers depicted in Figure 4.1 but about real performance, that is, whether real bandwidth numbers help to reduce the performance difference among the external InfiniBand fabric and the internal PCIe or, on the contrary, make that difference larger, thus increasing the penalty of remote GPU virtualization techniques.

To address this concern, in this section we compare the EDR InfiniBand fabric against the PCIe 3.0 link, when used by their respective upper software layers: the network driver in the case of InfiniBand and the CUDA driver, along with an NVIDIA GPU, in the case of the PCIe 3.0. This analysis will expose the maximum performance that might be achieved by a remote GPU virtualization solution and, additionally, will put the performance of such a virtualization solution into the right perspective.

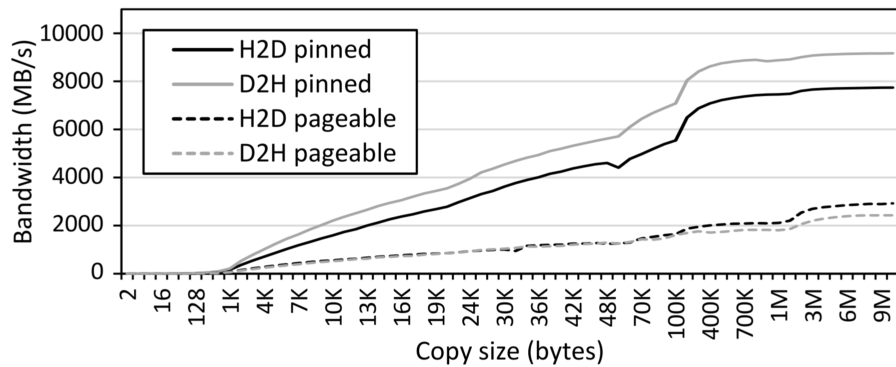
The testbed used in this analysis consists of two 1027GR-TRF Supermicro nodes featuring two Intel Xeon E5-2620v2 processors (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 memory at 1600 MHz. The computers also include an EDR InfiniBand adapter. One of the nodes has installed an NVIDIA Tesla K40 GPU and also an NVIDIA Tesla K80 GPU. Linux CentOS 6.4 was used along with CUDA 7.0 (NVIDIA driver 340.46) and Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools).

The test servers used are NUMA machines. NUMA effects matter for the performance evaluation presented in this section. For this reason, the InfiniBand card and the K40 GPU are connected to the same PCIe root (i.e., processor 0) whereas the K80 GPU is connected to processor 1. For all the experiments shown in this section, processes and memory buffers are appropriately bound either to processor 0 or 1. Additionally, the persistence mode of the NVIDIA GPUs was enabled in order to obtain the best performance.

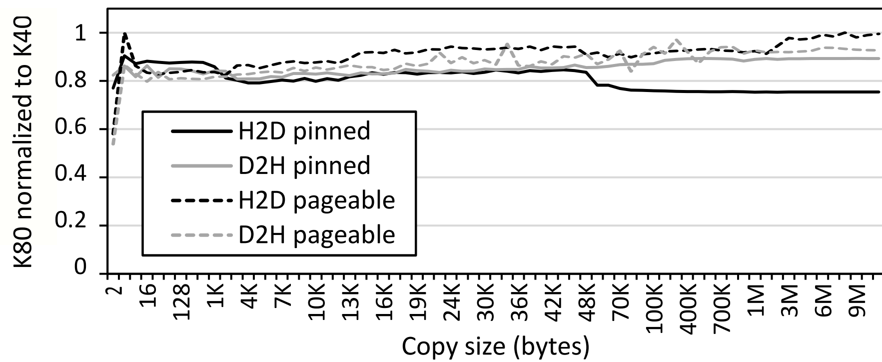
Figure 4.33 shows the performance attained by the NVIDIA K40 and K80 GPUs when the `bandwidthTest` benchmark from the NVIDIA samples [74] is used. It can be seen in Figure 4.33(a) that the bandwidth achieved by copies using pinned and pageable host memory is very different. Actually, copies using pinned memory achieve up to four times the performance of copies using pageable memory. Moreover, it is interesting to notice that even for the largest performance attained by the K40 GPU, maximum effective bandwidth is about 35% lower than the PCIe 3.0 theoretical bandwidth.



(a) Performance of the NVIDIA Tesla K40 GPU.



(b) Performance of the NVIDIA Tesla K80 GPU.



(c) Performance of K80 normalized to that of K40.

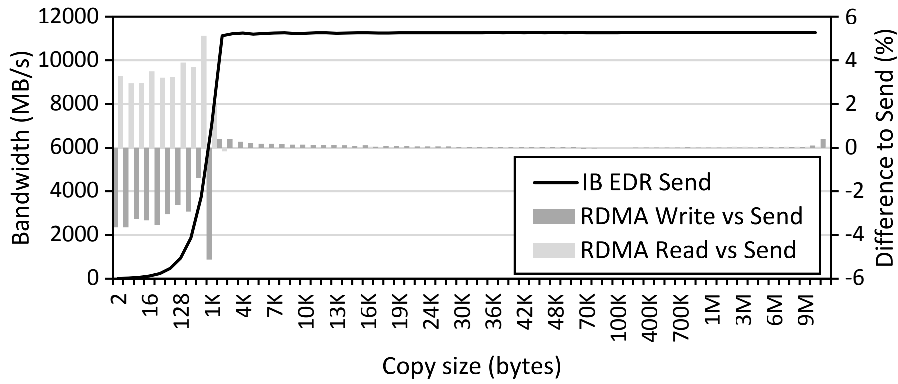
Figure 4.33: Bandwidth test for copies between host memory and GPU memory using pinned and pageable host memory. H2D refers to the copies from host to the GPU, whereas D2H refers to the opposite direction.

Furthermore, applications usually leverage pageable memory. Hence, in practice the performance of memory copies to/from the K40 GPU is almost six times lower than the theoretical performance of PCIe 3.0. This fact supports our initial assumption that effective bandwidth should be considered instead of theoretical numbers. Regarding the performance of the K80 GPU shown in Figure 4.33(b), it can be seen that similar conclusions can be reached. Nevertheless, as Figure 4.33(c) shows, the performance of the K80 GPU is lower than that of the K40 one. For this reason, in the rest of the

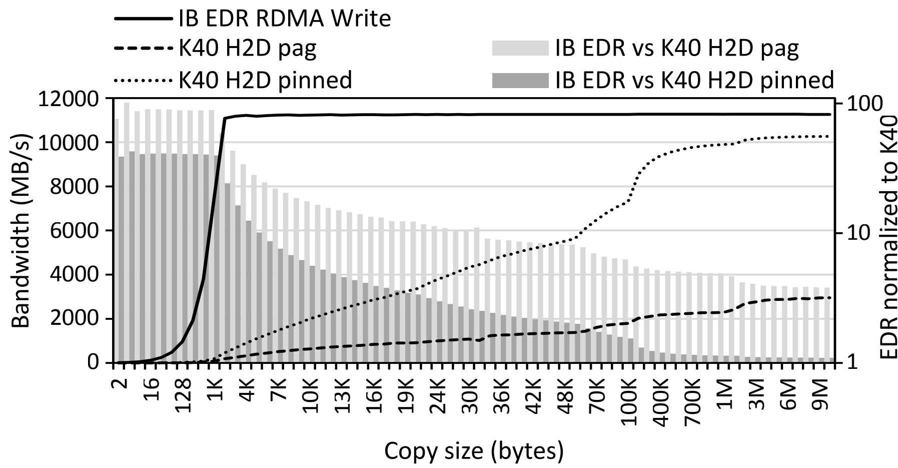
section we will only consider the use of the K40 GPU, because it represents a harder burden for our study.

Once the performance of the PCIe 3.0 link, along with NVIDIA GPUs, has been analyzed, we continue our study with the performance of the EDR InfiniBand network. Figure 4.34 presents the performance attained by this interconnect when the `ib_write_bw`, `ib_read_bw`, and `ib_send_bw` benchmarks from the Mellanox OFED are used. These tests measure the bandwidth when copying different data sizes using the channel semantics (i.e. send/receive verbs not using RDMA, `ib_send_bw` benchmark), and the memory semantics (i.e. RDMA read and write, `ib_read_bw` and `ib_write_bw` benchmarks, respectively). First, Figure 4.34(a) presents the performance of the `ib_send_bw` benchmark and compares it with the other two InfiniBand benchmarks. It can be seen that the three benchmarks provide very similar results. Only for copy sizes below 1K byte there is a maximum of 5% of difference among them. On the other hand, Figure 4.34(b) compares the performance of the `ib_write_bw` benchmark with that of the K40 GPU for copies H2D using pageable and pinned host memory. The left Y axis shows absolute bandwidth numbers whereas the right Y axis shows the performance of the EDR network normalized to that of the K40 (in logarithmic scale). It can be seen in Figure 4.34(b) that the performance of the EDR InfiniBand fabric is almost 100 times larger than that of the K40 using pageable memory for small copy sizes, whereas for larger copy sizes the K40 GPU is about 4 times slower. In a similar way, Figure 4.34(c) depicts a performance comparison involving the `ib_read_bw` benchmark as well as D2H memory copies using pageable and pinned memory. Again, the EDR InfiniBand network fabric performs better (between 100x and 5x than the K40 GPU when pageable memory is used).

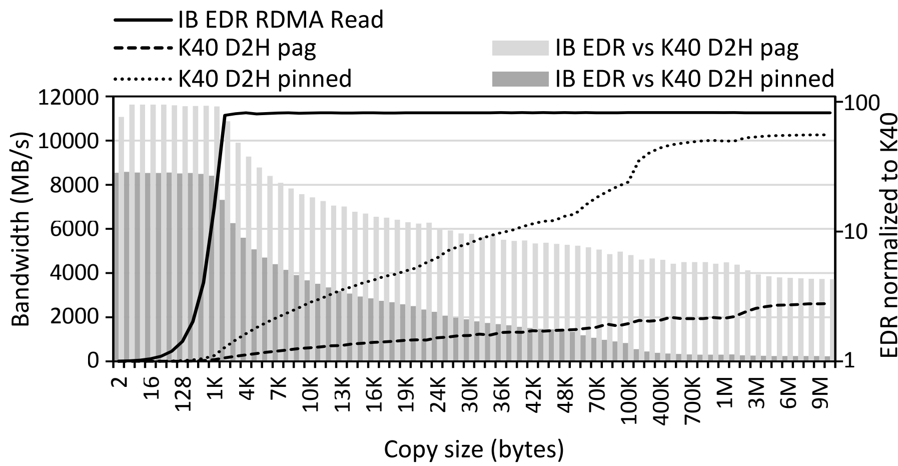
As a summary of the analysis presented in this section, when the effective bandwidth numbers of PCIe 3.0 and EDR InfiniBand interconnects are considered instead of the theoretical values, the main conclusion is that the EDR InfiniBand network fabric provides higher bandwidth than the latest NVIDIA GPUs. This difference in performance provides an interesting conclusion: the effective bandwidth of the external interconnect (EDR InfiniBand) is noticeably larger than the maximum bandwidth attained by the internal interconnect (PCI 3.0 x16 + CUDA). This fact introduces new possibilities for remote GPU virtualization solutions, which so far were strongly limited by the lower performance of the external interconnect. This conclusion is the main motivation for the study presented in this section about the performance of remote GPU virtualization middleware systems.



(a) Comparison among send, read, and write benchmarks.



(b) Comparison among InfiniBand RDMA write and H2D CUDA copies.



(c) Comparison among InfiniBand RDMA read and D2H CUDA copies.

FIGURE 4.34: Performance of the InfiniBand (IB) bandwidth tests, compared to the performance of the NVIDIA K40 GPU.

4.5.3 Experiments

This section presents the impact that the new EDR InfiniBand interconnect has on the performance of the rCUDA middleware. For this purpose, we first analyze the maximum bandwidth attained for memory copies between host memory and remote GPU memory. Next, we use the Rodinia benchmark suite [75] to characterize how several application kernels perform in this virtualized scenario. Finally, we make use of seven different applications to analyze how the improved features of the new interconnect influence the performance of applications using rCUDA.

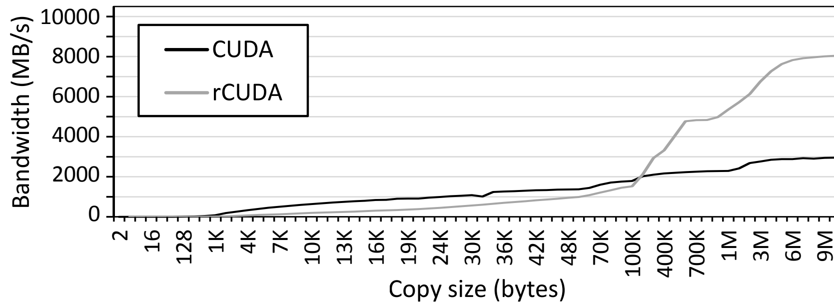
4.5.3.1 Bandwidth analysis

Figure 4.35 presents a comparison between the bandwidth attained by CUDA when using the local K40 GPU, as in the traditional way, and the performance achieved when using the rCUDA middleware with a remote K40 GPU. The testbed used is the same as in Section 4.5.2. The `bandwidthTest` benchmark from the NVIDIA samples is used. Figures 4.35(a) and 4.35(b) use pageable host memory whereas Figures 4.35(c) and 4.35(d) use pinned host memory. It can be seen that when pageable host memory is used, the bandwidth achieved when using a remote GPU is higher than that obtained for a local GPU for copy sizes larger than 100K bytes. Actually, for the H2D direction, the rCUDA middleware achieves almost 3 times the bandwidth attained by CUDA. In the case of the D2H direction, rCUDA doubles the performance of CUDA. This is a well-known effect thoroughly described in previous works on rCUDA [15] 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, when pinned memory is considered, rCUDA is able to achieve more than 90% of the bandwidth attained by CUDA for the largest copy sizes. Nevertheless, remember than applications usually employ pageable memory instead of pinned memory.

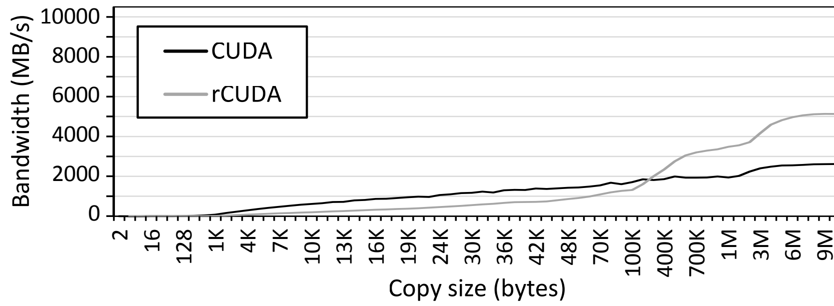
As a brief summary of the results presented in Figure 4.35, the main conclusion is that CUDA applications may experience some additional acceleration when using the rCUDA middleware due to the higher bandwidth used to move data to/from the GPU. Nevertheless, in addition to bandwidth, other issues should also be considered, as it will be shown in next subsection.

4.5.3.2 The Rodinia Benchmark Suite

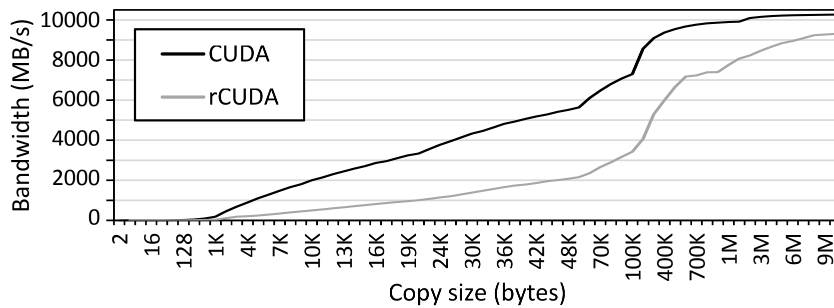
Rodinia [75] is a popular benchmark suite for heterogeneous computing aimed to help architects study platforms such as GPUs. It includes applications and kernels which



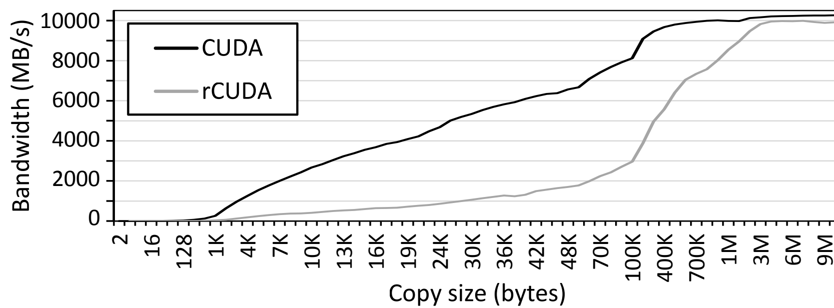
(a) Copies from pageable host memory to GPU memory.



(b) Copies from GPU memory to pageable host memory.



(c) Copies from pinned host memory to GPU memory.



(d) Copies from GPU memory to pinned host memory.

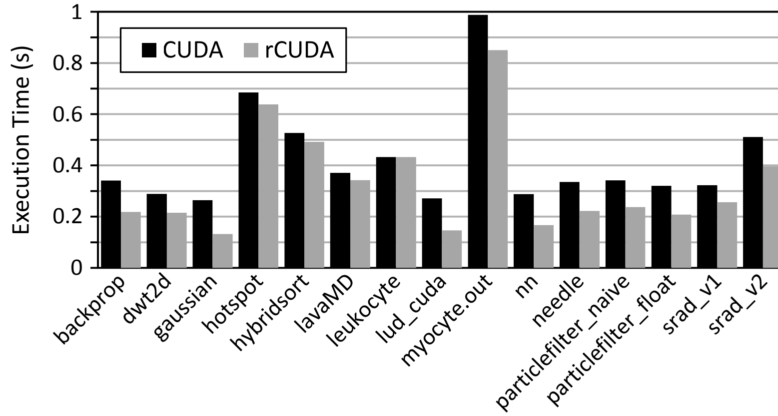
FIGURE 4.35: Bandwidth test for copies between host memory and GPU memory using CUDA and the rCUDA middleware over the EDR InfiniBand fabric.

target multi-core CPU and GPU platforms. The Rodinia benchmarks cover a wide range of parallel communication patterns and synchronization techniques, which we consider useful for an initial study.

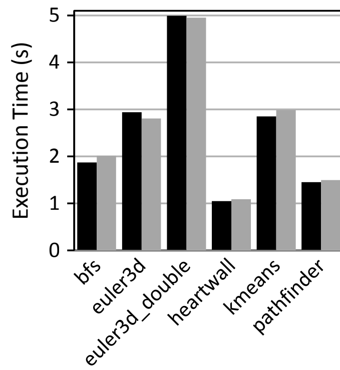
For the experiments shown in this section, we have used the version 3.0 of the Rodinia benchmark, following the instructions inside each benchmark for running them. Figures 4.36(a), 4.36(b), and 4.36(c) compare the execution time of several of the benchmarks included in this suite (for a complete description of each of these benchmarks please refer to [75]). Figure 4.36(a) presents results for benchmarks whose execution time is under 1 second. Figure 4.36(b) presents results for benchmarks that require less than 5 seconds to complete their execution and Figure 4.36(c) is devoted for a benchmark presenting a much longer execution time. Hence, benchmarks with very different execution times are considered in this study. It can be seen in these three figures that the execution of these benchmarks with the rCUDA middleware is, in general, faster than with CUDA. Figure 4.36(d) summarizes, in terms of percentage, the difference of time between rCUDA and CUDA. This figure shows that for short benchmarks, the remote case is in general noticeably faster than the local case. However, for the medium benchmarks there is not such a predictable improvement. Finally, for the long benchmark analyzed, the remote case performs better than the local case.

One may think that the better performance of rCUDA is due to the higher bandwidth attained for copies using pageable memory. However, a deeper profiling revealed that some of the analyzed codes have synchronization points, such as calls to `cudaDeviceSynchronize` or `cudaStreamWaitEvent`, that take more time when using CUDA than when using the rCUDA middleware. For instance, a call to `cudaDevice-Synchronize` takes about 530 microseconds in CUDA, whereas it takes only about 40 microseconds in rCUDA. The reason for this large difference lies in the internal algorithm used in the rCUDA middleware to determine the finalization of the CUDA tasks, which benefits rCUDA in these short benchmarks, as shown before in Section 3.3. Nevertheless, the time saved in synchronization points in rCUDA does not fully explain the shorter execution time of these benchmarks. One additional factor that affects execution time is the network polling interval (i.e., the frequency used to poll the network for work completions). This interval in the rCUDA implementation is lower than the default polling interval used by CUDA to poll the PCIe link, as previously demonstrated in Section 3.3. Thus, for short benchmarks where the aggregated weight of these small waits becomes a large fraction of the total execution time, the net result is that rCUDA performs better than CUDA.

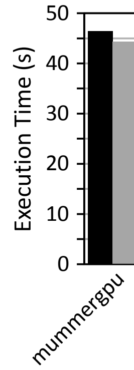
In summary, there are many different factors that simultaneously contribute to increase and reduce the overhead of rCUDA with respect to CUDA. Hence, for each of the benchmarks executed, the exact combination of these factors results in a better or worse execution time. Nevertheless, as shown in Figure 4.36, the average overhead of rCUDA with respect to CUDA is about 5% (considering only those cases where rCUDA increases total execution time).



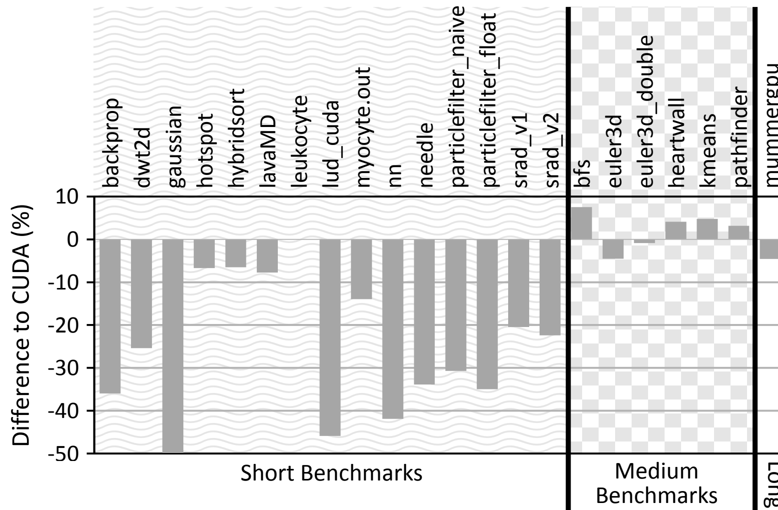
(a) Execution time of short benchmarks.



(b) Execution time of medium benchmarks.



(c) Execution time of a long benchmark.



(d) Difference in execution time of rCUDA with respect to CUDA.

FIGURE 4.36: Execution time of several Rodinia benchmarks using CUDA and the rCUDA middleware over EDR InfiniBand.

4.5.3.3 Production Applications

This section completes the analysis of the performance of the rCUDA middleware by applying it to 7 production applications[3]: CUDASW++, GPU-LIBSVM, LAMMPS, GPU-BLAST, MAGMA, CUDA-MEME, and BarraCUDA.

CUDASW++ [77] 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. We have used its last release, version 3.1, for our study, along with the Latest Swiss-Prot database and a query with 5,478 sequences. Both the database and the query are available in the application's website: http://cudasw.sourceforge.net.

GPU-LIBSVM [78] is an integrated software that supports vector classification, (C-SVC, nu-SVC), regression (epsilon-SVR, nu-SVR) and distribution estimation (one-class SVM). In addition, it supports multi-class classification. For our experiments, we have used version 3.18, and the input data included in the package. More specifically, we have scaled the available training data, without the use of the shrinking heuristics and utilizing a 10-fold cross validation mode.

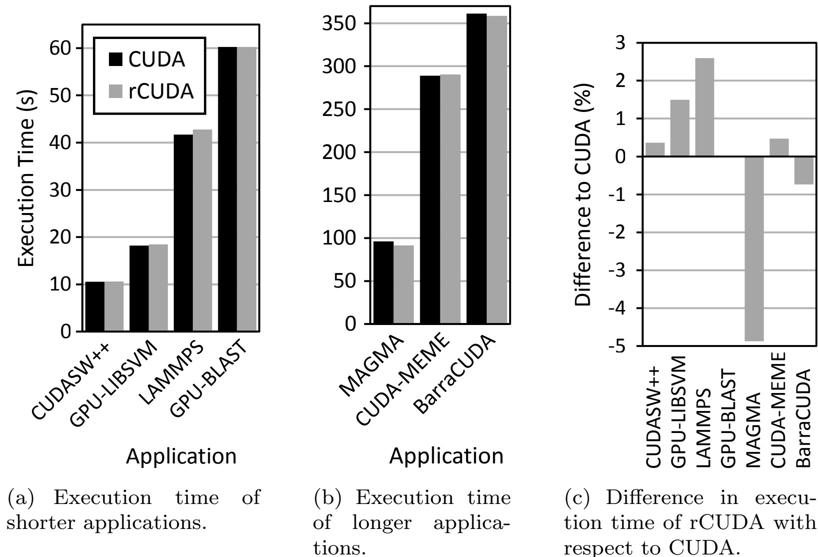
LAMMPS [40] is a classic molecular dynamics simulator that can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, mesoscopic, or continuum scale. For the test in this work, we use the release from May 15, 2015, and the benchmark "in.eam" installed with the application. We run the benchmark with one processor and one GPU, scaling by a factor of 5 in all three dimensions (i.e., a problem size of 4 million atoms).

GPU-BLAST [79] has been designed to accelerate the gapped and ungapped protein sequence alignment algorithms of the NCBI-BLAST implementation using GPUs. We utilize release 1.1, following the installation instructions for sorting a database and creating a GPU database. To search the database, we then use a query with 1,400 sequences that comes with the application package.

The MAGMA [80] project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures (current multi- core+GPU systems). We have run the DLARFB tests, which applies a real block reflector H or its transpose H^H to an M by N matrix C, from the left. The range of matrix sizes used (M=N=K) was from 1,088 to 4,160 in increments of 1,024.

CUDA-MEME [81] is a parallel formulation and implementation of the MEME motif discovery algorithm using CUDA. Version 3.0.15 was used in our study along with

[3]All the applications have been extracted from the NVIDIA list of GPU applications [76].



(a) Execution time of shorter applications.

(b) Execution time of longer applications.

(c) Difference in execution time of rCUDA with respect to CUDA.

FIGURE 4.37: Execution time of several applications using CUDA and the rCUDA middleware over the EDR InfiniBand fabric.

the testcases available in the application website, choosing OOPS model for motif distribution, with 500 sequences using DNA alphabet, and 0.5 million of sites for each motif.

The aim of the BarraCUDA [82] project is to develop a sequence mapping software that utilizes the massive parallelism of GPUs to accelerate the inexact alignment of short sequence reads to a particular location on a reference genome. We have used a simulation of the most current genome build called "Illumina" as database, and a query using a method called "STAMPY". The read length was 37.

Figures 4.37(a) and 4.37(b) depict a comparison between the execution time of these applications in the local and remote scenarios. In a similar way to the Rodinia benchmarks, these applications have been selected because they present a wide range of execution times. Figure 4.37(c) presents the overhead of rCUDA with respect to CUDA. It can be seen that, in general, the overhead is lower than 3%. Moreover, some of the applications present small improvements. Again, a deeper profiling reveals similar conclusions as in Section 4.5.3.2. Therefore, as also explained in that sections, the exact reasons for the increase or decrease of execution time when using rCUDA depends on each application. In general, the reasons lay on the same basis as the ones detailed in Section 3.3.

4.5.4 Summary

In this section the impact of EDR InfiniBand on the rCUDA middleware has been investigated. The main motivation for this study is the higher bandwidth of the EDR InfiniBand network fabric with respect to the bandwidth attained by current NVIDIA GPUs leveraging the PCIe 3.0 link. In addition to basic bandwidth tests, the remote GPU virtualization rCUDA middleware has been analyzed by using 20 different benchmarks from the Rodinia suite and 7 production applications selected from the NVIDIA list of GPU applications.

The main conclusion from this work is the noticeable reduction of the overhead experienced by remote GPU virtualization solutions, generally speaking, and more specifically by the rCUDA middleware. In this regard, for most applications the overhead is below 5%. Furthermore, some applications experience a reduction in their execution time with respect to the local CUDA case. We can thus conclude that local and remote GPUs perform similar when using EDR InfiniBand.

4.6 Conclusions

This chapter has presented performance analysis and optimizations carried out for different InfiniBand networks in the context of the rCUDA middleware. First, we have analyzed the influence of FDR InfiniBand on the performance of rCUDA. Then, we have introduced a new version of rCUDA supporting InfiniBand dual-port adapters, such as the InfiniBand Connect-IB (FDR x 2) ones, and have also evaluated its performance. Next, we have exposed different optimizations to be used when developing applications using InfiniBand Verbs and we have studied its impact on rCUDA. Finally, we have analyzed how the high bandwidth provided by EDR InfiniBand, along with the optimizations exposed in previous chapters, allows rCUDA not only to perform very similar to local GPUs, but also to improve overall performance for some applications.

Chapter 5

Peer to Peer Memory Copies between Remote GPUs

Prior to this thesis, rCUDA did not support memory copies between remote GPUs located in different nodes. This chapter presents an efficient mechanism devised for addressing the support for this kind of memory copies among GPUs located in different cluster nodes. Several options are explored and analyzed, such as the use of the GPUDirect RDMA mechanism. Results show that it is possible to implement this kind of memory copies in such an efficient way that performance is even improved with respect to the original performance attained by CUDA when GPUs located in the same cluster node are leveraged. The work presented in this chapter has been published in [83]. It has also been submitted to the Journal of Parallel and Distributed Computing. Finally, the work presented in this chapter was introduced in version 16.11 of rCUDA, which was officially released at the Supercomputing Conference 2016 in Salt Lake City, Utah, USA.

Link with Industry

As the use of GPUs for accelerating applications becomes widespread, systems with more than one GPU begin to appear. In response to this, NVIDIA improves CUDA by offering faster multi-GPU programming tools, such as (1) the Unified Virtual Addressing (UVA), which provides a single address space for both system memory and GPUs memory; and (2) a new version of GPUDirect with support for Peer to Peer communication (i.e., direct transfers between GPUs).

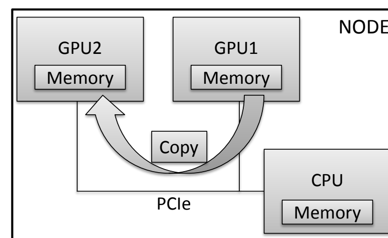
All this leads to the development of applications leveraging multiple GPUs. Once again, rCUDA needs to evolve in response to industry trends: it is necessary to offer Peer to Peer support for remote virtual GPUs, regardless of whether the GPUs are located in the same remote server or in different remote servers.

5.1 Introduction

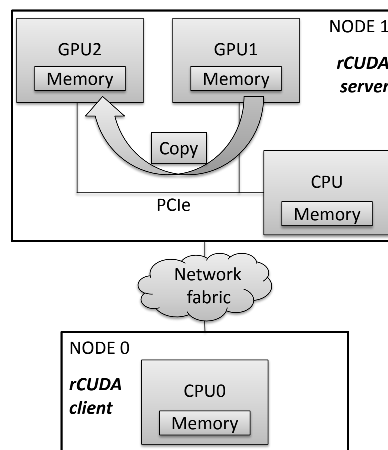
In order to better introduce the idea around P2P memory copies, Figure 5.1 presents the possible scenarios when carrying out this kind of memory copies with CUDA and rCUDA. As we can see, with CUDA there is only one possible scenario, depicted in Figure 5.1(a), where the GPUs are located in the same node and are interconnected by the PCIe link. On the contrary, when using rCUDA there are two possible scenarios for performing copies between remote GPUs: (i) the remote GPUs are located in the same remote node and are interconnected by the PCIe link as shown in Figure 5.1(b), and (ii) the remote GPUs are located in different remote nodes and therefore they are interconnected by the network fabric, as depicted in Figure 5.1(c).

Prior to this thesis, rCUDA already supported the scenario exposed in Figure 5.1(b). In this manner, it was possible to carry out memory copies between remote GPUs located in the same server node. However, the scenario presented in Figure 5.1(c) was not supported. In this regard, although the scenario depicted in Figure 5.1(c) may not be strictly required for many use cases, having the flexibility to use any of the GPUs in the cluster regardless of their exact location provides a large improvement in overall performance, as shown in [84]. Additionally, remote GPU virtualization frameworks aim to provide the same semantics as CUDA does. Therefore, providing support within the virtualization framework for P2P memory copies between remote GPUs located in different nodes is mandatory.

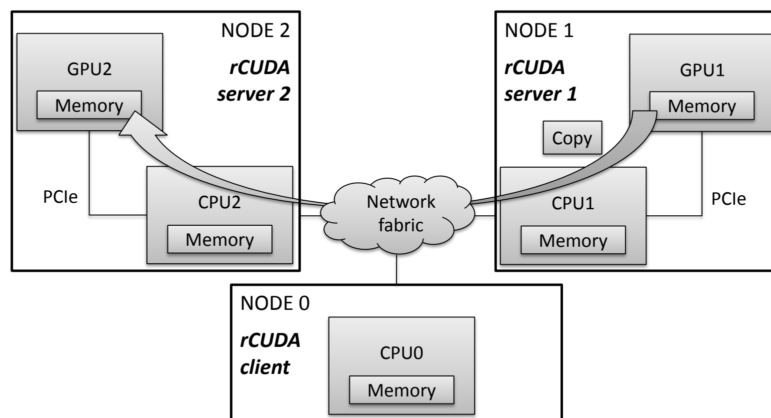
In this chapter we explore different options to implement memory copies between remote GPUs located in different nodes of the cluster. In this way, researching on different



(a) CUDA scenario.

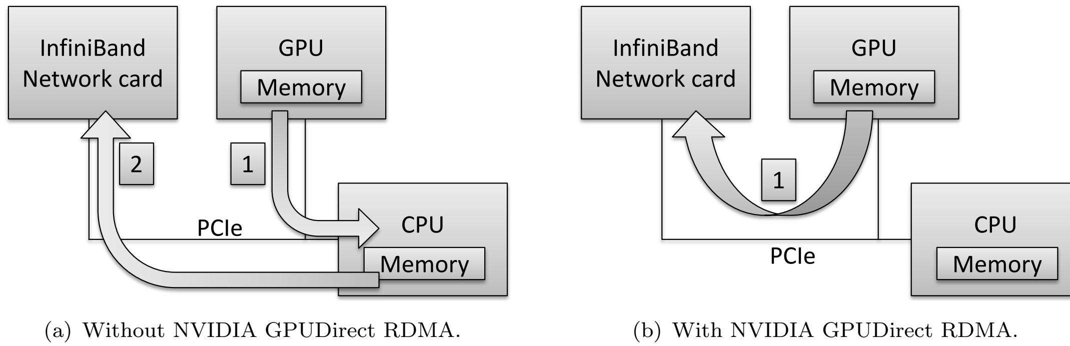


(b) rCUDA scenario 1: remote GPUs in the same server node.



(c) rCUDA scenario 2: remote GPUs in different server nodes.

Figure 5.1: Possible scenarios when carrying out P2P memory copies with CUDA and rCUDA.



(a) Without NVIDIA GPUDirect RDMA.

(b) With NVIDIA GPUDirect RDMA.

FIGURE 5.2: Scenarios with and without NVIDIA GPUDirect RDMA used with an InfiniBand network adapter.

mechanisms to efficiently implement this support is the major contribution of this chapter, where we show how we have adapted rCUDA to accomplish this purpose. To the best of our knowledge, this is the first analysis on this kind of memory copies, given that none of the existing GPU virtualization frameworks provides support for P2P memory copies between remote GPUs located in different nodes.

The rest of the chapter is organized as follows. Section 5.2 presents previous work, mainly related to the GPUDirect RDMA mechanism implemented by NVIDIA. Section 5.3 addresses the main goal of this chapter, where we explore different options to carry out memory copies between remote GPUs located in different cluster nodes. A performance comparison is carried out among the several implementation options considered in this study. A synthetic benchmark is leveraged to that end. Next, Section 5.4 provides a performance evaluation of the implemented P2P copies within the rCUDA middleware using a real application. Finally, Section 5.5 summarizes the main conclusions of this chapter.

5.2 Related Work on P2P Memory Copies

In order to efficiently copy data between the memory of GPUs located in different nodes of the cluster, NVIDIA introduced GPUDirect RDMA [85] in 2012. It is a technology that enables, by using standard features of the PCIe link, a direct path for data exchange between the GPU and a third-party peer device, such as an InfiniBand network adapter (see Figure 5.2). Support for GPUDirect RDMA was introduced by Mellanox into its InfiniBand network adapters [86] in order to provide high-speed InfiniBand networking for GPU-to-GPU communications.

Table 5.1: Summary of the bandwidth results reported for GPUDirect RDMA in the study "*Benchmarking GPUDirect RDMA on Modern Server Platforms*", by Davide Rossetti.

Path between GPU and network card	Bandwidth (GB/s)	
	Host to Host (regular RDMA)	GPU to GPU (GPUDirect RDMA)
Intra-socket (FDR)	6.1	3.7
Intra-socket (FDRx2)	12.3	3.7
Inter-socket (FDR)	6.1	1.1(TX)/0.25(RX)
PCIe switch (FDR)	6.1	5.8
PCIe switch (FDRx2)	12.3	7

Exploring the use of GPUDirect RDMA for InfiniBand networks has become very popular since it appeared. In this manner, a lot of researchers have attempted to use this technology for improving performance in different scenarios. One such example is the work by Hamidouche et al. in [87], which investigates the use of GPUDirect RDMA for improving the communication operations of OpenSHMEM, a Partitioned Global Address Space (PGAS) programming model. Another such example can be found in the work by Younge et al. in [88], where GPUDirect RDMA is one of the technologies used for running high performance molecular dynamics simulations in virtualized environments.

Other researchers have also presented improvements in the field of Message Passing Interface (MPI) communication libraries. For instance, Potluri et al. in [89] increase the efficiency of the inter-node MPI communication library MVAPICH2 [90] by using GPUDirect RDMA.

In addition to all the previous works referred above, an extensive performance analysis of NVIDA GPUDirect RDMA over InfiniBand [91] shows the real capabilities of this technology when used in modern server platforms. Table 5.1 presents a summary of the bandwidth results extracted from the referred analysis. According to the authors of that study, the testbed system used was composed of two servers with Ivy Bridge Xeon processors (E5-2690v2 at 3.00GHz), NVIDIA K40m GPUs, and Mellanox dual-port Connect-IB network adapters (each port offering FDR performance). Results in Table 5.1 show the bandwidth in GB/s attained when copying messages of 64KB from host memory to host memory, using regular RDMA and from GPU memory to GPU memory, using GPUDirect RDMA. Different scenarios are considered depending on the exact path between the GPU and the network card:

- Intra-socket: the GPU and the network adapter are connected to the same socket processor in both servers

- Inter-socket: in one of the servers (server 1), the GPU and the network adapter are connected to different sockets; in the other server (server 2), the GPU and the network adapter are attached to the same socket. TX refers to the case when copying data from GPU memory in server 1 to GPU memory in server 2. RX refers to the reverse data path
- PCIe switch: the GPU and the network card are mounted on a riser-card and they are connected by a PCIe switch. This configuration is used at both ends

Results marked with "FDR" were obtained using only 1 port of the dual-port Connect-IB network adapters, whereas the ones marked with "FDRx2" were carried out using 2 ports.

As we can see, GPUDirect RDMA introduces, in general, an important performance loss with respect to regular RDMA for copying data among host memory at both servers instead of using GPU memory. This loss in performance is clearly affected by the path traversed by data. In this manner, the best results are obtained in the PCIe switch scenario, where the bandwidth when using 1 port of the network adapter is close to the bandwidth when copying Host to Host memory (5.8GB/s vs. 6.1GB/s). In contrast, using 2 ports turns into a very clear drop in performance (from 12.3GB/s to 7GB/s). In the case of the intra-socket scenario, the bandwidth decreases to 3.7GB/s. The worst results are obtained in the inter-socket scenario, where the performance loss is more evident: 1.1GB/s and 0.25GB/s for TX and RX, respectively. The authors of [91] point out that maybe the PCIe host interface integrated into the CPU is restricting the number of in-flight transactions. Therefore, if there are not enough outstanding transactions the read bandwidth becomes latency limited.

The study in [91] also provides results for latency when copying messages of 4 bytes. In this way, the latency obtained when copying from host memory to host memory is 1.3μs, whereas the latency when moving data from GPU memory to GPU memory is 1.9μs in all the scenarios. The only exception is in the inter-socket one, where the RX copy results in a latency of 2.2μs.

Finally, the study in [91] concludes that:

- Bandwidth: according to the authors, GPUDirect RDMA is faster than a staging approach[1] for message sizes up to 400-500KB. For larger data sizes, staging to/from host memory and moving data via InfiniBand using regular RDMA probably provides better performance.

[1]A staging approach comprises copying from the source GPU memory to host memory; then copying from host memory to remote host memory using regular RDMA, instead of GPUDirect RDMA; and finally, in the remote node, copying from host memory to the destination GPU memory.

- Latency: GPUDirect RDMA provides low latency, usually below 2μs, which is better than staging to/from host memory and moving data via InfiniBand using regular RDMA. The authors state that using intermediate buffers requires either synchronous (`cudaMemcpy`) or asynchronous (`cudaMemcpyAsync`) memory copies, which take around 8μs and 9μs, respectively. Additionally, we have to add the InfiniBand host-to-host latency: 1.3μs. Thus, the expected GPU-to-GPU latency would be: 8μs (`cudaMemcpy` from GPU 1 to host 1) + 1.3μs (copy from host 1 to host 2) + 8μs (`cudaMemcpy` from host 2 to GPU 2). This is clearly larger than the latency provided by GPUDirect RDMA.

5.3 Implementing Efficient P2P Memory Copies within rCUDA

This section presents the main contribution of this chapter, which is the implementation of the memory copy mechanism devised for supporting memory copies between remote GPUs located in different nodes of the cluster. Several options are explored, such as the GPUDirect RDMA technique previously detailed.

Performance is evaluated and compared for each of the considered options. To that end, the setup used for the experiments reported in this chapter consists of three 1027GR-TRF Supermicro servers connected by an SX6025 InfiniBand switch (FDR). Each of the servers has two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz with 32 GB of DDR3 SDRAM memory at 1.6 GHz. They also include one Mellanox ConnectX-3 single-port InfiniBand adapter and one NVIDIA Tesla K20m GPU. The CentOS 6.4 operating system with Mellanox OFED 2.4-1.0.4 (InfiniBand drivers and administrative tools) and CUDA 7.5 with NVIDIA driver 352.39 are used.

The testbed servers used in our experiments are NUMA machines and therefore NUMA effects matter for the results shown in this chapter. For this reason, both the NVIDIA GPU and the InfiniBand adapter are attached to the same processor socket (processor 0). Additionally, memory buffers and processes are bound to this processor in the experiments.

In addition, and for comparison purposes, when performing P2P memory copies with CUDA, executions have been carried out using a 7047GR-TRF Supermicro server, with similar characteristics to those of the 1027GR-TRF servers mentioned before, but with two NVIDIA Tesla K20m GPUs (note that memory copies with CUDA must be carried out within the same node).

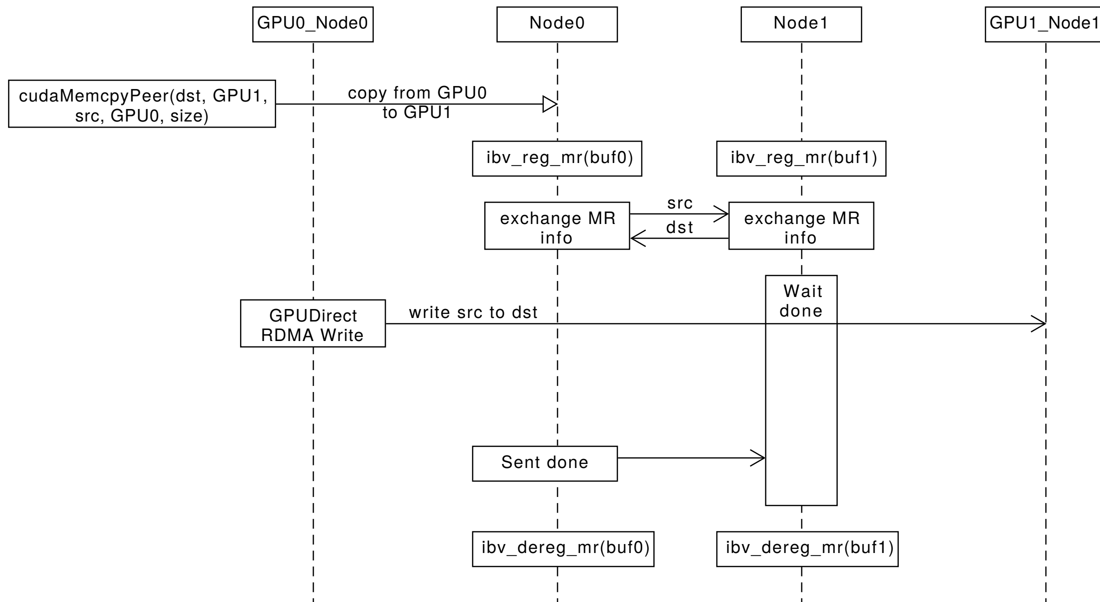


FIGURE 5.3: Sequence diagram of rCUDA version 1 for memory copies between different remote GPUs. This version uses GPUDirect RDMA to transfer the data.

5.3.1 Version 1: Using GPUDirect RDMA

The first approach considered to copy data between remote GPUs located in different nodes of the cluster is based on the use of GPUDirect RDMA. Figure 5.3 presents a sequence diagram of the proposed solution. When a request for copying memory between GPUs is received (i.e., `cudaMemcpyPeer`), the following steps are followed:

1. InfiniBand memory regions (MRs) associated with the GPU memory addresses to be copied are registered in the nodes hosting the GPUs
2. Information related to the registered MRs is exchanged between nodes
3. Data is copied from the source GPU memory to the destination one
4. When the data copy has finished, the MRs are unregistered

Figure 5.4 presents the bandwidth obtained for different transfer sizes when using this version (labeled as *rCUDA P2Pv1*) for copying data between remote GPUs. For comparison purposes, the bandwidth obtained by CUDA when transferring data between local GPUs is also depicted (labeled as *CUDA*). As we can observe, the maximum bandwidth obtained with this version of rCUDA, over 1.40GB/s, is very low. The reason is that, for each copy, we must set-up the connection (i.e., register the MRs and exchange information with the remote node). This set-up introduces a high overhead, which turns into very low performance in this test.

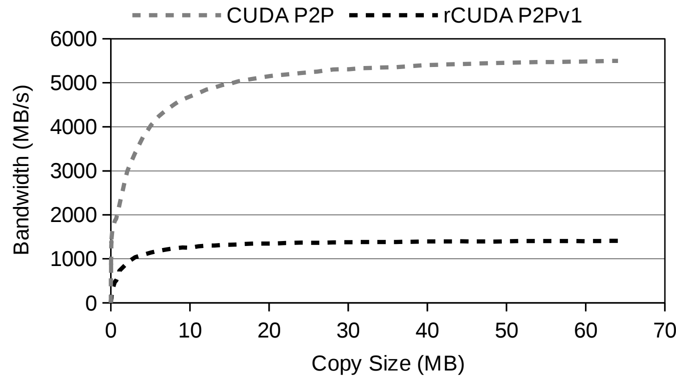


FIGURE 5.4: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and version 1 of rCUDA are shown.

5.3.2 Version 2: Pre-allocating Intermediate Buffers

To improve version 1 previously explained, next we present a new version which pre-allocates intermediate buffers at initialization to avoid the overhead introduced by registering and exchanging MRs information for each data copy. Figure 5.5 presents a sequence diagram of this approach. When a request for copying memory between GPUs is received, the following steps are followed:

1. New GPU memory buffers are allocated and InfiniBand memory regions (MRs) are registered. Information related to the registered MRs is exchanged between nodes involved in the copy. This is done only once at initialization. These buffers will be reused until the application finishes
2. Data is copied from the source GPU memory to the local intermediate buffer just allocated in the GPU in the previous step
3. Data is copied from the local intermediate buffer to the remote intermediate buffer at the remote GPU by using GPUDirect RDMA
4. Data is copied from the remote intermediate buffer to the destination GPU memory

The question that arises now is the following: given that we are using intermediate buffers, and taking into account that RDMA transfers between host memory attain higher bandwidth than with GPU memory, would it be a better choice to use host intermediate buffers instead of the GPU intermediate buffers used in this version?

To answer this question we have implemented a new version, similar to the previous one, but using host intermediate buffers. From now on, versions using GPU intermediate buffers and GPUDirect RDMA will be labeled as the version number plus the letter *A*, while versions using host intermediate buffers (and not using GPUDirect RDMA) will

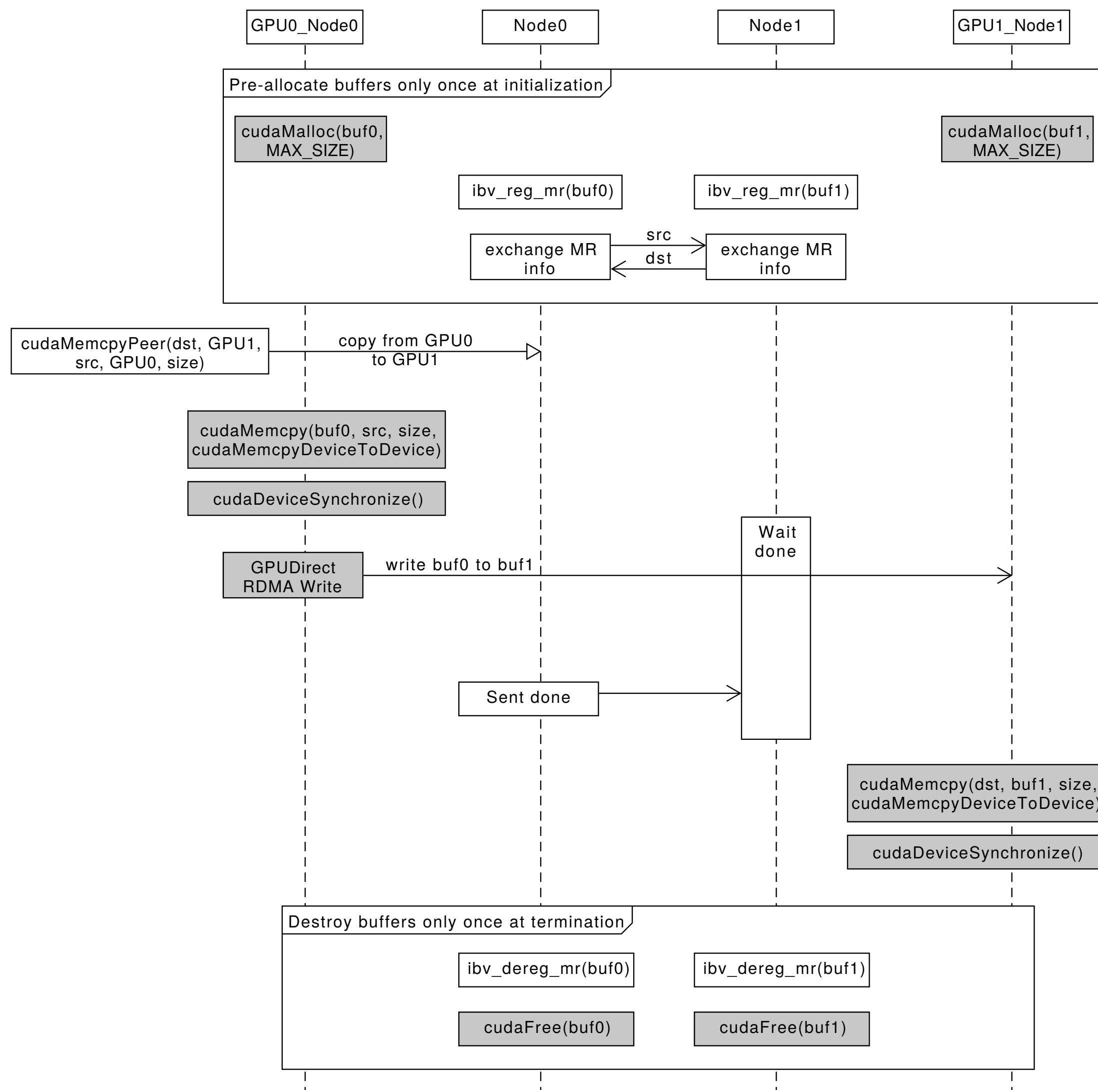


FIGURE 5.5: Sequence diagram of rCUDA version 2A for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2B in Figure 5.6. This version uses GPUDirect RDMA to transfer data, and pre-allocates intermediate buffers at initialization.

be referred to as the version number followed by the letter *B*. This is why we referred to the version 2 previously explained as version *2A* in Figure 5.5.

Figure 5.6 presents a sequence diagram of the new proposed version 2B using host intermediate buffers. Differences with respect version 2A are highlighted in gray for clarity. As it can be observed, the two versions are similar, the only difference being the intermediate buffers: in version 2A they are allocated using GPU memory, while in version 2B they are allocated using host memory.

Figure 5.7 presents the bandwidth obtained when using the new versions. Bandwidth significantly improves, achieving a maximum of 2.25GB/s when GPU buffers are used,

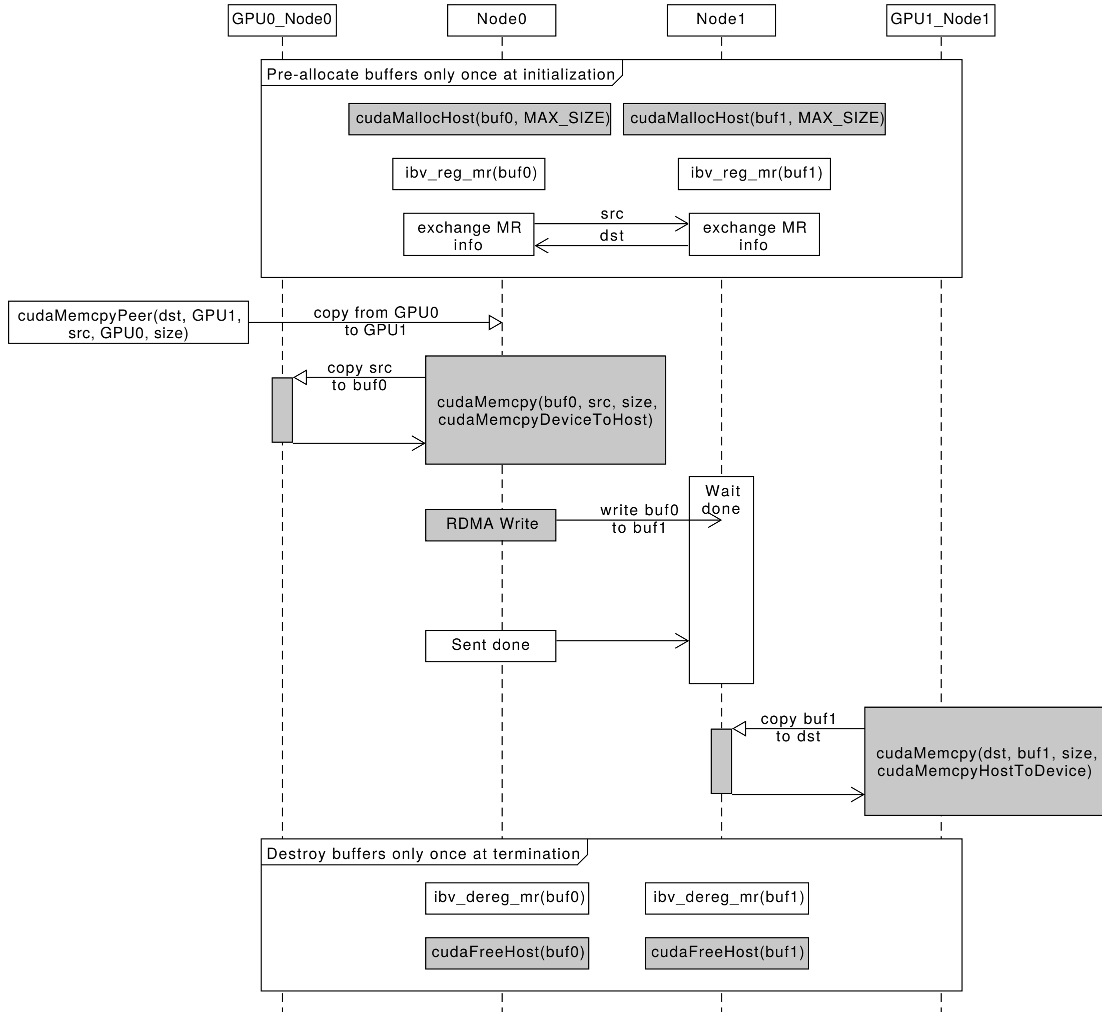


FIGURE 5.6: Sequence diagram of rCUDA version 2B for memory copies between different remote GPUs. Gray boxes denote the differences with respect to version 2A in Figure 5.5. This version uses regular RDMA to transfer data and then copy it to GPU memory. Intermediate buffers are pre-allocated at initialization.

and a maximum of 2.90GB/s when using host buffers.

Using this approach improves performance. However, we must allocate one extra buffer, either in GPU memory or in host memory, in each node involved in the data copy. The size of this buffer must be the biggest possible copy size performed by the application. For instance, in our bandwidth test, the buffer had a size equal to 64MB, the maximum copy size in our test. Of course, this approach is only valid for testing purposes, and cannot be used with real applications because it would require knowing in advance the maximum data size transferred by the application. Additionally, even if this size were known, this approach requires to use a lot of memory for the intermediate buffers.

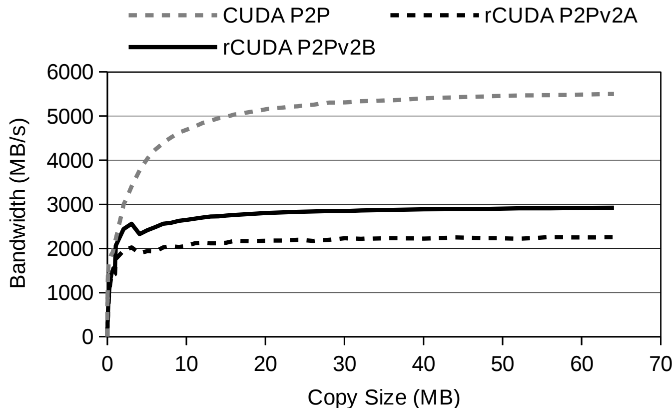


FIGURE 5.7: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 2A and 2B of rCUDA are shown.

5.3.3 Version 3: Using Multiple Intermediate Buffers

In order to address the concerns commented in version 2, we next present a new version which, instead of using one large intermediate buffer in each node involved in the copy, uses multiple smaller intermediate buffers. Thus, the whole amount of data to be copied is split into several chunks of the size of the smaller intermediate buffers, and the copy is performed following a pipelined approach, overlapping copies in the different stages:

1. Data chunk *i-1* is copied from the source GPU memory to the local intermediate buffer *i-1*
2. Data chunk *i* is copied from the local intermediate buffer *i* to the remote intermediate buffer *i*
3. Data chunk *i+1* is copied from the remote intermediate buffer *i+1* to the destination GPU memory
4. Steps 1, 2 and 3 are overlapped and repeated until all the data has been copied

Figure 5.8 and Figure 5.9 present sequence diagrams of the new proposed versions 3A and 3B, respectively. As commented, version 3A uses GPU intermediate buffers, whereas version 3B uses host intermediate buffers. Again, differences between each version are highlighted in gray for clarity.

Figure 5.10 presents the bandwidth obtained when using these new versions (labeled as *rCUDA P2Pv3A* and *rCUDA P2Pv3B*, respectively). We can see that bandwidth has considerably increased in both versions, obtaining maximum values around 2.60GB/s and 5.40GB/s for versions 3A and 3B, respectively. In the latter case, version 3B, performance is almost the same as the one obtained by CUDA with local GPUs.

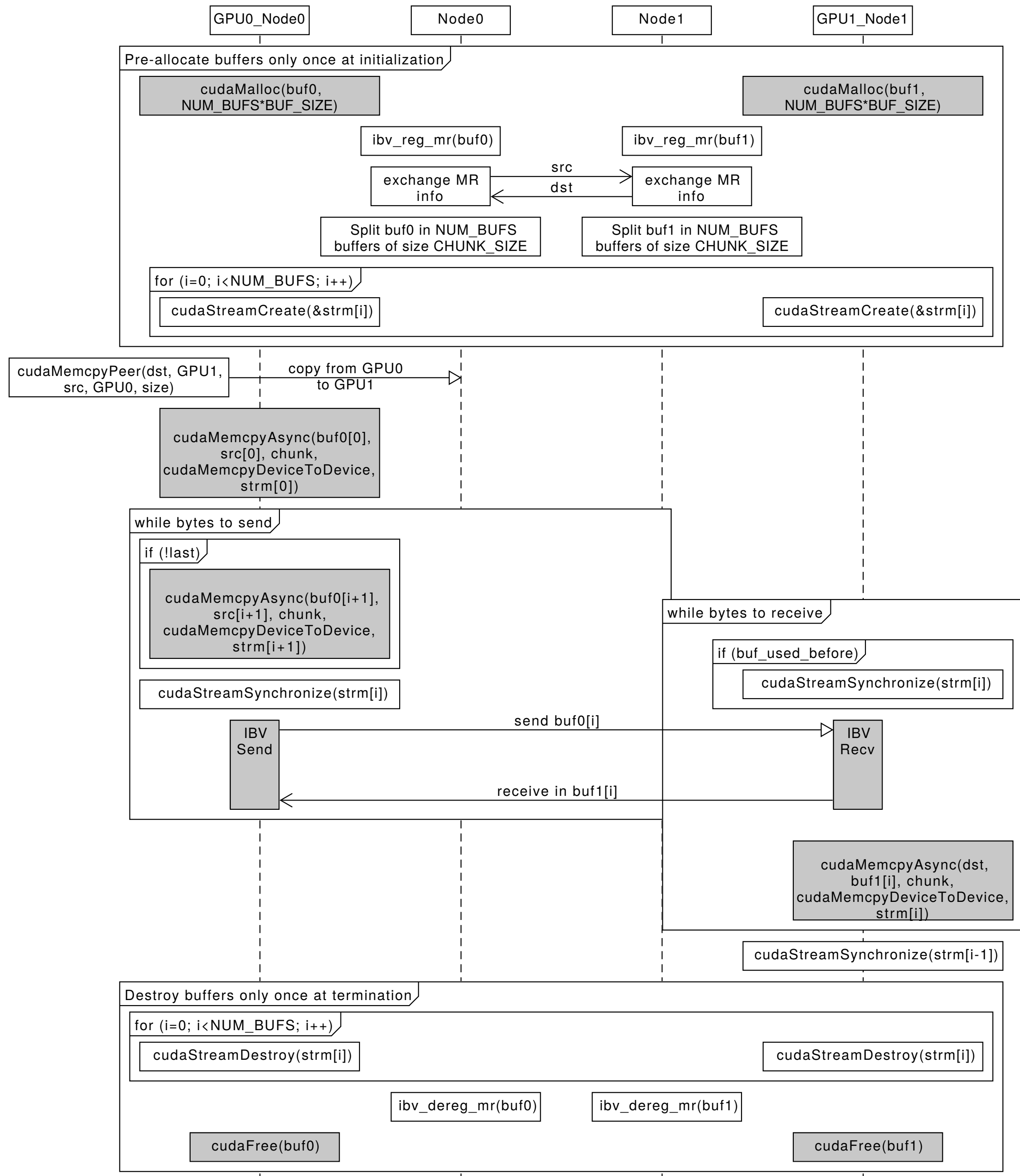


FIGURE 5.8: Sequence diagram of rCUDA version 3A for memory copies between different remote GPUs. Gray boxes depict differences with respect to version 3B shown in Figure 5.9. This version is similar to version 2A, but uses multiple intermediate buffers at the GPU.

5.3.4 Version 4: Adaptive Intermediate Buffer Size

In the previous version we have used a fixed number of intermediate buffers, all of them with the same size. However, the optimal amount of buffers and the optimal buffer size probably depends on the actual transfer size. To analyze the influence of these two factors we have run the same bandwidth test as in Figure 5.10 with versions 3A and 3B, but varying the number of intermediate buffers: 2, 4, 8, 16, 32 and 64. For each number

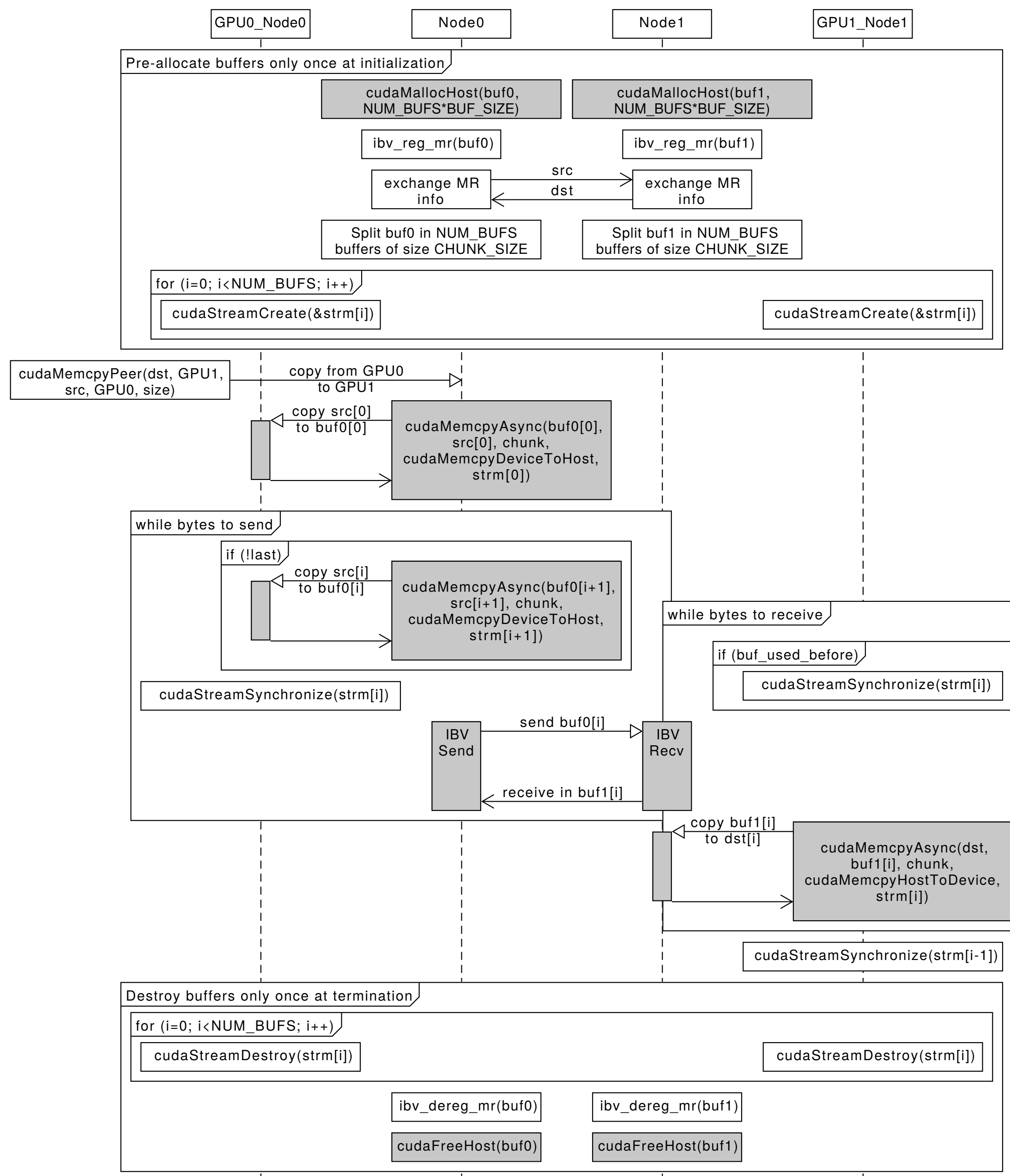


FIGURE 5.9: Sequence diagram of rCUDA version 3B for memory copies between different remote GPUs. Gray boxes refer to differences with respect to version to 3A shown in Figure 5.8. This version is similar to version 2B, but uses multiple intermediate buffers at host memory.

of buffers, we have also run the test with different buffer sizes: 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, and 8MB.

In the case of using GPU intermediate buffers (version 3A), the results of the analysis point to select the following values as the optimal ones:

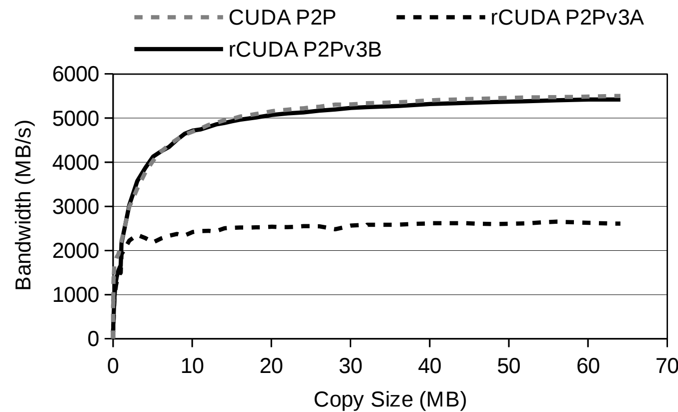


FIGURE 5.10: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 3A and 3B of rCUDA are shown.

- Copy sizes over 4MB: the optimal number and size of intermediate buffers is 16 buffers of 256KB
- Copy sizes up to 4MB: the optimal number and size of intermediate buffers is 2 buffers of 512KB

In the case of using host intermediate buffers (version 3B), this analysis reveals the following results:

- Copy sizes over 14MB: the optimal number and size of intermediate buffers is 4 buffers of 1MB
- Copy sizes between 4MB and 14MB: the optimal number and size of intermediate buffers is 4 buffers of 512KB
- Copy sizes between 600KB and 4MB: the optimal number and size of intermediate buffers is 8 buffers of 256KB
- Copy sizes below 600KB: there is no apparent optimal value for the two factors under analysis. We decided to select as the optimal value 8 buffers of 128KB, following the trend of previous values

With the results of this analysis we have implemented a new version which automatically varies the number and size of intermediate buffers depending on the actual size of the data to be copied. Following our version nomenclature, we have named these new versions as 4A and 4B. Figure 5.11 shows the bandwidth results for these new versions, showing that version 4A improves over its predecessor, version 3A, for copy sizes up to 4MB. For larger sizes, the results are similar to those of the previous version. With

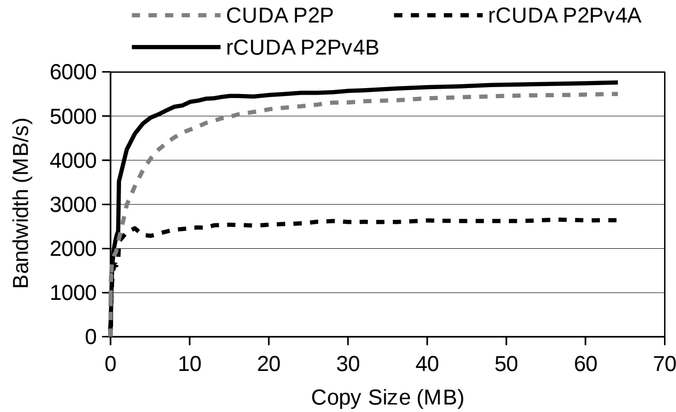


FIGURE 5.11: Bandwidth obtained for different transfer sizes when copying data between remote GPUs. Results from CUDA P2P and versions 4A and 4B of rCUDA are shown.

regard to version 4B, we can see that it clearly outperforms version 3B, regardless of the data size of the copy.

It is also noteworthy that version 4B obtains, in general, better performance than CUDA. For copy sizes up to 300KB, CUDA achieves a higher bandwidth, but for larger copy sizes, rCUDA attains better results (up to 1.42x speed-up). The explanation for this higher bandwidth of rCUDA can be found in Figure 5.12. This figure shows the bandwidth attained by the different CUDA and InfiniBand memcpy functions involved in the analysis presented in this section. When using CUDA, a single call

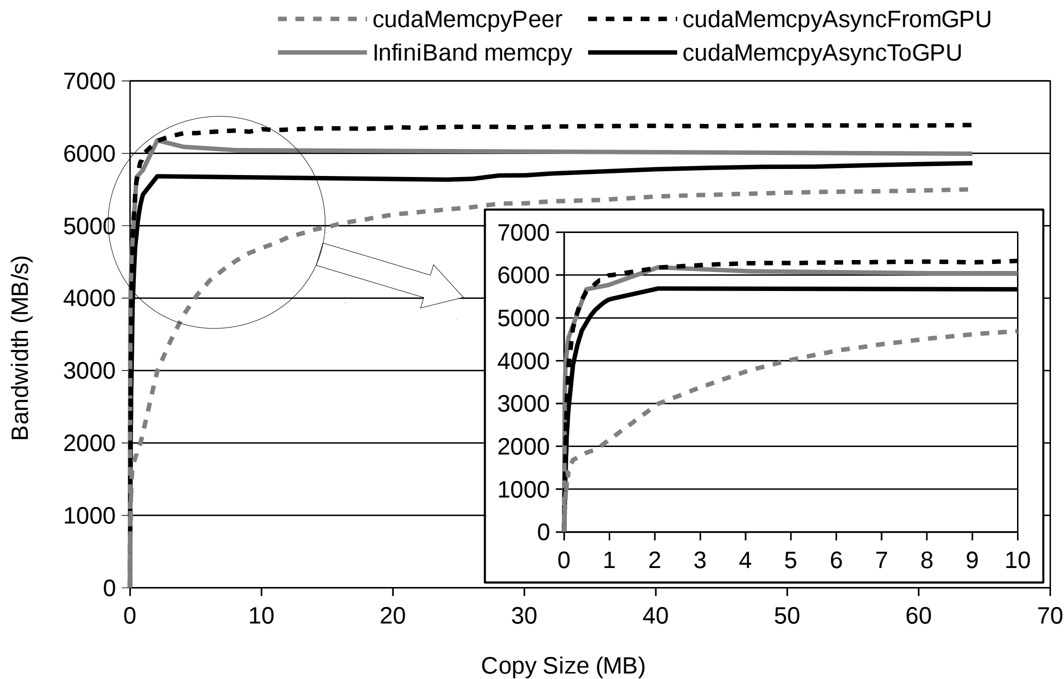


FIGURE 5.12: Bandwidth comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter.

Table 5.2: Summary of the different rCUDA versions implemented for supporting memory copies between remote GPUs.

	rCUDA version						
Feature	1	2A	2B	3A	3B	4A	4B
GPUDirect RDMA	x	x		x		x	
Pre-allocated buffers		x	x	x	x	x	x
Multiple buffers				x	x	x	x
Adaptive buffer size						x	x

to `cudaMemcpyPeer` is done to copy the whole bunch of data between the two GPUs. The PCIe bus is used to move data from one GPU to the other. On the contrary, when using rCUDA, the data to be copied is split into smaller chunks of data and the movement of the several chunks is overlapped by using several simultaneous calls to: (1) `cudaMemcpyAsync` from GPU to host memory, (2) InfiniBand memory copy from local host memory to remote host memory, and (3) `cudaMemcpyAsync` from host to GPU memory. As we have previously commented, a pipelined approach is followed. Therefore, the maximum bandwidth that can be achieved with this technique is the minimum bandwidth of each individual stage. In this case, the minimum bandwidth of the three mentioned calls is the one obtained by `cudaMemcpyAsync` from host to GPU memory (labeled as *cudaMemcpyAsyncToGPU* in Figure 5.12). As we can see, this bandwidth is larger than the one obtained by `cudaMemcpyPeer`, which explains why rCUDA is attaining more bandwidth than CUDA.

5.3.5 Latency Analysis

In previous sections we have thoroughly analyzed the bandwidth attained by each P2P version of rCUDA. Next, we analyze latency. As a summary, Table 5.2 presents the most important features of the different rCUDA versions implemented for supporting memory copies between remote GPUs.

Figure 5.13 presents the latency obtained when using CUDA and different versions of rCUDA to copy data between remote GPUs located in different nodes of the cluster. For clarity, results from versions 1 and 2 are not shown. It can be seen in the figure that the best results are obtained by CUDA, with a minimum latency of 22μs. Regarding the different versions of rCUDA, version 4A, using GPUDirect RDMA and all the improvements analyzed in previous sections, seems to achieve the best results, with a minimum latency of 72μs. Version 4B, not using GPUDirect RDMA, presents a minimum latency of 78μs. Version 4A is more stable than version 4B and presents, in

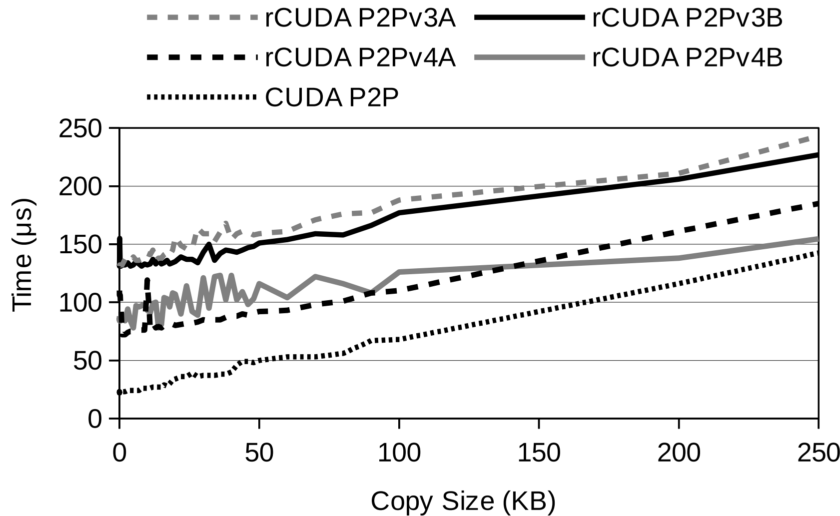


FIGURE 5.13: Latency obtained for transfer sizes up to 200KB when copying data between remote GPUs. Results from CUDA and different versions of rCUDA are shown.

general, the best latency for copy sizes up to 200KB. Finally, versions 3A and 3B present a higher latency, both with minimum values of 131μs.

In the bandwidth analysis presented in previous sections we have seen that rCUDA obtained, in general, better results than CUDA, the only exception being copy sizes smaller than 300KB. The latency results match those conclusions, given that CUDA presents lower latency than rCUDA for that range of data copy sizes. In previous sections, we explained that rCUDA achieved, in general, higher bandwidth than CUDA because the internal functions involved in the memory copy were different and also presented different performance, in addition to follow a pipelined approach. This better performance was shown in Figure 5.12. Now the question is whether the reason for rCUDA presenting a higher latency than CUDA is also because of this.

To answer this question, Figure 5.14 presents a latency comparison of the different CUDA and InfiniBand memcpy functions involved in the memory copy between GPUs. It can be seen that the `cudaMemcpyPeer` function used by CUDA to copy data between local GPUs achieves the highest latency. In the case of copying 4 bytes of data, this latency is equal to 22μs. On the contrary, the operations involved in data copies with rCUDA present, individually, a smaller latency. For instance, in the case of copying 4 bytes of data, the `cudaMemcpyAsync` call to copy data from the GPU to host memory at the source node requires 9μs. At the destination node, the `cudaMemcpyAsync` call to copy data from host memory to GPU memory requires 17μs. Finally, the latency for moving those 4 bytes from one node to the other across the InfiniBand fabric is 1.3μs. Notice, however, that the data copy with rCUDA follows a pipelined approach. Therefore, the total latency of the P2P data copy is the result of the addition of the

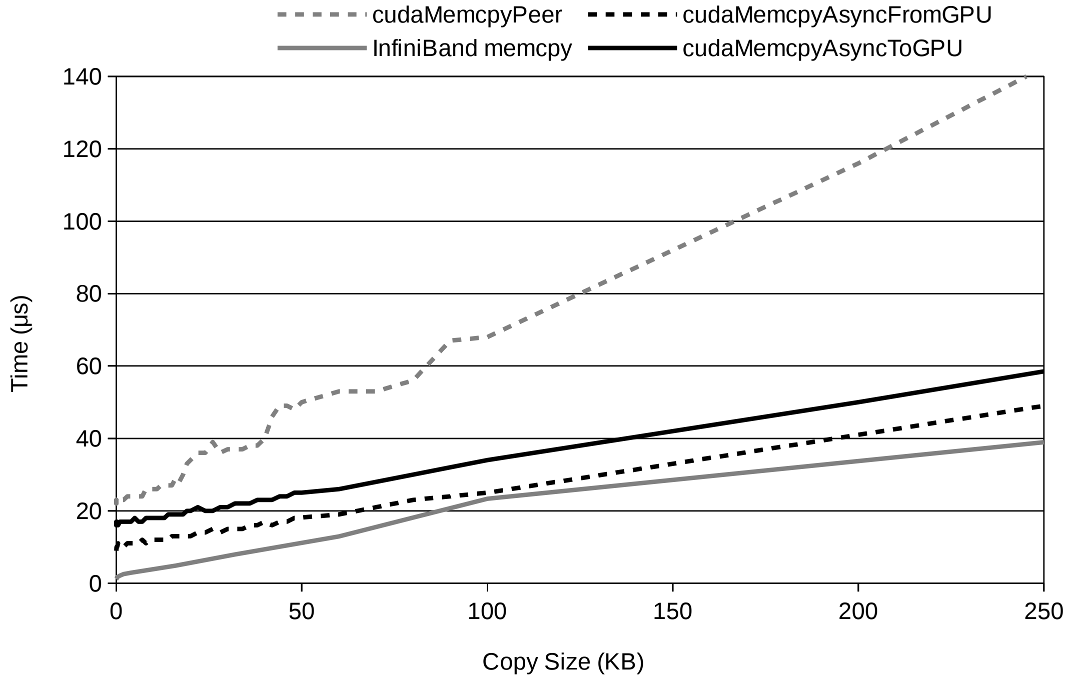


FIGURE 5.14: Latency comparison of the different CUDA and InfiniBand memcpy functions used in the analysis shown in this chapter.

TABLE 5.3: Minimum latency achieved by the different rCUDA versions implemented for supporting memory copies between remote GPUs.

	rCUDA version						
	1	2A	2B	3A	3B	4A	4B
Latency (μs)	596	136	131	131	131	72	78

previous values, turning out a latency of 27.3μs.. This value is, however, smaller than the latency measurement provided in Table 5.3. The reason for the difference among both values is due to the management required to run our highly tunned pipeline, what seems to penalize our approach in terms of latency. In this regard, one could think about using more simple approaches for small copy sizes in order to improve latency. However, notice that we have already tried simpler proposals, such as versions 1 and 2, and results are worse than the ones obtained with versions presenting higher complexity, such as versions 3 and 4 (see Table 5.3).

5.3.6 Final Version: Hybrid Approach

After analyzing different versions in the previous sections, we conclude that the optimal version for the P2P implementation within rCUDA would be a hybrid approach combining versions 4A and 4B. Thus, for small copy sizes up to 200KB, the best

results are achieved by version 4A, using GPUDirect RDMA. For larger copy sizes, the best results are obtained with version 4B, not using GPUDirect RDMA. We have implemented one last version featuring this hybrid approach. This new version will be the one used from now on in the rest of the chapter. Bandwidth results for this new version are omitted for brevity, given that the results are very similar to the ones already shown: bandwidth of version 4B in Figure 5.11, and latency of version 4A in Figure 5.13.

5.4 Experiments with a Real Application

In previous sections we have assessed the performance of the implemented P2P memory copy mechanism by using synthetic tests. This section presents the experiments carried out with a real application that makes use of the P2P copies provided by CUDA.

The selected application belongs to the area of network analysis, where the clustering coefficient and the transitivity ratio are concepts often used, creating the need for fast practical algorithms devoted to count triangles in large graphs. Furthermore, these algorithms can be programmed to be executed in GPUs so that total execution time is reduced. Therefore, for the performance evaluation in this section we have used an application for counting triangles in large graphs on GPUs [92]. From now on, we will refer to this application as TRICO (triangle count). 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. Additionally, this application performs P2P data copies among the GPUs involved in its execution.

The setup used for the experiments reported in this section is the same as the one presented in Section 5.3. In the case of the experiments with CUDA, and given that the application can make use of several GPUs, the scenario used in the tests is the one depicted in Figure 5.1(a) of Section 5.1. In particular, two GPUs will be provided to the application. Regarding the experiments with rCUDA, the scenario is the one depicted in Figure 5.1(c) of Section 5.1. Therefore, the application is running in a node without GPUs, and it is using two remote GPUs located in two different server nodes.

Seven different graphs have been leveraged for the experiments in this section. These graphs are described in Table 5.4. The largest one, referred to as graph number 7, is a 180 million edge graph containing 8.8 thousand millions of triangles. We have used this graph in order to characterize the TRICO application. Figure 5.15 shows the CPU and GPU utilization when executing the application with this largest graph in the CUDA scenario (local GPUs). It can be seen that the CPU presents a high utilization, almost

Table 5.4: Graphs used in the experiments with the TRICO application.

Graph	Nodes	Edges	Triangles
1	65,535	4,912,142	118,811,321
2	131,068	10,227,970	287,593,439
3	540,486	30,491,458	444,095,058
4	434,102	32,073,440	872,040,567
5	524,287	43,561,574	1,625,559,121
6	1,048,576	89,238,804	3,803,609,518
7	2,097,152	182,081,864	8,815,649,682

100% during all the execution of TRICO. In the case of the GPUs, GPU0 starts being used after 2 seconds of execution, while GPU1 starts working after almost 4 seconds. Then, both GPUs also present a high usage, close to the 100% until near the end of the execution of the application. In summary, Figure 5.15 shows that the TRICO application makes an intensive use of the available GPUs as well as the CPU.

Figure 5.16 shows the execution time and speed-up when running the TRICO application using the different graphs described in Table 5.4. Results are the average of 10 repetitions. The primary Y-axis of the figure shows the execution time of the TRICO application when using CUDA and rCUDA. As we can observe, the application runs slightly faster with rCUDA than with CUDA for all the graphs evaluated. The secondary Y-axis shows the exact speed-up of rCUDA with respect to CUDA. On average, the speed-up is 1.13. These results showing that the application achieves better performance with rCUDA than with CUDA require a deeper analysis.

One possible explanation to the better results of rCUDA with respect to CUDA could be that rCUDA provides a better bandwidth when copying data between GPUs, as

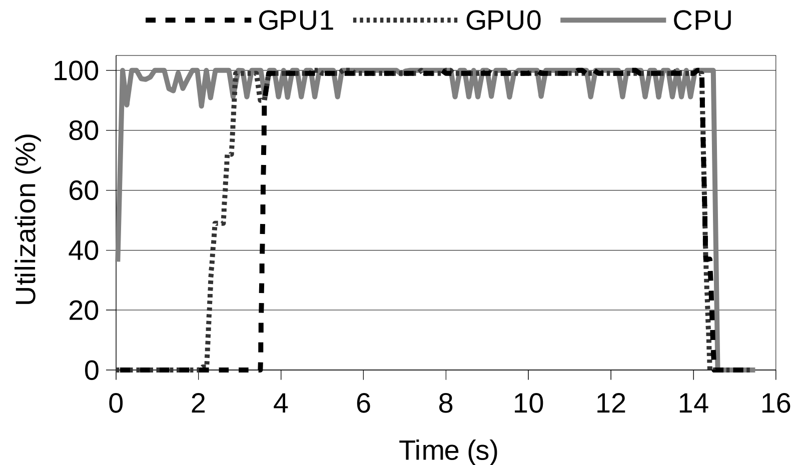


Figure 5.15: CPU and GPU utilization of TRICO when running the largest graph.

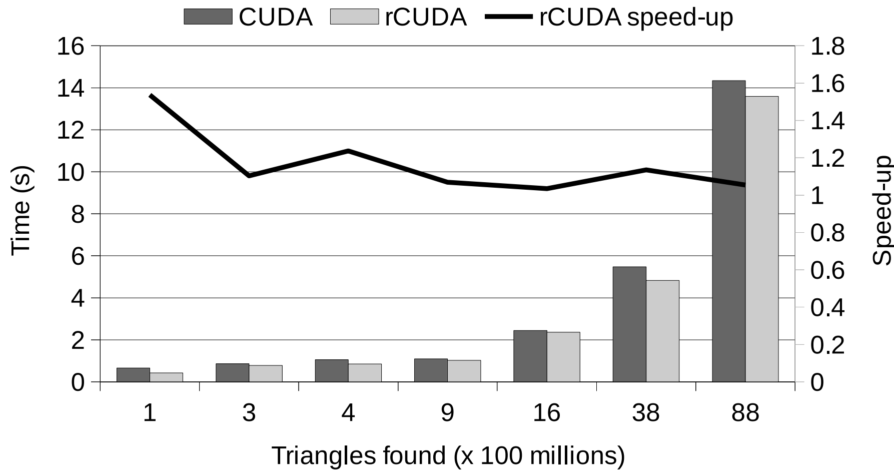


FIGURE 5.16: Primary Y-axis shows TRICO application execution time using CUDA and rCUDA. Secondary Y-axis presents the speed-up of rCUDA with respect to CUDA.

shown in Figure 5.11. Therefore, that higher bandwidth is causing the difference in the execution time with respect to CUDA. To find out whether this is the reason for the better results when using rCUDA, we have profiled the TRICO application in order to know the amount of calls to the functions that perform P2P data copies. Table 5.5 presents such profiling, showing for each CUDA function used in the application the total time employed by the application in that function as well as the number of calls done to the it. The average, minimum and maximum time per call is also displayed. Surprisingly, it can be seen in Table 5.5 that there are only two calls to functions involving copies between GPUs (i.e., calls to the `cudaMemcpyPeer` function, highlighted in bold in Table 5.5). Furthermore, the time used by these calls is 34.990ms, what only accounts for 0.27% of the total execution time of the application. This small percentage of time does not seem to be the reason for the better results of rCUDA over CUDA. However, let us further examine these calls to the `cudaMemcpyPeer` function.

In order to continue with our analysis of the better execution times achieved by the TRICO application when using rCUDA instead of CUDA, we have next measured the time employed in the P2P copies between GPUs with rCUDA (notice that the times depicted in Table 5.5 were obtained with CUDA in the scenario shown in Figure 5.1(a)). In this regard, we have measured the time required to carry out these P2P copies with rCUDA in the scenario presented in Figure 5.1(c). Table 5.6 compares the time results obtained with CUDA and rCUDA. The table also shows the amount of megabytes copied during each call to the `cudaMemcpyPeer` function.

Interestingly, it can be seen that the first call to the `cudaMemcpyPeer` function takes more time with rCUDA than with CUDA, despite that the size of the copy, 695MB, is large enough to expose the better performance of rCUDA shown in the bandwidth

Table 5.5: Profiling of the TRICO application when running the largest graph with CUDA.

CUDA function	% Time	Time	Calls	Average	Min	Max
cudaDeviceSynchronize	87.05%	11.49s	8	1.433s	5.19us	11.42s
cudaMemcpy	4.41%	582.44ms	6	97.07ms	24.10us	531.04ms
cudaFree	4.39%	579.42ms	18	32.19ms	151.75us	219.06ms
cudaMemcpyAsync	3.81%	502.65ms	2	251.33ms	28.79us	502.63ms
cudaMemcpyPeer	0.27%	34.99ms	2	17.49ms	1.55ms	33.43ms
cudaMalloc	0.05%	6.13ms	16	383.17us	146.45us	1.19ms
cudaGetDeviceProperties	0.01%	1.95ms	4	489.61us	462.27us	543.63us
cuDeviceGetAttribute	0.01%	983.12us	166	5.92us	160ns	225.88us
cudaLaunch	0.01%	949.26us	65	14.60us	9.61us	61.66us
cudaFuncGetAttributes	0.00%	140.81us	41	3.43us	2.55us	13.43us
cuDeviceTotalMem	0.00%	109.11us	2	54.55us	53.71us	55.39us
cuDeviceGetName	0.00%	98.33us	2	49.16us	44.18us	54.15us
cudaSetupArgument	0.00%	58.70us	278	211ns	177ns	693ns
cudaGetDeviceCount	0.00%	38.47us	1	38.47us	38.47us	38.47us
cudaSetDevice	0.00%	36.23us	14	2.58us	833ns	11.89us
cudaConfigureCall	0.00%	24.50us	65	377ns	239ns	1.85us
cudaGetDevice	0.00%	18.25us	26	702ns	325ns	1.73us
cudaFuncSetCacheConfig	0.00%	3.77us	2	1.88us	1.46us	2.31us
cuDeviceGetCount	0.00%	3.61us	2	1.80us	447ns	3.16us
cudaDeviceGetAttribute	0.00%	3.25us	3	1.08us	701ns	1.69us
cuDeviceGet	0.00%	1.43us	4	357ns	202ns	498ns

Table 5.6: Time employed in the copies between GPUs by CUDA and rCUDA when running the TRICO application with the largest graph.

		Time	
CUDA function	Size	CUDA	rCUDA
1st cudaMemcpyPeer	695MB	33.433ms	125.225ms
2nd cudaMemcpyPeer	8MB	1.557ms	1.349ms

experiments (see Figure 5.11). Additionally, the time required by rCUDA to perform the data copy is much larger than the time required by CUDA. This difference does not match the bandwidth results shown in previous sections. On the other hand, in the case of the second call to the `cudaMemcpyPeer` function, the time required to complete the copy with CUDA and rCUDA do not follow the trend of the first copy. On the contrary, in this second copy the time results in Table 5.6 match the trend observed in the experiments of previous sections where rCUDA presented a better performance than CUDA. Thus, the question at this point is what happened with the first P2P copy shown in Table 5.6.

After a deeper analysis to find out the reason for this behavior when rCUDA is used, we found out that, in the first call to the `cudaMemcpyPeer` function, rCUDA needs to carry out the necessary initialization of the P2P mechanism to copy data between the remote

TABLE 5.7: Time employed by CUDA and rCUDA in two consecutive copies of the same size between GPUs.

CUDA function	Size	Time	
		CUDA	rCUDA
1st cudaMemcpyPeer	695MB	33.433ms	125.225ms
2nd cudaMemcpyPeer	695MB	123.758ms	116.951ms

GPUs (i.e., creating the InfiniBand connections between the remote nodes where the remote GPUs are located and pre-allocating intermediate buffers for RDMA transfers). On the contrary, in the second call to the `cudaMemcpyPeer` function, this start-up is already done and therefore rCUDA achieves the higher bandwidth already shown in Figure 5.11.

To show that this was the reason, a synthetic test performing two consecutive copies of the same size was carried out with CUDA and rCUDA. Table 5.7 presents the results of this experiment. Regarding rCUDA, it can be seen that, as in the previous case shown in Table 5.6, the first copy needs more time than the second one because it initializes the P2P memory copy mechanism between the remote GPUs. Additionally, when considering the second copy (notice that this time the second copy has the same size as the first one), it can be seen that it is completed 8ms earlier with rCUDA than with CUDA. This result matches the conclusions obtained in previous sections where rCUDA obtained better performance as shown in Figure 5.11.

Interestingly, when CUDA is used, it can be seen that the second copy takes longer to be completed than the first one, despite they transfer the same amount of data. In particular, the second copy requires 92ms more than the first copy. This result was unexpected because it was assumed that both copies would last the same amount of time given that they are transferring the same amount of data. The explanation to this behavior was found after reviewing the CUDA documentation of the function `cudaMemcpyPeer` [93]. According to its documentation, this function is asynchronous with respect to the host. This means that, with CUDA, this call does not return the control to the application once the copy is completed but control is returned much earlier and the copy will be performed asynchronously with respect to the non-GPU part of the application. To ensure that the copy is finished, the application needs to add a subsequent synchronization point (such as `cudaDeviceSynchronize`).

In order to further analyze this behavior, we modified the synthetic test program used in Table 5.7 so that a call to `cudaDeviceSynchronize` is performed after each call to the `cudaMemcpyPeer` function. In this way we force that the first P2P copy is completed

Table 5.8: Time employed by CUDA and rCUDA in two consecutive copies of the same size between GPUs. Synchronization calls are leveraged in order to get accurate measurements.

CUDA function	Size	Time	
		CUDA	rCUDA
1st cudaMemcpyPeer	695MB	123.869ms	125.336ms
2nd cudaMemcpyPeer	695MB	123.745ms	117.062ms

before the execution of the second P2P copy begins. Table 5.8 shows the new results. It can be seen that now both calls to the `cudaMemcpyPeer` function last the same time when CUDA is used. Additionally, the time required by each call when rCUDA is used is not changed. The reason why times are kept the same with rCUDA is that, as previously explained in Section 5.3.3, the P2P copy with rCUDA is performed following a pipelined approach, overlapping multiple smaller copies in the different stages of the pipeline:

- Stage 1: data is copied from the source GPU memory to the local intermediate buffer at host memory
- Stage 2: data is copied from the local intermediate buffer at host memory in the source node to the remote intermediate buffer at host memory in the destination node
- Stage 3: data is copied from the remote intermediate buffer at host memory to the destination GPU memory

In order to smoothly run this pipeline, before starting the copy in stage 2, rCUDA needs to make sure that the copy from the GPU memory to the intermediate buffer has finished. For that purpose, a synchronization point is added. That is the reason why both copies in Table 5.8 take the same time with rCUDA as in Table 5.7, because they already included the synchronization points.

As a summary, we started our analysis about the better performance of the TRICO application when it makes use of rCUDA instead of CUDA (shown in Figure 5.16) by presuming that one possible explanation to that behavior could be the better bandwidth achieved by rCUDA for the P2P copies (shown in Figure 5.11). However, after our analysis we conclude that, not only this is not the explanation for the better performance, but our implementation of the P2P copies also shows a potential overhead of rCUDA with respect to CUDA due to the necessary initialization of the P2P mechanism to copy data between GPUs located in different server nodes. This initialization is performed at the first call to a CUDA function that performs P2P copies. However,

Table 5.9: Breakdown of the TRICO application when running the largest graph with CUDA and rCUDA. Stages marked with * include synchronization points.

Stage	Time (ms)		Difference
	CUDA	rCUDA	(CUDA-rCUDA)
Read file	853.0	862.5	-9.5
Pre-initialize context for all GPUs	365.0	277.5	87.5
Memcpy edges from host to GPUs*	479.0	469.0	10.0
Calculate number of vertices	12.0	10.0	2.0
Sort edges*	538.0	459.5	78.5
Calculate node array for 2-way zipped edges*	23.0	18.0	5.0
Remove backward edges*	185.0	179.2	5.8
Unzip edges*	27.0	24.83	2.2
Calculate node array for 1-way unzipped edges*	10.0	9.0	1.0
Calculate triangles on multi GPU**	11541.0	11033.8	507.2
TOTAL SUM	14033.0	13343.3	689.7

the overhead associated with this initialization could be avoided if it were carried out during application start up. But in this case some memory could be wasted. We will investigate further during future work on this matter.

Let us come back to our original question: why does the TRICO application perform better with rCUDA than with CUDA? In this regard, Table 5.7 has shown that control after P2P copies is returned to the application earlier with CUDA than with rCUDA. On the other hand, kernels take the same time to be completed regardless of being executed in a local GPU with CUDA or in a remote GPU with rCUDA. As a consequence, the application should perform better with CUDA than with rCUDA, which is not the case. Therefore, where during the execution of the application with CUDA, the time saved by the P2P copies is later lost?

In order to find the answer to this question, there must be in the application source code one or more CUDA functions that perform better with rCUDA than with CUDA. To see whether this is the case, Table 5.9 presents a breakdown of the TRICO application when running the largest graph with CUDA and with rCUDA. A detailed explanation of each stage can be found in [92]. As additional information, stages marked with the symbol * indicate that there is a synchronization point in that stage. In the case of the stage *"Calculate triangles on multi GPU"*, it includes two synchronization points, one per GPU. Time required by each stage to be completed is shown in the table.

It can be seen in the table that most stages are completed faster with rCUDA than with CUDA. Actually, those stages presenting significant improvement when using rCUDA include synchronization points[2]. Remember that in Table 5.5 we showed that the major

[2] The only exception is the stage *"Pre-initialize context for all GPUs"*. In this case the improvement is due to the fact that the rCUDA server pre-initializes contexts on the GPUs at start-up. In this manner, the rCUDA server is waiting for requests from client applications with the context already

part of the execution time of the TRICO application was spent in synchronization points (i.e., 87.05% of the time was used by 8 calls to function `cudaDeviceSynchronize`). Now Table 5.9 shows that the improvement of rCUDA with respect to CUDA in those stages containing those 8 synchronization points is 609ms. These savings, together with the time saved in the context initialization in stage *"Pre-initialize context for all GPUs"*, explains the total time saved when using rCUDA. The reason for this large time difference in the synchronization points lies in the internal algorithm used in the rCUDA middleware to determine the finalization of the CUDA tasks, as shown in Section 3.3, which performs better than the method used within CUDA.

As a summary of this section, the TRICO application was considered as use-case of an application using P2P copies. Execution times show that the application runs faster with rCUDA than with CUDA, being the most immediate reason the better bandwidth achieved by rCUDA for P2P copies, as shown in the previous section. However, although rCUDA attains more bandwidth than CUDA, finally the reason for the better performance of the TRICO application with rCUDA was the better management that this middleware makes for determining CUDA task finalization. In any case, the implementation presented in this chapter for the P2P data copies does not penalize applications, although it could be still improved, as explained in next section.

5.5 Conclusions

This chapter has presented an efficient memory copy mechanism devised for enhancing the rCUDA framework with support for memory copies between remote GPUs located in different nodes of the cluster. The finally proposed mechanism is the result of a thorough analysis of several implementation options, which have been explored and whose performance has been evaluated and compared with each other. The proposed mechanism consists in the use of multiple intermediate buffers following a pipelined approach, where several memory copies are overlapped in different stages. In this regard, we began our exploration with the GPUDirect RDMA technique proposed by NVIDIA (which initially was thought to the best implementation choice) but we have finally showed that a pipelined approach presents better performance. Notice that using a pipeline approach for communicating data is not new. Actually, this approach can be found in many areas such as computer networks, high performance on-chip and off-chip interconnects, etc. However, although the concept of the mechanism is basically the same in all these areas, the exact implementation is very different. In fact, in none of these areas the research emphasis is put on the pipeline mechanism but on its implementation.

initialized. On the contrary, applications running with CUDA must spend some time in the context pre-initialization.

Chapter 6

schedGPU: Fine-Grain Dynamic and Adaptive Scheduling for GPUs

GPUs in High-Performance Computing systems remain under-utilized due to the unavailability of schedulers that can safely schedule multiple applications to share the same GPU. The research reported in this chapter is motivated to improve the utilization of GPUs by proposing a framework, referred to as schedGPU, to facilitate intra-node GPU co-scheduling such that a GPU can be safely shared among multiple applications by taking memory constraints into account. Two approaches, namely a client-server and a shared memory approach are explored. However, the shared memory approach is more suitable due to lower overheads when compared to the former approach. Four policies are proposed in schedGPU to handle applications that are waiting to access the GPU, two of which account for priorities. The feasibility of schedGPU is validated on three real-world applications. The key observation is that a performance gain is achieved. For single applications, a gain of over 10 times, as measured by GPU utilization and GPU memory utilization, is obtained. For workloads comprising multiple applications, a speed-up of up to 5x in the total execution time is noted. Moreover, the average GPU utilization and average GPU memory utilization is increased by 5 and 12 times, respectively. The work presented in this chapter has been partially published in [94], and it has also been submitted to the IEEE Transactions on Parallel and Distributed Systems journal. The work was carried out in two different internships: one in a major international investment bank, and the other one at Queen's University Belfast (United Kingdom).

Link with Industry

In one of the multiple events aimed at promoting rCUDA [95], we starting a collaboration with a major investment bank in order to consider if the use of rCUDA would be beneficial for their GPU accelerated applications. What turned out in the end to be most valuable to that company was the ad-hoc solution described in this chapter.

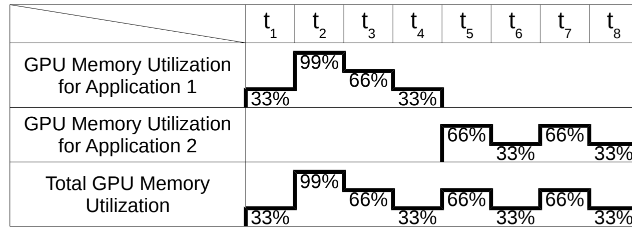
At the beginning of this dissertation, however, we stated that the main objective of this dissertation was to enhance the rCUDA remote GPU virtualization framework for its use in high-performance clusters. Even though it may seem that the work presented in this chapter is totally disconnected from the main objective of this thesis, notice that we also stated that this was an industry-driven thesis, and that all the research was primarily conducted to ultimately transfer the rCUDA technology to industry. In this manner, the solution presented in this chapter also shares the basis of that aim: it is developed in response to industry needs and it was intended to be eventually transferred to industry. In addition, note also that some of the knowledge and experience acquire in previous chapters, as well as the related research, has been used to developed the technology presented in this chapter.

6.1 Introduction

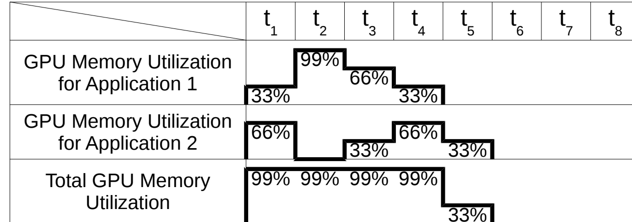
HPC systems are becoming heterogeneous in pursuit of exascale computing [96–98]. These systems not only offer CPUs, but also provide accelerators, such as Graphics Processing Units (GPUs). For example, Tianhe-2 and Titan listed among the top three supercomputers in the June 2016 Top500 list use accelerators[1]. Heterogeneity can be leveraged for improving performance by decomposing compute-intensive components of an application and offloading them on to GPUs. Existing schedulers, such as Slurm [11] and Torque [99], cannot safely schedule multiple applications for sharing the same GPU [100], thereby exclusively locking a GPU for a single application. This results in the under-utilization of GPUs and will have negative implications on the performance of future exascale systems [101]. Hence, the research reported in this chapter aims to improve the utilization of GPUs.

One way of addressing accelerator under-utilization, given that GPUs are usually coupled to a CPU node, is by sharing GPUs among multiple applications that execute on different CPU cores of the same node. However, there is a risk of running out of GPU memory which can cause applications to unexpectedly end. Current techniques that incorporate scheduling [102–104], kernel-based [105–107], synchronization [108] and

[1] https://www.top500.org/list/2016/06/



(a) Using existing workload managers



(b) Using proposed approach

FIGURE 6.1: Example execution of applications on a node with two CPUs and one GPU. In Figure 6.1(a), two applications need to access the GPU, but can only be executed sequentially using existing workload managers. Figure 6.1(b) shows the proposed approach, in which GPU memory is accounted for to maximize utilization allowing for co-scheduling of multiple applications on the same GPU.

architectural [109, 110] approaches cannot safely share GPUs among applications while eliminating the above risk. Therefore, applications are currently executed sequentially, although they may use the GPU for a relatively small fraction of the entire execution time, as shown in Figure 6.1(a). This raises the need for a scheduler that can account for GPU memory required by applications to safely share GPUs as indicated in Figure 6.1(b).

In this chapter, we propose an intra-node, memory safe GPU co-scheduling framework, referred to as *schedGPU*. The framework safely handles multiple application requests to access GPUs by ensuring that memory overruns do not occur during execution. Two implementation approaches are incorporated in the framework, namely a client-server and a shared memory approach. The shared memory approach has fewer overheads than the client-server approach and is therefore suitable. The access of applications to shared memory is synchronized by developing a custom protocol that employs file locks and system signals. This protocol avoids abandonment, the problem that arises when the framework employs other interprocess synchronization mechanisms, such as mutexes. Four policies are proposed and investigated in schedGPU to handle applications that wait to be scheduled on the GPU.

The feasibility of schedGPU is validated on three real-world applications that have

varying GPU utilization. Using schedGPU performance gain in terms of average speed-up, average GPU utilization and average GPU memory utilization when executing concurrent instances of an application using schedGPU is noted to be up to 10 times over conventional execution. For workloads comprising multiple applications, using Slurm in conjunction with schedGPU results in a speed-up of up to 5 times in the total execution time. Moreover, the average GPU utilization and average GPU memory utilization is increased by 5 and 12 times, respectively, when compared to not using schedGPU.

The research contribution of this chapter is an approach for intra-node scheduling in real-time. Conventional workload managers schedule applications ahead-of-time typically over multiple nodes. However, they do not optimize scheduling on each node. The merit of our approach is that scheduling is performed on the fine-grain level in real-time, therefore allowing any application to be executed without knowledge of its GPU requirements prior to execution. The proposed approach safely co-schedules taking GPU memory requirements into account. Existing schedulers exclusively lock a given GPU for an application. Our novel approach safely shares GPUs among multiple applications. For this our approach monitors the GPU resources to service applications. The advantage of our approach is highlighted by the improvement observed in terms of speed-up, GPU utilization and GPU memory utilization when executing applications. For example, a performance gain of up to 10 times is observed.

The remainder of this chapter is organized as follows. Section 6.2 considers the related research. Section 6.3 presents the key concepts of our GPU co-scheduling framework. Section 6.4 provides the implementation approaches considered in our framework. Section 6.5 describes the typical life cycle of an application using our framework. Section 6.6 details a set of four policies incorporated in the framework for scheduling applications. Section 6.7 identifies suitable real-world use-cases for schedGPU. Section 6.8 highlights the benefits of using the proposed framework on three applications through experimental studies. Section 6.9 summarizes this chapter.

6.2 Related Work

Numerous efforts have been made for efficiently scheduling GPU jobs. These include (i) scheduling approaches, (ii) kernel-based approaches, (iii) synchronization approaches, and (iv) architectural approaches.

Scheduling approaches usually include coarse-grain and fine-grain job scheduling. Coarse-grain job scheduling has demonstrated to improve the overall cluster throughput by scheduling concurrent applications on to the nodes of the cluster [102]. While

throughput is improved, the research focused only on inter-node scheduling of jobs, without considering a further level of optimization at the intra-node level. Load balancing is a commonly used approach for fine-grain job scheduling in multiple GPU environments [103, 104]. However, the focus has been to uniformly distribute the executing workload across the GPUs, but not to improve utilization of the GPUs. In this chapter, the focus is on intra-node scheduling at the fine-grain level to maximize GPU utilization thereby improving the overall throughput.

Kernel-based approaches have included event-driven programming models for system-wide scheduling on shared GPUs [105]. This approach does not concurrently share the GPU, but interleaves kernel executions on the GPU, thereby limiting the potential improvement. A mechanism for concurrent execution of GPU kernels from different contexts have been proposed [111]. However, current GPUs account for multiple context kernel concurrency. In addition, the mechanism does not safely handle GPU memory, such that sufficient GPU memory is available for the executing applications. A scheduler to facilitate multiple concurrent kernel executions in OpenCL has been proposed [112]. The scheduler is constrained in that it is limited to two kernels and may lead to potential deadlocks. A more complex framework for synchronizing GPUs in real-time has been investigated [106]. However, this requires modifying the Linux kernel, which constraints its use in production environments. Similarly, an alternate real-time GPU scheduler has been developed by modifying the GPU drivers, only supporting a specific open-source driver [107]. Most kernel-based approaches reported in literature are limiting because extensive modifications are required. The schedGPU framework proposed for intra-node scheduling is not kernel-based, and only requires minimum modification at the application level by making use of a generic API to maximize the utilization of GPUs. It is noted that this is a significantly simpler approach than modifying kernels. Our framework further takes GPU memory into account and safely manages the access of an application to the GPU. The shared-memory approach incorporated in the framework is scalable for use with executing multiple instances of the same application as well as workloads with multiple applications since we avoid the use of a centralized scheduler.

Synchronization approaches aim to manage the implicit and explicit synchronizations in the GPU hardware and software stack for improving application concurrency [108]. This approach avoids concurrent GPU operations to be executed sequentially. This is limited, because an application cannot make use of multiple kernel streams and cannot support unified memory. Our framework achieves synchronization by developing a custom protocol that employs file locks and system signals. The problem of abandonment that arises when using other interprocess synchronization mechanisms, such as mutexes, is therefore mitigated.

Architectural approaches, such as Multi-Process Service (MPS) [109, 113] or Hyper-Q [110, 114], improves the GPU utilization rate by allowing multiple processes, or threads, to simultaneously access a single GPU. However, in this research, GPU memory is not considered, and therefore, jobs fail when GPU memory is not available. The schedGPU framework on the other hand, safely handles GPU memory, and therefore applications do not fail due to insufficient memory but wait in a queue.

We also note that workload managers such as Torque [99], PBS [115] or Slurm [11] include mechanisms for scheduling jobs when computing platforms have GPUs. We differentiate our framework, schedGPU, from such managers in the following two ways. Firstly, workload managers operate at the cluster level (inter-node) and are capable of coarse-grain job scheduling, whereas schedGPU operates at the node level (intra-node) and performs fine-grain job scheduling to share the same physical GPU among multiple CPUs. Secondly, the workload managers work ahead-of-time in that the configurations need to be set before execution of the workload. However, schedGPU works just-in-time, such that scheduling is dynamic and occurs during the execution of the workload. Moreover, it is noted that one of the merits of schedGPU is that jobs can be scheduled at the sub-millisecond timescales due to minimum overheads. Scheduling policies are incorporated in schedGPU for improving GPU and GPU memory utilization, and providing the required quality of service to applications requesting preferential services.

6.3 GPU Scheduling Framework

Consider a typical server, which comprises multiple CPU cores and one GPU. There are two challenges in executing multiple applications on the same GPU. Firstly, consider a scenario in which a conventional workload manager, such as Slurm, is employed to schedule applications from multiple users, then the workload manager will handle multiple requests to the GPU by simply executing the jobs sequentially. While such workload managers can schedule applications on multiple servers they schedule them ahead-of-time, leaving no room for adapting to real-time requirements. Therefore, the jobs are executed sequentially on each server since the workload manager cannot ensure whether the GPU memory requirements of requesting application can be met at any time (for example, whether there is sufficient GPU memory for a second application on the GPU).

Secondly, assume a workload manager can schedule multiple applications on the same GPU. While this can improve GPU utilization, it could lead to potentially terminating the job (for example, if there is insufficient GPU memory an out of memory error will

be returned). This is because there is no safe handling of GPU memory requirements for co-scheduling jobs.

In this chapter, we address the above challenges by presenting a framework for intra-node GPU scheduling, referred to as schedGPU[2], that facilitates the simultaneous execution of multiple applications on a GPU. Using schedGPU multiple applications can request GPU memory during execution time. schedGPU safely co-schedules the applications by taking memory requirements into account and thereby avoids potential memory allocation errors due to unavailable memory on the GPU. The schedGPU framework is proposed and developed for CUDA-based [116] GPU applications. CUDA is widely used in production and commercial environments when compared to the OpenCL alternative [117].

The features of the GPU scheduling framework are:

- *Intra-node scheduling*: most workload managers schedule applications over multiple nodes at the coarse-grain level. However, schedGPU schedules at the fine-grain level to improve GPU utilization of each node.
- *Scheduling in real-time*: unlike conventional workload managers that schedule applications ahead-of-time, our framework can schedule applications on to the GPU in sub-millisecond timescales during execution.
- *Memory-based safe co-scheduling*: typically schedulers allow for executing an application on multiple GPUs. Our approach facilitates the execution of multiple applications on the same GPU concurrently to improve GPU utilization. We consider memory requirements of each application and ensure that no application unexpectedly ends due to insufficient GPU resources.
- *Scalability*: the control of most workload managers that are employed in production is centralized. On the contrary, the control of schedGPU is distributed on each node hence avoiding single points of failure and use on a large number of nodes.

An application that needs to safely use a GPU through our proposed framework follows a four stage life cycle. The first stage is initializing an instance of schedGPU for the application to allow interaction between the application and the framework. The second stage is reserving GPU memory required by an application, we refer to as pre-allocation. The GPU memory requests made by the application are appropriately handled by the framework. The third stage is releasing reserved GPU memory after the application makes use of the GPU, we refer to as post-free. Applications still waiting for the GPU

[2]The schedGPU framework can be requested for download from `http://mural.uv.es/caregon/schedgpu.html`

are potentially serviced. The fourth stage is shutting down the instance of schedGPU that was initialized.

6.4 Implementation Approaches

The schedGPU framework incorporates two approaches to achieve intra-node memory safe co-scheduling, namely a client-server and a shared memory model[3].

A prototype of the client-server approach was briefly reported elsewhere [94]. However, in this chapter, a GPU is shared between multiple applications simultaneously to improve the overall system performance measured by metrics relevant to modern servers.

While the focus in this chapter is the shared memory approach, note that the implementation of the client-server approach is similar to the shared memory approach in a number of ways. For example, the data stored by the shared memory structure is similar to that stored by the server. The access to the data in the server is synchronized using intraprocess mutexes and conditions, which are conceptually similar to interprocess mutexes employed in the shared memory approach. The lifecycle of an application using this approach is similar to the client-server, with the exception that the initialization and the shutdown stages are distributed between the client and the server. Hence, there are client and server initialization and shutdown stages. Additionally, the policies for notifying waiting applications are not approach specific.

6.4.1 Client-Server

In this approach, each CUDA application is a client that requests GPU memory to a centralized server daemon, both of which are executed on the same node. The server permits the application to continue execution if there is sufficient memory on the GPU, otherwise the client may choose to be either blocked until memory is available or informed using CUDA error codes.

Figure 6.2 presents the architecture of the client-server model. In this model, the CUDA application is minimally modified by explicitly calling functions from the client library to pre-allocate the GPU memory required by the application. The calls are forwarded to the server using a UNIX domain socket. We chose UNIX domain sockets over TCP Loopback

[3] In this chapter, 'shared memory' does not refer to aggregating host and device memory for offering a unified address space. This is because schedGPU schedules the access of applications to the GPU from the host side. Instead, we refer to shared memory as the host memory which is accessible for different applications using schedGPU. We refer to 'shared memory data structure' as the format of the data and its contents that is stored in the shared memory.

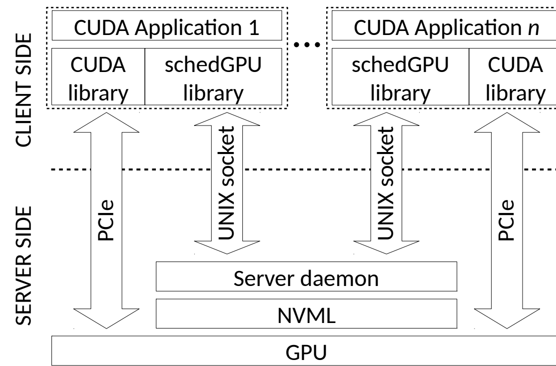


FIGURE 6.2: Architecture of the client-server model

sockets due to the superior performance of the former [118]. The server creates a new thread for each client. A global view of the memory used by all clients is maintained by the server through the NVIDIA Management Library (NVML)[4]. We chose to use NVML instead of the CUDA library to avoid the creation of an additional GPU context that consumes GPU memory. In addition, using NVML the physical devices are accessed instead of using logical devices to avoid any ambiguities in the framework (for example, applications using different identifiers for logical devices referring to the same physical device).

6.4.2 Shared Memory

One disadvantage of running a centralized server is that it may unexpectedly end resulting in the failure of the framework. Therefore, an alternate distributed approach was considered using shared memory in which the clients are responsible for maintaining the global state of the GPU memory in use.

The client library makes decisions based on the information available in the shared memory data structure. Figure 6.3 shows the architecture of the shared memory model. The client library directly makes use of NVML, unlike the library in the client-server model. The shared memory structure is created and managed using the Boost Interprocess library[5].

The shared memory approach not only overcomes the disadvantages of the client-server approach, but also achieves better performance that will be considered in Section 6.8. Therefore, this chapter will focus on presenting the schedGPU framework using the shared memory approach in more detail. Four features are incorporated in the shared memory approach to enhance the robustness of the model. Firstly checkpointing by

[4] `https://developer.nvidia.com/nvidia-management-library-nvml`

[5] `http://www.boost.org/doc/libs/1_59_0_b1/doc/html/interprocess.html`

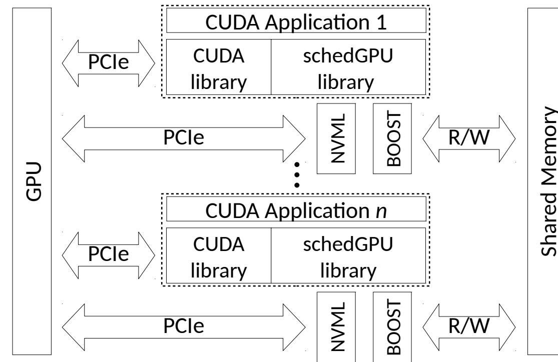


FIGURE 6.3: Architecture of the shared memory model

storing a backup of the shared memory structure by each client library when the application ends.

Secondly, an integrity check and recovery. When a new client application starts, the library checks that the shared memory structure is not corrupt. If corrupt, then it is recovered from a previous backup. If there are no backups or if they are corrupt, then the shared memory structure is freshly initialized.

Thirdly, a sanity check. When any client has ended or is blocked and waiting for free memory, the client library checks that processes with allocated memory are still alive (this is done to free the memory of clients that unexpectedly terminate). If not, the previously allocated memory is freed for the waiting clients.

Fourthly, mitigating abandonment. If a client application unexpectedly ends, the access to the shared memory structure is not blocked and the framework could be used by other clients transparently. If the client had memory allocated, it will be freed.

6.4.2.1 Shared Memory Data

The data structure used for implementing the shared memory approach comprises the following information:

- Total GPU memory: the total installed or physical device memory accessible to the schedGPU framework.
- Total Used memory: the total memory utilized by active schedGPU client applications.
- Itemized Used memory: memory utilized by each client application that is uniquely identified by schedGPU.

- Queue of client applications waiting to access GPU memory: a queue of applications that requested more GPU memory than what was available. The priority of an application is also included depending on the policy in use. Policies will be described in Section 6.6.

6.4.2.2 Synchronizing Access to Shared Memory

It is necessary to synchronize the access of multiple clients to the shared memory data structure to avoid inconsistencies. To this end we explored two methods.

The first method is based on using interprocess mutexes and conditions both provided by the same Boost Interprocess Library that manages the shared memory data structure. Access of each client application to the shared memory data is controlled by a mutex which ensures that only one client modifies the shared memory structure at a given time. A condition is associated with the mutex for either notifying clients that memory has been freed or waiting for notifications on freed memory.

Although this is the most common method, it is restricted due to the problem referred to as *abandonment*: if a process owning the mutex unexpectedly ends, then the mutex becomes unusable and other processes endlessly wait for it. This can be avoided by the use of lock-free methods such as robust mutexes. Such methods are currently available for intraprocess communication (multi-threads). In the case of schedGPU, multiple applications will need to communicate with schedGPU simultaneously and this additionally requires interprocess communication along with intraprocess communication. However, there are no standard and publicly available solutions for interprocess communication[6].

Therefore, in order to surmount the challenge of abandonment when using interprocess communication in schedGPU, we investigated and developed a second method that employs file locks instead of interprocess mutexes for controlling the access of client applications to the shared memory data structure. If a client owning a file lock unexpectedly ends, then the file lock can still be safely used by other clients.

However, conditions cannot be associated with file locks. So a custom protocol using system signals[7] was developed for interprocess communication. The protocol issues a user system signal for notifying a waiting client that memory has been freed. The waiting client on receiving the signal continues execution on the GPU. This method is more appropriate than the first method and is therefore incorporated in the schedGPU framework.

[6] http://www.boost.org/doc/libs/1_61_0/boost/interprocess/detail/robust_emulation.hpp
[7] http://man7.org/linux/man-pages/man7/signal.7.html

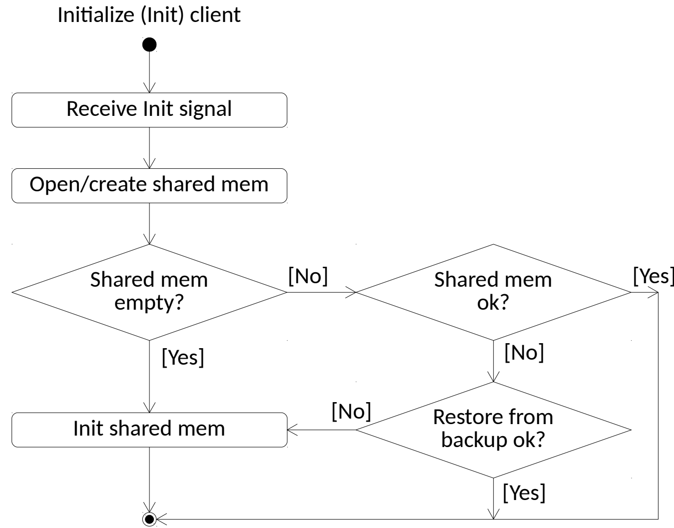


FIGURE 6.4: Initialization of a schedGPU client in the shared memory approach

6.5 The Life Cycle

Three functions are offered by schedGPU[8] that implements the life cycle. They include the initialization, memory pre-allocation and memory post-freeing functions. The shutdown stage in the framework is implicitly called when the CUDA application terminates execution.

Initialization: Figure 6.4 shows how a schedGPU client is initialized in the shared memory approach using the `schedGPUInit()` function. First of all, the client is made ready to handle system signals that are used for internal notifications. The shared memory data structure is then accessed or created. If the shared memory structure is empty, then it is initialized by gathering information of the GPUs using NVML. If the shared memory structure already contains GPU information, then an integrity check is performed to ensure that the data is not corrupted (recovers from the shared memory backup if the integrity check fails).

Pre-allocation: As shown in Figure 6.5 for pre-allocating memory using the `preCudaMalloc()` function, the client requests the ownership of the file lock. When the ownership of the lock is obtained, the client checks that the GPU requested is a valid device and the requested memory is available on the GPU. If memory is insufficient, then the client performs a sanity check on whether other clients with pre-allocated memory are still alive. If memory is freed from other clients, then the client re-checks if there is sufficient memory.

[8]The API reference can be accessed from `http://mural.uv.es/caregon/schedgpu.html`

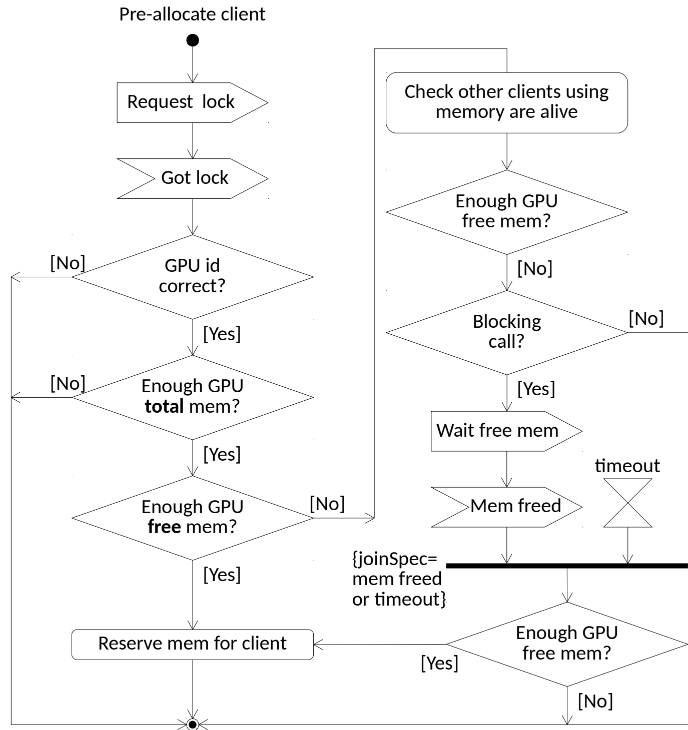


FIGURE 6.5: Pre-allocation of memory by a schedGPU client in the shared memory approach

If the available memory is still insufficient and provided that the pre-allocation call is non-blocking, then control is returned to the application with an error code (`cudaErrorNotReady`). If the call is blocking, then the client waits for a specified time period defined by the application until free memory is available. In the event that the client does not pre-allocate all free GPU memory, then it does not notify other clients of free memory. This notification is carried out during post-freeing.

Post-free: As shown in Figure 6.6 for post-freeing memory using the `postCudaFree()` function, the client requests the ownership of the file lock. When the ownership of the lock is obtained the client checks that (i) the GPU requested is a valid device and (ii) the requested memory for freeing is already pre-allocated on the GPU. If memory is freed, then the clients that may be waiting for memory are notified (refer to Section 6.6).

Shutdown: As shown in Figure 6.7 the client requests the ownership of the file lock. When the ownership is obtained the client (i) ensures that it has post-freed all pre-allocated memory, and (ii) performs a sanity check whether other clients with pre-allocated memory are still alive. If memory is freed, then the waiting clients are notified (refer to Section 6.6). The shutdown is implicitly handled by the `postCudaFree()` function.

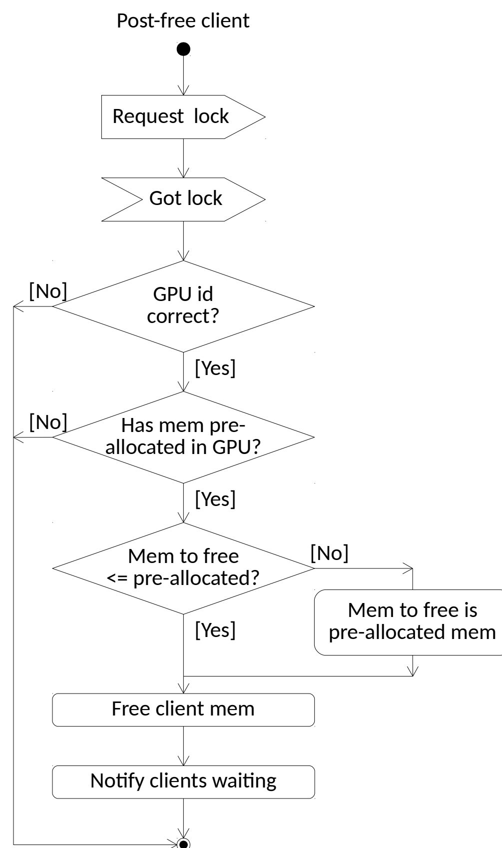


FIGURE 6.6: Post-free of memory by a schedGPU client in the shared memory approach

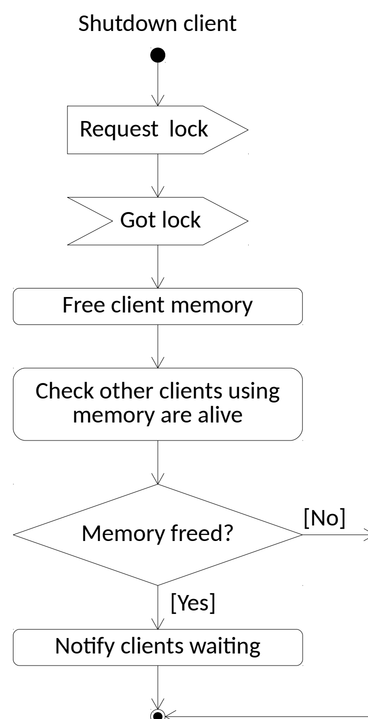


FIGURE 6.7: Shutdown of a schedGPU client in the shared memory approach

6.6 Notification Policies

Client applications that wait in a queue for GPU memory are notified when memory is available because another application released it (post-free stage). Policies are required to schedule memory requests of waiting clients.

A variety of scheduling policies are reported for managing CPU resources [119–123]. Popular policies include First-In, First-Out (FIFO) and those that maximize resource utilization. Given that an operating system will need to prioritize applications, priority-based policies are implemented. We adapted these policies in the context of GPU co-scheduling.

FIFO policy is generally favored due to its simplicity. However, it is limited in that if the first waiting application's GPU memory request cannot be furnished, then even if there was a subsequent waiting client that could be scheduled to access the GPU has to wait. This potentially reduces the utilization rate of the GPU. This can be mitigated by using policies based on consumable resources. In the context of GPU co-scheduling, a policy to maximize the usage of memory on the device is ideal, which in turn increases utilization.

The basic version of both FIFO and maximizing memory utilization policies do not consider the quality of service offered to the clients. This requires the priority of the waiting applications to be accounted for, which provides a preferential service to applications with higher priorities. Taking the above into account, in this chapter, we considered four policies, two basic policies and two priority-based policies.

Consider there are n waiting clients, represented as $C = \{C_1, C_2, \cdots, C_n\}$. We use the following notations:

- C_i is the ith waiting client in the queue
- C_i^p is the ith waiting client with maximum priority p in the queue (used only for priority-based policies)
- Δ_i is maximum time client C_i waits for free memory
- $m_d^{C_i}$ is the memory required by client C_i on device d
- M_d is the free memory available on device d

Policy 1 - First-In, First-Out (FIFO)

In this naive policy, the first waiting client in the queue is the first to be benefited. Client C_1 waits until there is sufficient free memory. If memory is available, then it is preallocated to C_1, and if there is more free memory then the following client is served. If there is insufficient memory, then the client waits for a maximum time of Δ_i since the client is blocked by the scheduler. This is represented as Equation 6.1.

$$preallocate(C_i) = \begin{cases} \big((M_d = M_d - m_d^{C_i}) \wedge (C - \{C_i\}) \wedge \\ \quad (preallocate(C_{i+1}))\big), \text{ if } \big((m_d^{C_i} \leq M_d) \wedge (i < n)\big) \\ \big((M_d = M_d - m_d^{C_i}) \wedge (C - \{C_i\}) \wedge \\ \quad exit()\big), \text{ if } \big((m_d^{C_i} \leq M_d) \wedge (i \geq n)\big) \\ wait(\Delta_i), \text{ otherwise} \end{cases} \tag{6.1}$$

Policy 2 - Maximum Memory Utilization (MMU)

In this policy, the aim is to use maximum GPU memory and therefore the request of the first client in the queue that can be pre-allocated memory is furnished. If no clients in the queue can be serviced, then the clients continue waiting until a subsequent client terminates and more memory is available. This is shown in Equation 6.2.

$$preallocate(C_i) = \begin{cases} M_d = M_d - m_d^{C_i} \wedge C - \{C_i\} \wedge preallocate(C_{i+1}), \\ \quad \text{if } m_d^{C_i} \leq M_d \\ wait(\Delta_i) \wedge preallocate(C_{i+1}) \text{ concurrently}, \text{ if } i < n \\ exit(), \text{ otherwise} \end{cases} \tag{6.2}$$

Policy 3 - Priority FIFO

This policy is similar to the FIFO policy, but has a priority associated with each client. Therefore, in the queue, the clients with the highest priority are pre-allocated memory. The first client with the highest priority will be served, but if there is insufficient memory to serve this request or there is more memory available after serving a request, then a following client with the same priority is served. This policy is represented in

Equation 6.3.

$$preallocate(C_i^p) \quad = \quad \begin{cases} \big((M_d = M_d - m_d^{C_i^p}) \wedge (C - \{C_i^p\}) \wedge \\ \quad (preallocate(C_{i+1}^p))\big), \text{ if } \big((m_d^{C_i^p} \leq M_d) \wedge (i < n)\big) \\ \big((M_d = M_d - m_d^{C_i^p}) \wedge (C - \{C_i^p\}) \wedge \\ \quad exit()\big), \text{ if } \big((m_d^{C_i^p} \leq M_d) \wedge (i \geq n)\big) \\ wait(\Delta_i), \text{ otherwise} \end{cases} \tag{6.3}$$

Policy 4 - Priority MMU

This policy is similar to the MMU policy, but has a priority associated with each client. The aim is again to always use maximum GPU memory and therefore the request of the first client with the highest priority in the queue that can be pre-allocated memory is furnished. If no clients in the queue with the highest priority can be attended to, then the clients continue waiting until a subsequent client terminates and more memory is available. This is shown in Equation 6.4.

$$preallocate(C_i^p) \quad = \quad \begin{cases} M_d = M_d - m_d^{C_i^p} \wedge C - \{C_i^p\} \wedge preallocate(C_{i+1}^p), \\ \quad \text{if } m_d^{C_i^p} \leq M_d \\ wait(\Delta_i) \wedge preallocate(C_{i+1}^p) \text{ concurrently}, \text{ if } i < n \\ exit(), \text{ otherwise} \end{cases} \tag{6.4}$$

The implications of using the above policies for executing workloads with multiple applications is explored in the experiments shown in the subsequent sections.

6.7 Experimental Setup and Use-Cases

The hardware platform and the use-cases employed for validating the feasibility of schedGPU are presented.

6.7.1 Hardware Platform

The experimental test-bed used for our experiments is one 1027GR-TRF Supermicro server comprising two Intel Xeon hexa-core processors E5-2620 v2 (Ivy Bridge) operating at 2.1 GHz and 32 GB of DDR3 SDRAM memory at 1,600 MHz. One NVIDIA Tesla

K20m GPU which has 4,799 MiB of memory is available on the server. The CentOS 6.4 operating system and the CUDA 7.5 with NVIDIA driver 352.39 is used.

6.7.2 Use-cases

Three real-world applications are considered as use-cases in this chapter. The first is a catastrophe risk simulation employed in the financial risk industry, referred to as Aggregate Risk Analysis (ARA) [124]. This simulation computes a key risk metric, namely Probable Maximum Loss (PML) on an industry size input comprising 150,000 catastrophic event trials and a collection of one thousand events and their corresponding losses.

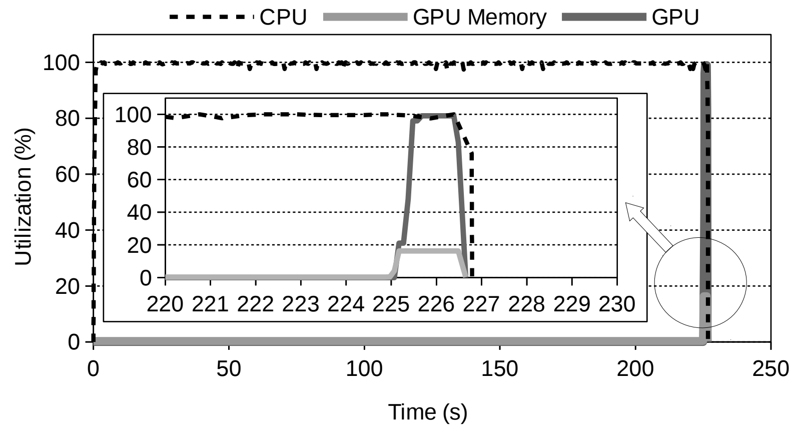
The second and third are applications for aligning DNA sequences in bioinformatics. The second application, which is referred to as MUMmerGPU[9] [125], is used for aligning DNA sequence data to a reference sequence which is useful in genotyping and genomics. In our experiments, the search pattern is a sequence length of 25 base pairs that is matched against the reference, which is a complete genome of Bacillus Anthracis allowing up to five differences in an alignment for 500,000 reads.

The third application is referred to as the GPU Basic Local Alignment Search Tool (GPU-BLAST)[10] [79]. The application searches a database of proteins for a nucleotide with a sequence length of 5,000.

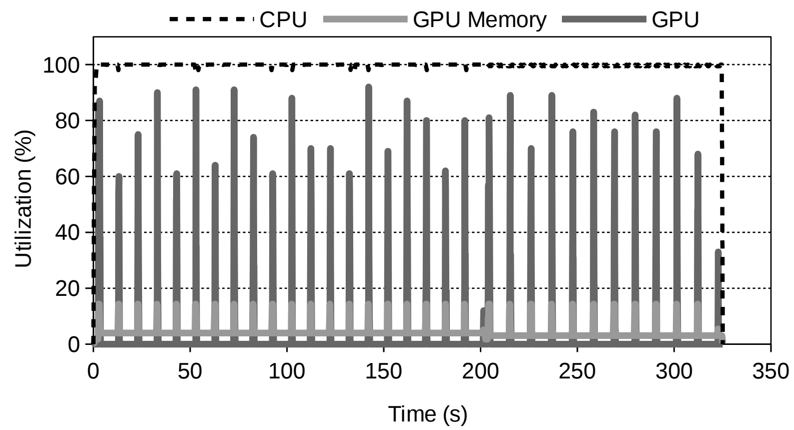
The use-cases were chosen based on the following three observations from Figure 6.8, which shows the CPU and GPU utilization and the GPU memory in use during execution. Firstly, *low GPU utilization.* ARA, in Figure 6.8(a), uses GPU acceleration for a short time period at the end of the simulation. For the given input, over 16% of GPU memory is used and therefore, up to a maximum of 6 concurrent instances of the application can be safely executed on this GPU without potential GPU memory allocation errors (in this chapter, we refer to this as 'maximum concurrent instances'). Such concurrent applications that have low GPU utilization are ideal candidates for schedGPU since the framework can coordinate the access of multiple applications to the GPU, which otherwise would execute sequentially.

Secondly, *moderate GPU utilization.* MUMmerGPU, in Figure 6.8(b), harnesses GPU acceleration at regular intervals. For the given input, the GPU is used for approximately 50% of the total execution time and the maximum GPU memory used is nearly 15% allowing for up to 6 parallel instances of the application to be reliably executed. Concurrent executions of moderate GPU utilizing applications are again ideal candidates

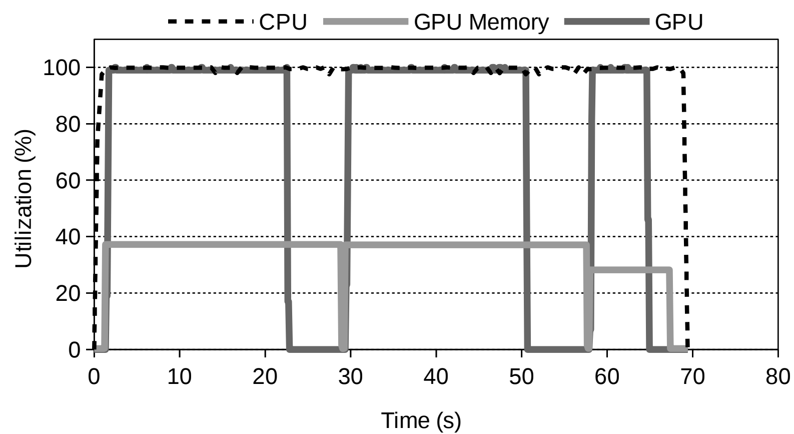
[9] `http://www.mummergpu.sourceforge.net`
[10] `http://archimedes.cheme.cmu.edu/biosoftware.html`



(a) ARA



(b) MUMmerGPU



(c) GPU-BLAST

Figure 6.8: CPU and GPU utilization and GPU memory used for one execution of the applications.

for schedGPU since the framework can maximize the number of these applications safely using the GPU.

Thirdly, *high GPU utilization (and GPU memory is still available).* GPU-BLAST (Figure 6.8(c)) uses the GPU nearly 80% of the total execution time, but for the given input maximum GPU memory used is over 36% of total available memory. This allows for safe execution of up to 2 concurrent application instances. This is not an ideal candidate for schedGPU, however performance gains may be obtained when GPU memory usage drops below 30% towards the end of the execution.

6.8 Evaluation

In this section, we present the experiments carried out for validating the feasibility of schedGPU. For this we (i) evaluate the overheads associated with the client-server and shared memory approaches, (ii) highlight the benefits of employing schedGPU to improve the throughput of concurrent executions of an individual application, and (iii) consider the performance gain of workloads comprising multiple applications.

6.8.1 Overhead of the approaches

Figure 6.9 compares different stages of the schedGPU life cycle. Both the client-server and shared memory approaches are considered. For the former, an additional server initialization and server shutdown stages are required since these are distributed between the client and the server. For the latter, initialization and shutdown are carried out by the client since no servers are present.

It is observed that the server initialization and server shutdown stages for the client-server approach are costly in terms of time although they occur only once. The client initialization and client shutdown stages are however shorter. Regardless, even when excluding the time for initializing and shutting down the server, the total time taken by the client-server is nearly twice as taken by the shared memory approach. This is because communications over TCP sockets introduce an overhead. Therefore, only the shared memory approach is employed in the experiments considered in subsequent sections.

Both shared memory approaches using mutexes and file locks offer similar performance. In the initialization stage, the file locks method requires more time since the notification protocol using system signals needs to be set-up. With mutexes the notification protocol

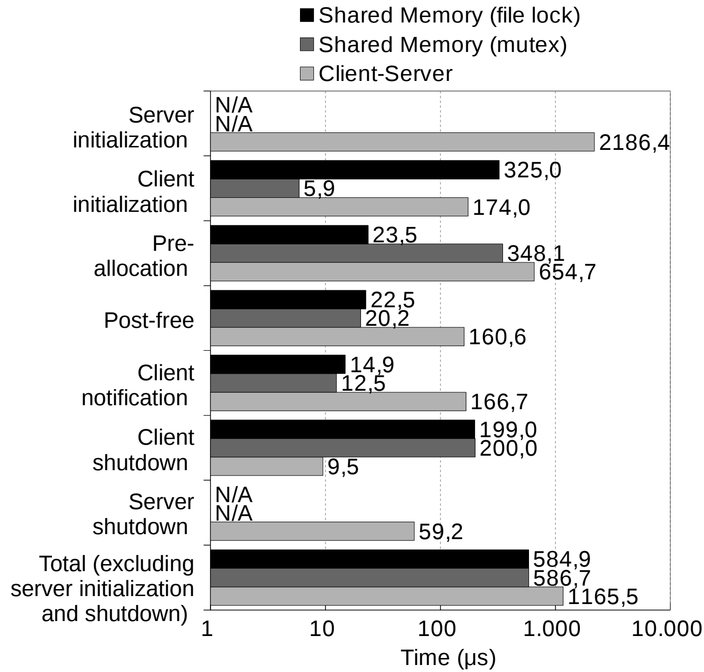


FIGURE 6.9: Comparison of the stages of the schedGPU life cycle for the client-server and shared memory approaches.

using conditions is set-up in the pre-allocation stage and hence an increase in time for the pre-allocation stage is noted.

Given that both the shared memory approaches have similar performance, in the following sections we consider the file lock method since it is more robust than mutexes by avoiding the problem of abandonment.

6.8.2 Performance Gain

We further explore performance in terms of utilization of GPU resources, speed-up and throughput in the following two ways: (i) on an experimental environment, to study mechanisms to achieve maximum performance of the three use-cases with and without schedGPU, and (ii) on a production environment, to assess the potential of schedGPU using real-world workloads.

6.8.2.1 Concurrent Execution of Individual Applications

Three schedGPU functions considered in Section 6.5 were included in the three applications. The initialization function was included at the beginning of the CUDA program, the pre-allocation function was inserted before the CUDA memory allocations and the post-free function was placed after CUDA release memory calls. Up to 12

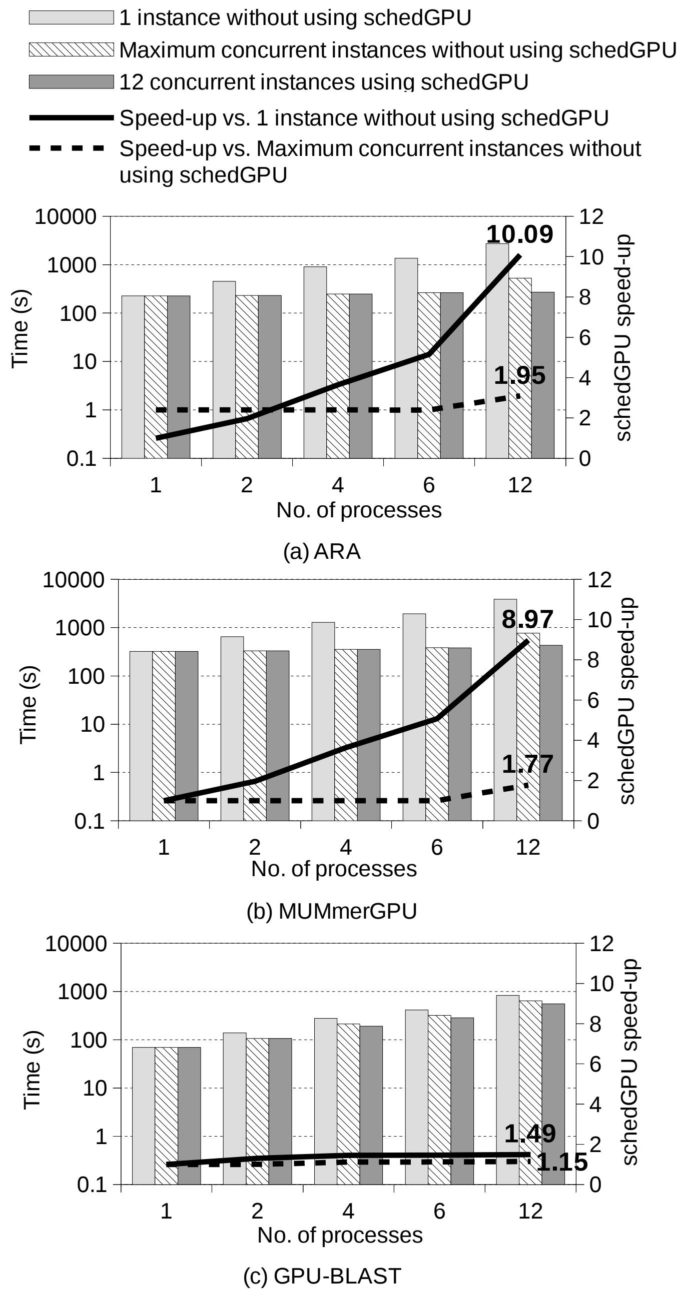


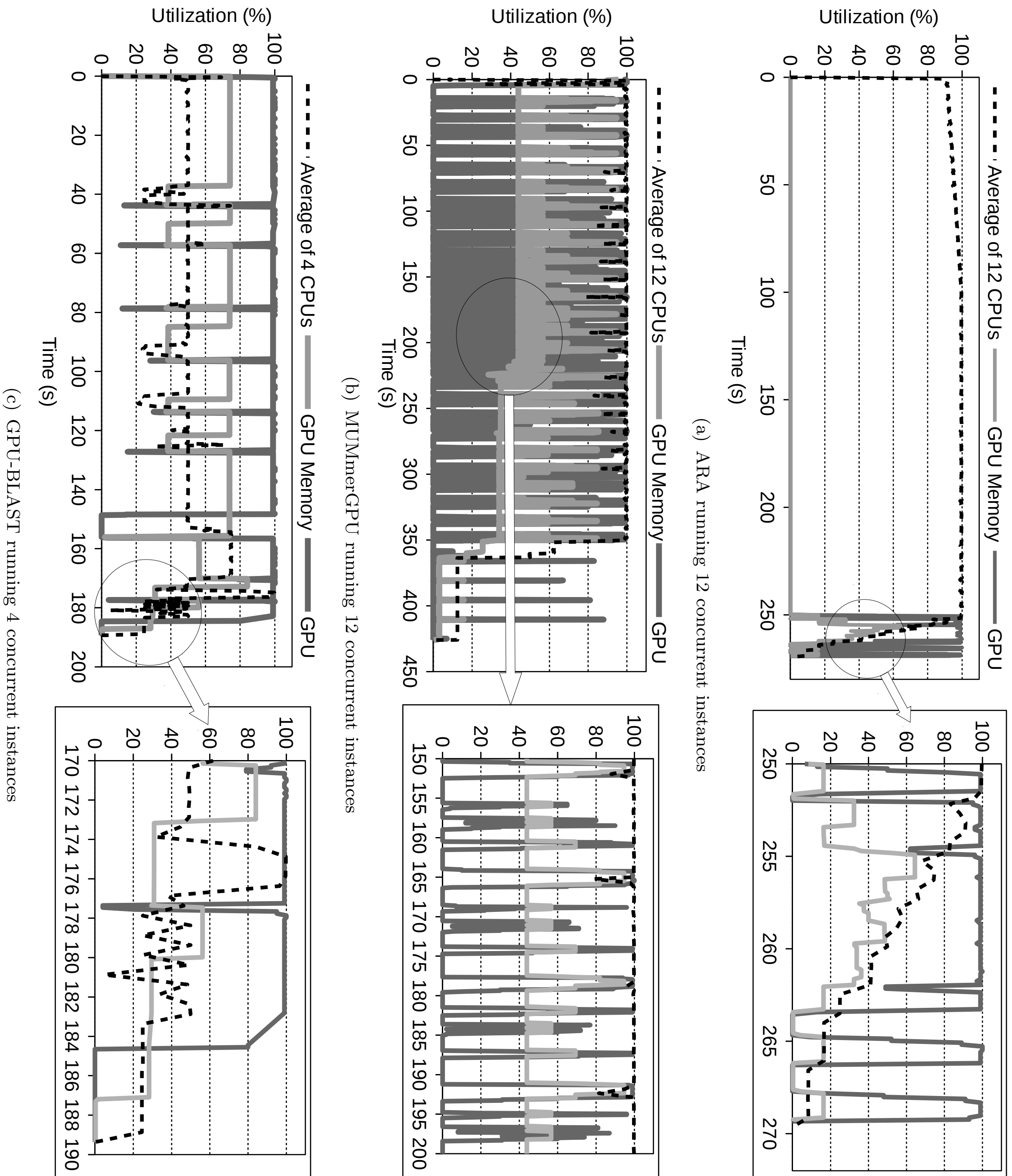
FIGURE 6.10: Execution time and speed-up obtained using schedGPU when varying number of instances of the same application that are concurrently executed.

instances of each application were concurrently executed (the number of CPUs in the experimental test-bed).

Figure 6.10 shows the improvement in execution time and speed-up of the three applications when schedGPU is employed. Three scenarios were considered. The first scenario does not use schedGPU and only one instance of the application can be safely executed at a time. Hence, if 12 instances of an application were required to be executed, then they are executed sequentially. The second scenario, similarly does not use schedGPU. However, we have manually packed tasks to maximize the usage of GPU memory. This is done for the purpose of comparison and is not realistic as the workload managers do not know in advance the GPU memory required by the application. The third scenario employs schedGPU and can safely run multiple instances of the application. It is immediately inferred that when comparing our proposed approach (the third scenario) using schedGPU against (i) the first scenario that does not employ schedGPU there is a 10x speed-up for ARA, nearly 9x speed-up for MUMmerGPU and close to 1.5x speed-up for GPU-BLAST, and (ii) the second scenario the speed-up is approximately doubled when running 12 concurrent instances of ARA and MUMmerGPU. These applications have low and moderate GPU utilization allowing schedGPU to take advantage of the time periods that the GPU remains under-utilized. During these time periods schedGPU services instances that request the GPU to maximize GPU utilization. In the second scenario, the execution of large number of instances of an application (more than 6 for ARA and MUMmerGPU respectively) at the same time will not be possible due to insufficient GPU memory. schedGPU still outperforms this unrealistic scenario of manually packing tasks. Not only is it feasible to execute large number of instances using schedGPU, but also a profitable speed-up is noted.

However, there is only a small improvement in performance for GPU-BLAST with the execution of 4 concurrent instances achieving maximum speed-up. The application has high GPU memory utilization almost during all of its execution time (over 30%, on average, as shown in Figure 6.8(c)). Therefore, there is insufficient GPU memory for boosting the performance of concurrent instances. Nonetheless, even in such cases, schedGPU yields a small improvement in performance by making use of any spare GPU memory.

Figure 6.11 shows the average CPU, GPU and GPU memory utilization when maximum speed-up is obtained for the three applications using schedGPU. The amount of GPU memory utilized by each application is indicated in the figures. GPU utilization is maximized, which in turn results in an observed speed-up.

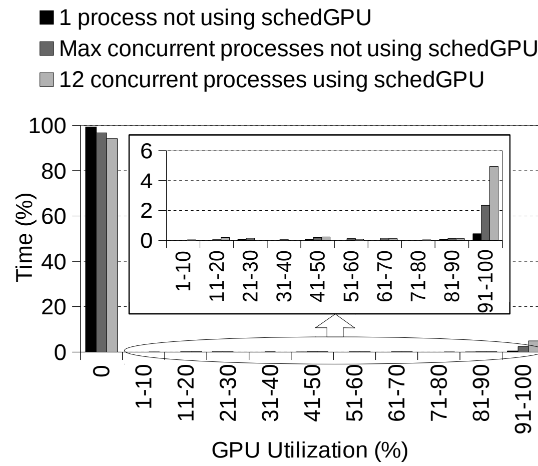


(a) ARA running 12 concurrent instances

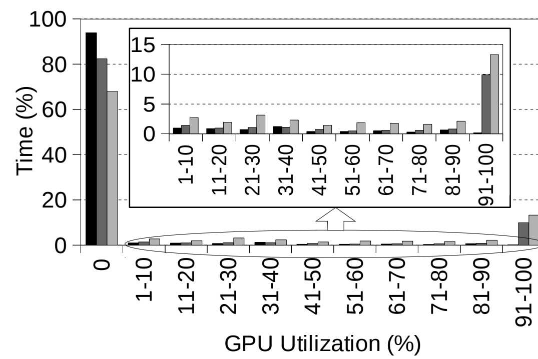
(b) MUMmerGPU running 12 concurrent instances

(c) GPU-BLAST running 4 concurrent instances

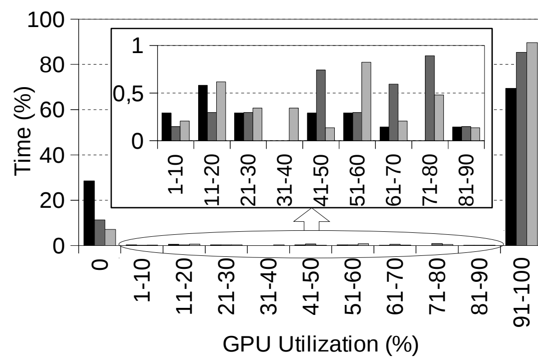
FIGURE 6.11: CPU and GPU usage when running concurrent instances of the applications using schedGPU.



(a) ARA



(b) MUMmerGPU



(c) GPU-BLAST

Figure 6.12: Frequency distribution of GPU utilization when executing the application with and without schedGPU.

TABLE 6.1: Comparison of GPU utilization and GPU memory utilization when executing the use-cases.

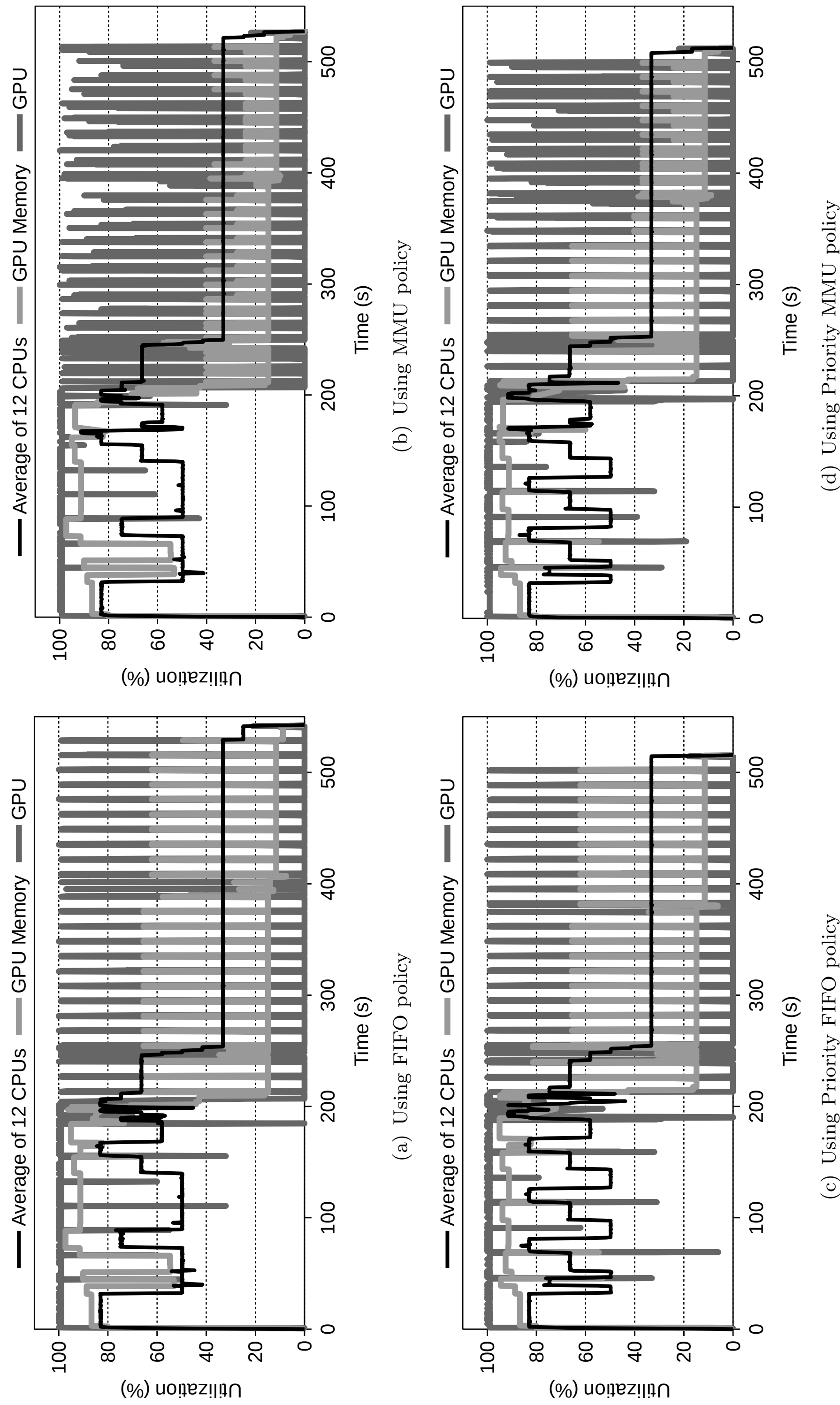
Application	Average GPU Utilization (%), Average GPU Memory Utilization (%)		
	1 instance without using schedGPU	Maximum concurrent instances without using schedGPU	12 concurrent instances using schedGPU
ARA	0.51, 0.37	2.74, 0.90	5.26, 2.02
MUMmerGPU	2.46, 4.04	12.73, 26.13	20.96, 41.59
GPU-BLAST	69.48, 33.73	86.28, 63.71	90.24, 70.09

Figure 6.12 shows the frequency distribution of GPU utilization for the three applications. For all applications it is observed that the amount of time the GPU achieves between 91% and 100% utilization is increased (the time the GPU is not utilized decreases - 0%) and yields a speed up as shown in Figure 6.10. This validates that schedGPU can improve the utilization of resources.

Table 6.1 shows the average GPU utilization and GPU memory utilization for the applications when one instance of the application is executed without using schedGPU, running the maximum number of concurrent instances of the application supported without using schedGPU and 12 concurrent instances using schedGPU are employed. It is immediately evident that schedGPU has superior performance in that the GPU utilization is improved by over 10 times over a single instance and nearly 2 times over six instances for ARA. Similarly, GPU memory utilization is improved by over 10 times for a single instance of MUMmerGPU and nearly 2.25 times over six instances for ARA. The memory utilization of GPU-BLAST is high without using schedGPU leaving little room for optimizing performance. However, a small improvement is noted.

6.8.2.2 Workloads Comprising Multiple Applications

Our experimental test-bed uses the Slurm [11] workload manager for scheduling jobs from multiple users. However, given that one GPU is used in the test-bed, Slurm handles multiple jobs requesting the GPU by sequentially executing them. As expected this results in the under-utilization of the GPU.



(a) Using FIFO policy

(b) Using MMU policy

(c) Using Priority FIFO policy

(d) Using Priority MMU policy

FIGURE 6.13: CPU and GPU usage when running a workload using schedGPU for different client notification policies.

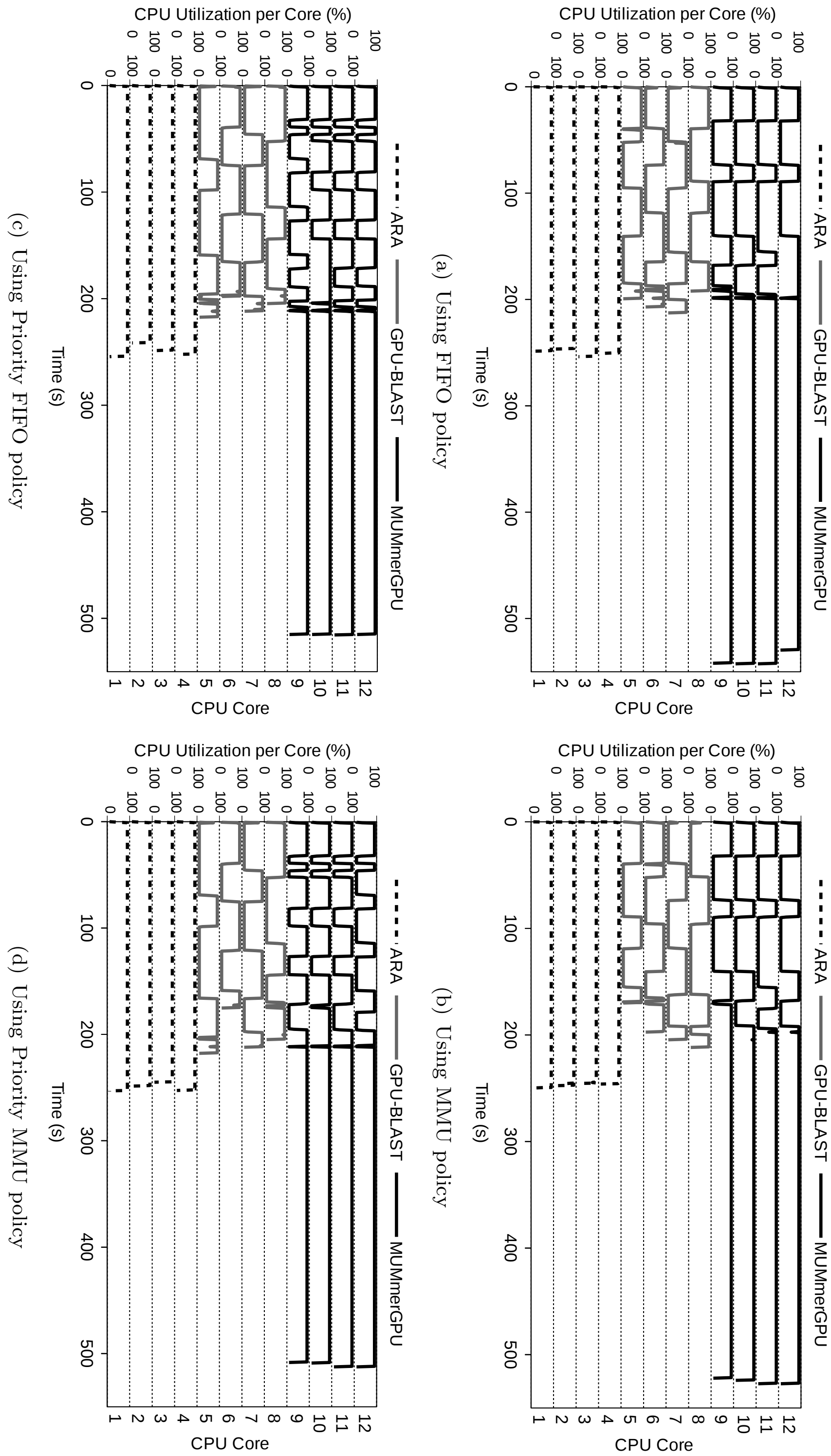


FIGURE 6.14: Usage per CPU core when running a workload using schedGPU for different client notification policies.

Table 6.2: Comparison of GPU utilization and GPU memory utilization when executing a workload comprising multiple applications.

Configuration	Time (s)	Average GPU Utilization (%)	Average GPU Memory Used (%)
Without schedGPU	2,485.20	9.24	3.79
schedGPU FIFO	542.51	43.09	42.65
schedGPU MMU	527.22	43.74	43.25
schedGPU Priority FIFO	515.57	45.59	46.59
schedGPU Priority MMU	512.53	45.75	47.15

On the other hand, schedGPU can be employed to mitigate the above problem by managing the access of multiple job requests requiring GPUs. If there are m real GPUs and n CPUs, then Slurm is reconfigured (by only making changes to the configuration file) to be in possession of $m \times n$ GPUs. On our test-bed Slurm is reconfigured to have 12 GPUs (1 real GPU $\times$ 12 CPUs). This allows for Slurm to execute up to 12 concurrent jobs as if each CPU had access to a GPU. SchedGPU ensures that the jobs make use of the GPU safely.

A workload comprising 12 concurrent jobs (12 jobs since there are 12 CPUs, and each job requires one CPU and one GPU for execution) using 4 instances each of ARA, MUMmerGPU and GPU-BLAST applications was submitted to Slurm. The applications have the same input as presented in the previous section. Figure 6.13 shows the average CPU and GPU utilization and average GPU memory utilization for the workload. Figure 6.14 shows the CPU utilization of the cores for each application in the workload (the Y-axis of the graphs plot utilization from 0-100% for each core).

When considering non-priority based policies, it is observed that using FIFO (refer Figure 6.13(a)) there are peaks in the GPU and GPU memory utilization. This is because when sufficient memory is not available to furnish a request, no further requests are considered and hence memory remains under-utilized. However, using the MMU policy (Figure 6.13(b)) GPU memory utilization is more evenly spread out. Requests of waiting clients are immediately furnished to maximize GPU memory usage. The MMU policy results in a reduction of nearly 3% in the execution time of the workload over the FIFO policy as shown in Table 6.2. An improvement of nearly 1.5% is also noted for both the average GPU utilization and average GPU memory usage for MMU over FIFO.

For non-priority based policies it is noted that CPU cores of the jobs waiting for GPU memory remain idle as shown in Figure 6.14(a) and Figure 6.14(b). This is noted for MUMmerGPU and GPU-BLAST instances since there is insufficient memory on the GPU to furnish all requests.

Since it is observed that the MUMmerGPU takes the most time for completing execution, all MUMmerGPU instances are assigned a high priority in an attempt to optimize the execution of the workload by further reducing the total execution time. When priority-based policies are taken into account, it is observed that the initial CPU utilization increases and similar trends to non-priority based policies is observed for GPU utilization.

In Figure 6.14(c) and Figure 6.14(d), assigning a higher priority to MUMmerGPU instances reduces waiting times for free GPU memory, thereby the CPU is idle for shorter periods of time. This translates into a reduction of total execution time using the priority-based policies by 15 seconds over the best case non-priority policy (MMU) as shown in Table 6.2. Similarly, an improvement of over 4.5% and 9% are noted for GPU utilization and GPU memory utilization, respectively, over the MMU policy.

6.8.3 Evaluation Summary

The experimental evaluation highlights that:

- The overhead of the shared memory approach is significantly less than that of the client-server approach, making it an ideal candidate for facilitating the schedGPU framework (refer Figure 6.9).
- The performance gain, measured in terms of average speed-up, average GPU utilization and average GPU memory utilization, when concurrently executing individual applications using schedGPU is noted to be up to 10 times better than when not using schedGPU.
- For workloads comprising multiple applications, using Slurm in conjunction with schedGPU results in a speed-up of up to 5 times in the total execution time over the execution without schedGPU. The average GPU utilization and average GPU memory utilization is increased by 5 and 12 times, respectively, when compared to not using schedGPU.

6.9 Conclusions

Currently, there are no schedulers that can safely co-schedule multiple GPU applications in terms of memory requirements. This results in the under-utilization of GPUs in high-performance computing systems. In this chapter, we aimed to improve the utilization of GPUs by proposing an intra-node GPU scheduling framework, referred to as *schedGPU*. We incorporated a client-server and shared memory approach for synchronizing the access of multiple applications to the GPU. The schedGPU framework was validated using real-world applications both individually as single applications and collectively as workloads. A gain of over 10 times, as measured by speed-up, GPU utilization and GPU memory utilization, was obtained for individual applications. For workloads, a speed-up of up to 5 times was noted. Moreover, the average GPU utilization and average GPU memory utilization was increased by 5 and 12 times, respectively.

Chapter 7

Conclusions

The work carried out in this PhD thesis has enhanced the rCUDA remote GPU virtualization framework for its use in high-performance computing clusters. In addition, this dissertation has also presented a new framework, referred to as schedGPU, to improve the utilization of GPUs and to facilitate intra-node GPU co-scheduling such that GPUs can be safely shared among multiple applications. In this chapter, the main contributions of these proposals are summarized, followed by an enumeration of the scientific publications related with this dissertation and a discussion about future work.

7.1 Contributions

Chapter 2 has introduced and compared several remote GPU virtualization frameworks that were publicly available at no cost at the beginning of this thesis. Results have shown that the rCUDA virtualization solution was the one offering both better performance and features.

Chapter 3 has presented a new version of rCUDA improving its usability and supporting multithreaded applications and CUDA libraries.

Chapter 4 has detailed the performance analysis and optimizations carried out for InfiniBand networks. First, the influence of FDR InfiniBand on the performance of rCUDA has been analyzed. Second, a new version of rCUDA supporting InfiniBand dual-port adapters has been presented, also evaluating its performance. Next, different InfiniBand Verbs optimizations have been exposed and their impact on rCUDA has been studied. The new version of rCUDA resulting from this study presenting a reduction of up to 35% in execution time of the applications analyzed with respect to the initial version. Finally, it has been shown how the high bandwidth provided by EDR InfiniBand, along with the optimizations exposed in previous chapters, have allowed rCUDA to perform very similar to local GPUs. In this manner, the overhead of the new version of rCUDA is below 5% for the applications studied when using EDR InfiniBand.

Chapter 5 has presented an efficient memory copy mechanism devised for enhancing the rCUDA framework with support for memory copies between remote GPUs located in different nodes of the cluster. The experiments carried out have shown that the enhanced version of rCUDA presents, in general, better bandwidth than CUDA in this kind of copies, obtaining an speed-up of up to 1.42x.

Finally, Chapter 6 has presented a new framework, referred to as schedGPU, which improves the utilization of GPUs. Results have shown a noticeable gain (up to 5x) in both application speed-up and GPU utilization.

7.2 Publications

The following papers, directly related with this dissertation, were published in different international journals and conferences.

Journals:

- C. Reaño and F. Silla. Tuning Remote GPU Virtualization for InfiniBand Networks. *The Journal of Supercomputing* (JoS), volume 72, issue 12, pages 4520-4545, 2016.
- C. Reaño, F. Silla, A. Castelló, A. J. Peña, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurrency and Computation: Practice and Experience* (CCPE), volume 27, issue 14, pages 3746-3770, 2015.

Conferences:

- C. Reaño and F. Silla. Extending rCUDA with Support for P2P Memory Copies between Remote GPUs. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications* (HPCC), pages 789-796, Sydney, Australia, 2016.
- C. Reaño and F. Silla. Reducing the Performance Gap of Remote GPU Virtualization with InfiniBand Connect-IB. In *Proceedings of the 21st IEEE Symposium on Computers and Communications* (ISCC), pages 920-925, Messina, Italy, 2016.
- C. Reaño, F. Silla, G. Shainer, and S. Schultz. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the 16th ACM International Middleware Conference* (Middleware), pages 4:1-4:7, Vancouver, BC, Canada, 2015.
- C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Influence of InfiniBand FDR on the Performance of Remote GPU Virtualization. In *Proceedings of the 2013 IEEE International Conference on Cluster Computing* (Cluster), pages 1-8, Indianapolis, IN, USA, 2013. This publication received the Best Technical Paper Award.
- C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. In *Proceedings of the 19th International Conference on High Performance Computing* (HiPC), pages 1-10, Pune, India, 2012.

Workshops, short papers, posters and demos:

- C. Reaño and F. Silla. Performance Evaluation of the NVIDIA Pascal GPU Architecture: Early Experiences. In *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications* (HPCC), pages 1234-1235, Sydney, Australia, 2016.
- C. Reaño, F. Silla, and M. J. Leslie. schedGPU: Fine-Grain Dynamic and Adaptative Scheduling for GPUs. In *Proceedings of the 14th International Conference on High Performance Computing & Simulation* (HPCS), pages 993-997, Innsbruck, Austria, 2016.
- C. Reaño and F. Silla. A Live Demo on Remote GPU Accelerated Deep Learning Using the rCUDA Middleware. In *Proceedings of the 16th ACM International Middleware Conference* (Middleware), pages 3:1-3:2, Vancouver, BC, Canada, 2015.
- C. Reaño and F. Silla. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing* (Cluster), pages 488-489, Chicago, IL, USA, 2015.
- C. Reaño and F. Silla. InfiniBand Verbs Optimizations for Remote GPU Virtualization. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing* (Cluster Workshops), pages 825-832, Chicago, IL, USA, 2015.
- C. Reaño, F. Pérez, and F. Silla. On the Design of a Demo for Exhibiting rCUDA. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGrid), pages 1169-1172, Shenzhen, China, 2015.
- C. Reaño, F. Silla, A. J. Peña, G. Shainer, S. Schultz, A. Castelló, E. S. Quintana–Ortí, and J. Duato. Boosting the performance of remote GPU virtualization using InfiniBand connect-IB and PCIe 3.0. In *Proceedings of the 2014 IEEE International Conference on Cluster Computing* (Cluster), pages 266-267, Madrid, Spain, 2014.
- C. Reaño, F. Silla, and J. Duato. Maximizing the performance of HPC clusters with rCUDA. In *Proceedings of the 10th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES), pages 125-128, Fiuggi, Italy, 2014.

Summer schools:

- C. Reaño, F. Silla, and J. Duato. Maximizing the performance of HPC clusters with rCUDA. In *Proceedings of the 10th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES), pages 125-128, Fiuggi, Italy, 2014.
- C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Remote GPU Virtualization: a Performance Evaluation of Freely Available CUDA Frameworks. In *Programming and Tuning Massively Parallel Systems Summer School* (PUMPS), Barcelona, Spain, 2013.

Journals currently under review:

- C. Reaño, F. Silla, D. S. Nikolopoulos, and B. Varghese. Intra-node Memory Safe GPU Co-Scheduling. Submitted to *IEEE Transactions on Parallel and Distributed Systems journal* (TPDS).
- C. Reaño and F. Silla. Implementing P2P Memory Copies within rCUDA to Support GPUs Located in Different Nodes. Submitted to *Journal of Parallel and Distributed Computing* (JPDC).

In addition, other related papers have been published in domestic conferences:

- C. Reaño and F. Silla. Extendiendo rCUDA con soporte para copias P2P entre GPUs remotas. In *Actas de las XXVII Jornadas de Paralelismo* (JJPP), pages 203-208, Salamanca, Spain, 2016.
- C. Reaño and F. Silla. Acelerando aplicaciones con CUDA: GPUs de sobremesa frente a GPUs para servidores. In *Actas de las XXVI Jornadas de Paralelismo* (JJPP), pages 198-204, Córdoba, Spain, 2015.
- C. Reaño, F. Silla, and J. Duato. Evaluación de Prestaciones de Copias 2D y 3D con rCUDA. In *Actas de las XXV Jornadas de Paralelismo* (JJPP), pages 81-86, Valladolid, Spain, 2014.
- C. Reaño, A. Castelló, S. Iserte, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Virtualización Remota de GPUs: Evaluación de Soluciones Disponibles para CUDA. In *Actas de las XXIV Jornadas de Paralelismo* (JJPP), pages 271-276, Madrid, Spain, 2013.

- C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. CU2rCU: a CUDA-to-rCUDA Converter. In *Actas de las XXIII Jornadas de Paralelismo* (JJPP), pages 44-49, Elche, Spain, 2012.

- C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. rCUDA: Uso Concurrente de Dispositivos Compatibles con CUDA de Forma Remota. Adaptación a CUDA 4. In *Actas de las XXII Jornadas de Paralelismo* (JJPP), pages 311-316, La Laguna, Spain, 2011.

The dissertation has also led to participate in several tutorials presented in international conferences:

- F. Silla and C. Reaño. Reducing Power Consumption of Data Centers with rCUDA. *2016 International Conference on High Performance and Embedded Architecture and Compilation* (HiPEAC), Prague, Czech Republic, 2016.

- F. Silla and C. Reaño. Improving overall performance and energy consumption of your cluster with remote GPU virtualization. *16th ACM International Middleware Conference* (Middleware), Vancouver, BC, Canada, 2015.

- F. Silla and C. Reaño. On the use of remote GPU virtualization for managing the GPUs of your cluster in a flexible way. *2014 IEEE International Conference on Cluster Computing* (Cluster), Madrid, Spain, 2014.

- C. Reaño. rCUDA hands on session - learn how it works. *2014 HPC Advisory Council Switzerland Conference* (HPCAC), Lugano, Switzerland, 2014.

During this thesis, the rCUDA framework has also been exhibited at the Mellanox Technologies[1] booth in the following international industrial/research exhibitions:

- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Salt Lake City (USA), 2016.

- International Supercomputing Conference (ISC), Frankfurt (Germany), 2016.

- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Austin (USA), 2015.

- International Supercomputing Conference (ISC), Frankfurt (Germany), 2015.

- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), New Orleans (USA), 2014.

[1] `https://www.mellanox.com/`

- International Supercomputing Conference (ISC), Leipzig (Germany), 2014.
- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Denver (USA), 2013.
- International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing, SC), Salt Lake City (USA), 2012.
- International Supercomputing Conference (ISC), Hamburg (Germany), 2012.

Finally, the following works indirectly related to this dissertation have been accepted for publication:

Book Chapters:

- J. Prades, F. Campos, C. Reaño, and F. Silla. GPU as a Service: providing GPU-acceleration to federated cloud systems. *Developing Interoperable and Federated Cloud Architecture*, chapter 10, pages 281-313. 2016.

Journals:

- F. Silla, S. Iserte, C. Reaño, and J. Prades. On the Benefits of the Remote GPU Virtualization Mechanism: the rCUDA Case. Accepted for publication in the journal *Concurrency and Computation: Practice and Experience* (CCPE).
- J. Prades, B. Varghese, C. Reaño, and F. Silla. Multi-tenant virtual GPUs for optimising performance of a financial risk application. Accepted for publication in the *Journal of Parallel and Distributed Computing* (JPDC).
- A. J. Peña, C. Reaño, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing* (PARCO), volume 40, issue 10, pages 574-588, 2014.

Conferences:

- F. Pérez, C. Reaño, and F. Silla. Providing CUDA Acceleration to KVM Virtual Machines in InfiniBand Clusters with rCUDA. In *Proceedings of the 16th IFIP International Conference on Distributed Applications and Interoperable Systems* (DAIS), pages 82-95, Heraklion, Greece, 2016.

- B. Varghese, J. Prades, C. Reaño, and F. Silla. Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application. In *Proceedings of the 11th IEEE International Conference on e-Science* (e-Science), pages 47-56, Munich, Germany, 2015.
- S. Iserte, A. Castelló, R. Mayo, E. S. Quintana–Ortí, F. Silla, J. Duato, C. Reaño, and J. Prades. SLURM Support for Remote GPU Virtualization: Implementation and Performance Study. In *Proceedings of the 26th IEEE International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD), pages 318-325, Paris, France, 2014.

Workshops, short papers, posters and demos:

- S. Iserte, J. Prades, C. Reaño, and F. Silla. Increasing the Performance of Data Centers by Combining Remote GPU Virtualization with Slurm. In *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGrid), pages 98-101, Cartagena, Colombia, 2016.
- F. Silla, C. Reaño, J. Prades, S. Iserte. Benefits of remote GPU virtualization: the rCUDA perspective. In *GPU Technology Conference* (GTC), San Jose, CA, USA, 2016.
- F. Silla, J. Prades, S. Iserte, and C. Reaño. Remote GPU Virtualization: Is It Useful?. In *Proceedings of the 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era* (HiPINEB), pages 41-48, Barcelona, Spain, 2016.
- J. Prades, C. Reaño and F. Silla. CUDA acceleration for Xen virtual machines in InfiniBand clusters with rCUDA. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP), pages 35:1-35:2, Barcelona, Spain, 2016.

Domestic conferences:

- J. Prades, B. Varghese, C. Reaño, and F. Silla. Reduciendo el Tiempo de Ejecución de una Aplicación de Cálculo de Riesgos Financieros a través del uso de GPUs Virtuales. In *Actas de las XXVII Jornadas de Paralelismo* (JJPP), pages 245-253, Salamanca, Spain, 2016.
- S. Iserte, A. Castelló, R. Mayo, E. S. Quintana–Ortí, C. Reaño, J. Prades, F. Silla, and J. Duato. Comparativa de políticas de selección de GPUs remotas en

clusters HPC. In *Actas de las XXVI Jornadas de Paralelismo* (JJPP), pages 67-72, Córdoba, Spain, 2015.

- F. Campos, J. Prades, C. Reaño, and F. Silla. Uso de aceleradores CUDA en entornos cloud mediante rCUDA y KVM. In *Actas de las XXVI Jornadas de Paralelismo* (JJPP), pages 393-402, Córdoba, Spain, 2015.

- R. Alegre, C. Reaño, J. Prades, and F. Silla. Uso de Rodinia y Parboil para evaluar las prestaciones de la virtualización remota de GPUs. In *Actas de las XXVI Jornadas de Paralelismo* (JJPP), pages 425-432, Córdoba, Spain, 2015.

- J. Prades, C. Reaño, F. Silla, and J. Duato. Virtualización de GPUs en entornos XEN usando rCUDA. In *Actas de las XXVI Jornadas de Paralelismo* (JJPP), pages 102-110, Córdoba, Spain, 2015.

- J. Prades, C. Reaño, F. Silla, and J. Duato. Virtualización Remota de GPUs: Reduciendo el Coste del Hardware Ocioso. In *Actas de las XXV Jornadas de Paralelismo* (JJPP), pages 145-153, Valladolid, Spain, 2014.

- F. Pérez, J. Prades, C. Reaño, F. Silla, and J. Duato. Uso de rCUDA en Máquinas Virtuales KVM: Análisis y Prestaciones. In *Actas de las XXV Jornadas de Paralelismo* (JJPP), pages 327-334, Valladolid, Spain, 2014.

- S. Iserte, A. Castelló, A. J. Peña, C. Reaño, J. Prades, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Extendiendo SLURM con Soporte para el Uso de GPUs Remotas. In *Actas de las XXV Jornadas de Paralelismo* (JJPP), pages 135-143, Valladolid, Spain, 2014.

- S. Iserte, A. Castelló, C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana–Ortí, and J. Duato. Un planificador de GPUs remotas para clusters HPC. In *Actas de las XXIV Jornadas de Paralelismo* (JJPP), pages 193-198, Madrid, Spain, 2013.

7.3 Future Directions

As for future work, this dissertation has left some matters which require further investigation, namely:

- In Section 4.4.3 we have shown that rCUDA internal paths seems to be limiting the potential bandwidth gain of using multiple QPs.

- In Section 5.4 we have commented that the initialization overhead of the mechanism devised for copying data between GPUs located in different server

nodes, could be avoided if it were carried out during application start up. However, as we have also pointed out, in this case some memory could be wasted.

It is also planned to implement a new communications module, based on the InfiniBand one, for supporting RoCE (RDMA over Converged Ethernet) [126] in the rCUDA framework. In addition, different optimizations are also planned to be explored at the communications stage.

Apart from improving rCUDA at the communications level, the following research lines are currently in progress:

- Virtual Machines: provide GPU acceleration to several virtual machines environments concurrently.
- GPU job migration: migrate applications that make use of GPUs.
- Deep Learning: support for deep learning frameworks accelerated by GPUs, such as Caffe [127], TensorFlow [128] or Theano [129].
- Low power processors: analyze and study the performance in environments using not only x86 processors, but also low power ones, such as ARM processors.
- Transatlantic remote GPUs: collaboration with the Agency for Science, Technology and Research (A*STAR) [130] for using rCUDA over InfiniCortex [131], a transatlantic InfiniBand link, to provide access to remote GPUs located in different continents.
- Windows: rCUDA currently targets the Linux operating system. Extend support to Windows platforms.
- Rendering: support for GPU rendering frameworks, such as Blender [132] or OctaneRender [133].
- Numerical computing: support for GPU accelerated numerical computing environments such as Matlab [134].

References

[1] Mellanox. InfiniBand Adapter Card, 2012. URL http://www.mellanox.com/related-docs/products/IB_Adapter_card_brochure_c_2_3.pdf.

[2] Cray. Cray XC Series Network, 2012. URL http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[3] F. Petrini, E. Frachtenberg, A. Hoisie, and S. Coll. Performance evaluation of the quadrics interconnection network. In *2013 IEEE International Conference on Cluster Computing*, pages 125–142, 2003.

[4] NVIDIA. *CUDA C Programming Guide 7.0*, 2015.

[5] 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, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

[6] 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.

[7] 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.

[8] 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.

[9] Vladimir Surkov. Parallel option pricing with fourier space time-stepping method on graphics processing units. *Parallel Computing*, 36(7):372 – 380, 2010. Parallel and Distributed Computing in Finance.

[10] 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.

[11] Andy B. Yoo, Morris A. Jette, and Mark Grondona. *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper*, chapter SLURM: Simple Linux Utility for Resource Management, pages 44–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[12] Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reaño. Remote GPU virtualization: Is it useful? In *2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era HiPINEB@HPCA 2016, Barcelona, Spain, March 12, 2016*, pages 41–48, 2016.

[13] Federico Silla, Carlos Reaño, Javier Prades, and Sergio Iserte. Benefits of remote GPU virtualization: the rCUDA perspective. In *GPU Technology Conference*, April 2016.

[14] 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.

[15] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Computing*, 40(10):574 – 588, 2014.

[16] Antonio J. Peña. *Virtualization of Accelerators in High Performance Clusters.* PhD thesis, University Jaume I, Castellón, Spain, 2013.

[17] Bitfusion.io. Software To Manage Deep Learning & GPUs, 2017. URL `http://www.bitfusion.io/`.

[18] Carlos Reaño and Federico Silla. A performance comparison of CUDA remote GPU virtualization frameworks. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 488–489, 2015.

[19] ZILLIANS. V-GPU: GPU virtualization, 2014. URL `https://github.com/zillians/platform_manifest_vgpu`.

[20] 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, March 2011.

[21] 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 *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1207–1214, Nov 2012.

[22] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6271 of *Lecture Notes in Computer Science*, pages 379–391. Springer Berlin Heidelberg, 2010.

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

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

[25] Georgia Tech. Shadowfax II - scalable implementation of GPGPU assemblies, 2014. URL `http://keeneland.gatech.edu/software/keeneland/kidron`.

[26] NVIDIA. *CUDA API Reference Manual 4.2*, April 2012.

[27] José Duato, Francisco D. Igual, Rafael Mayo, Antonio J. Peña, Enrique S. Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters. In Hai-Xiang Lin, Michael Alexander, Martti Forsell, Andreas Knüpfer, Radu Prodan, Leonel Sousa, and Achim Streit, editors, *Euro-Par Workshops*, volume 6043 of *Lecture Notes in Computer Science*, pages 385–394. Springer, 2009.

[28] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. Performance of CUDA virtualized remote GPUs in high performance clusters. In Guang R. Gao and Yu-Chee Tseng, editors, *ICPP*, pages 365–374. IEEE, 2011.

[29] José Duato, Antonio J. Peña, Federico Silla, Juan Carlos Fernández, Rafael Mayo, and Enrique S. Quintana-Ortí. Enabling cuda acceleration within virtual machines using rCUDA. In *HiPC*, pages 1–10. IEEE, 2011.

[30] Citrix Systems. Xen, 2013. URL http://xen.org/.

[31] NVIDIA. *The NVIDIA GPU Computing SDK Version 4*, 2011.

[32] Carlos Reaño, Antonio J. Peña, Federico Silla, José Duato, Rafael Mayo, and Enrique S. Quintana-Ortí. CU2rCU: Towards the complete rcuda remote GPU virtualization and sharing solution. In *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22, 2012*, pages 1–10, 2012.

[33] Carlos Reaño, Federico Silla, Adrián Castelló, Antonio J. Peña, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurrency and Computation: Practice and Experience*, 27(14):3746–3770, 2015.

[34] NVIDIA. *The NVIDIA CUDA Compiler Driver NVCC Version 5*, 2012.

[35] D. Quinlan, T. Panas, and C. Liao. ROSE, 2013. URL http://rosecompiler.org/.

[36] Free Software Foundation. GCC, the GNU Compiler Collection, 2013. URL http://gcc.gnu.org/.

[37] LLVM. Clang: a C language family frontend for LLVM, 2013. URL http://clang.llvm.org/.

[38] G. Martinez, M. Gardner, and Wu chun Feng. CU2CL: A cuda-to-opencl translator for multi- and many-core architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300–307, Dec 2011.

[39] LLVM. The LLVM Compiler Infrastructure, 2013. URL http://llvm.org/.

[40] Sandia National Labs. LAMMPS Molecular Dynamics Simulator, 2004. URL http://lammps.sandia.gov/.

[41] CUPTI. *CUDA Profiler User's Guide Version 5*, 2012.

[42] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. Van De Geijn, and Field G. Van Zee. The flame approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput.*, 72(9):1134–1143, September 2012.

[43] InfiniBand Trade Association (IBTA), 2015. URL http://www.infinibandta.org.

[44] Carlos Reaño, Rafael Mayo, Enrique S. Quintana-Ortí, Federico Silla, José Duato, and Antonio J. Peña. Influence of infiniband FDR on the performance of remote GPU virtualization. In *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, pages 1–8, 2013.

[45] 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 2015, Vancouver, BC, Canada, December 7-11, 2015*, pages 4:1–4:7, 2015.

[46] Carlos Reaño and Federico Silla. Reducing the performance gap of remote GPU virtualization with InfiniBand connect-ib. In *IEEE Symposium on Computers and Communication, ISCC 2016, Messina, Italy, June 27-30, 2016*, pages 920–925, 2016.

[47] Carlos Reaño and Federico Silla. Tuning remote GPU virtualization for infiniband networks. *The Journal of Supercomputing*, 72(12):4520–4545, 2016.

[48] NVIDIA. *NVIDIA CUDA Libraries 5.0*, 2012.

[49] NVIDIA. Popular GPU-Accelerated Applications, 2012. URL `http://www.nvidia.com/docs/IO/123576/nv-applications-catalog-lowres.pdf`.

[50] 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):117, 2013.

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

[52] Mellanox. Connect-IB Single and Dual QSFP+ Port PCI Express Gen3 x16 Adapter Card User Manual, 2014.

[53] NVIDIA. *CUDA Samples Reference Manual 7.0*, 2015.

[54] Mellanox. ConnectX-3 VPI Single and Dual QSFP+ Port Adapter Card User Manual, 2014.

[55] John D'Ambrosia. Ethernet in the TOP500, 2014. URL `http://www.scientificcomputing.com/blogs/2014/07/ethernet-top500`.

[56] TOP500 Supercomputer Sites, 2014. URL `http://www.top500.org/`.

[57] InfiniBand Trade Association (IBTA). *The InfiniBand Trade Association Specification*, 2007.

[58] Gregory Kerr. Dissecting a small infiniband application using the verbs API. *CoRR*, abs/1105.1827, 2011.

[59] Bob Woodruff, Sean Hefty, Roland Dreier, and Hal Rosenstock. Introduction to the infiniband core software. In *Linux Symposium*, volume 2, 2005.

[60] Tarick Bedeir. Building an RDMA-capable application with IB Verbs. Technical report, HPC Advisory Council, 2010. URL `http://www.hpcadvisorycouncil.com/pdf/building-an-rdma-capable-application-with-ib-verbs.pdf`.

[61] Qian Liu and Robert D. Russell. A performance study of InfiniBand fourteen data rate (FDR). In *Proceedings of the High Performance Computing Symposium*, HPC '14, pages 16:1–16:10, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[62] Nathan Hjelm. Optimizing one-sided operations in open MPI. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 123:123–123:124, New York, NY, USA, 2014. ACM.

[63] Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and DhabaleswarK. Panda. Designing MPI library with dynamic connected transport (DCT) of InfiniBand: Early experiences. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 278–295. Springer International Publishing, 2014.

[64] NVIDIA. *NVIDIA CUDA Samples 6.5*, 2014.

[65] University of Michigan. HOOMD-blue web page., 2014. URL `http://codeblue.umich.edu/hoomd-blue`.

[66] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008.

[67] University of Tennessee. MAGMA: Matrix Algebra on GPU and Multicore Architectures, 2014. URL `http://icl.cs.utk.edu/magma`.

[68] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[69] GROMACS web page, 2014. URL http://www.gromacs.org/.

[70] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[71] 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.

[72] Yongchao Liu. CUDA-MEME: a motif discovery software based on MEME, 2014. URL https://sites.google.com/site/yongchaosoftware/mcuda-meme.

[73] J.J. Rodriguez-Vazquez, J.L. Vazquez-Poletti, C. Delgado, A. Bulgarelli, and M. Cardenas-Montes. Performance evaluation of a signal extraction algorithm for the Cherenkov Telescope Array's Real Time Analysis pipeline. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 292–293, Sept 2014.

[74] NVIDIA. *CUDA Samples Reference Manual 7.0*, 2015.

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

[76] NVIDIA. GPU Applications, 2015. URL http://www.nvidia.com/object/gpu-applications.html.

[77] 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):117, 2013.

[78] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris. GPU Acceleration for Support Vector Machines. In *12th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2011)*, Apr 2011.

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

[80] Innovative Computing Laboratory (University of Tennessee). Magma: Matrix algebra on gpu and multicore architectures, 2014. URL http://icl.cs.utk.edu/magma.

[81] 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.

[82] Li H and Durbin R. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25:1754–1760, 2009.

[83] Carlos Reaño and Federico Silla. Extending rCUDA with support for P2P memory copies between remote GPUs. In *IEEE International Conference on High Performance Computing and Communications, HPCC 2016, Sydney, Australia, December 12-14, 2016*, pages 789–796, 2016.

[84] Sergio Iserte, Javier Prades, Carlos Reaño, and Federico 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)*, pages 98–101, May 2016.

[85] NVIDIA. Developing a linux kernel module using GPUDirect RDMA, 2015. URL `http://docs.nvidia.com/cuda/gpudirect-rdma/index.html#ixzz42PAc6GGX`.

[86] Mellanox. OFED GPUDirect RDMA Product Brief, 2014. URL `http://www.mellanox.com/related-docs/prod_software/PB_GPUDirect_RDMA.PDF`.

[87] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. Exploiting GPUDirect RDMA in designing high performance OpenSHMEM for NVIDIA GPU clusters. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 78–87, 2015.

[88] Andrew J. Younge, John Paul Walters, Stephen P. Crago, and Geoffrey C. Fox. Supporting high performance molecular dynamics in virtualized clusters using IOMMU, SR-IOV, and GPUDirect. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 31–38, New York, NY, USA, 2015. ACM.

[89] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient inter-node MPI communication using GPUDirect RDMA for infiniband clusters with NVIDIA gpus. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 80–89, 2013.

[90] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE. URL `http://mvapich.cse.ohio-state.edu/`.

[91] Davide Rossetti. Benchmarking GPUDirect RDMA on Modern Server Platforms, 2014. URL `http://devblogs.nvidia.com/parallelforall/benchmarking-gpudirect-rdma-on-modern-server-platforms/`.

[92] Adam Polak. Counting triangles in large graphs on GPU. *CoRR*, abs/1503.00576, 2015.

[93] NVIDIA. *CUDA API Reference Manual 7.5*, 2015. URL `http://docs.nvidia.com/cuda/cuda-runtime-api/index.html`.

[94] Carlos Reaño, Federico Silla, and Matthew J. Leslie. schedGPU: Fine-grain Dynamic and Adaptive scheduling for GPUs. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 993–997, 2016.

[95] Federico Silla. rCUDA: Virtualizing GPUs to reduce cost and improve performance. *Securities Technology Analysis Center (STAC) Summit*, 2014.

[96] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proceedings of the IEEE/ACM Conference on High Performance Computing, Networking and Storage Conference*, pages 47–, 2004.

[97] John Shalf, Sudip Dosanjh, and John Morrison. Exascale Computing Technology Challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, pages 1–25, 2011.

[98] M. B. Giles and I. Reguly. Trends in High-Performance Computing for Engineering Calculations. *Philosophical Transactions of the Royal Society of London Series A*, 372:20130319, 2014.

[99] Adaptive Computing. TORQUE Resource Manager, 2016. URL `http://www.adaptivecomputing.com/products/open-source/torque/`.

[100] Michael Showerman, Jeremy Enos, Avneesh Pant, Volodymyr Kindratenko, Craig Steffen, Robert Pennington, and Wen mei Hwu. QP: A Heterogeneous Multi-Accelerator Cluster. In *Proceedings of the 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[101] B. Varghese, J. Prades, C. Reaño, and F. Silla. Acceleration-as-a-Service: Exploiting Virtualised GPUs for a Financial Application. In *Proceedings of the 11th IEEE International Conference on e-Science*, pages 47–56, 2015.

[102] Vignesh T. Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling Concurrent Applications on a Cluster of CPU-GPU

Nodes. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 140–147, 2012.

[103] J. I. Agulleiro, F. Vázquez, E. M. Garzón, and J. J. Fernández. Dynamic Load Scheduling on CPU-GPU for Iterative Tomographic Reconstruction. In *Proceedings of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 603–608, 2012.

[104] L. Chen, O. Villa, and G. R. Gao. Exploring Fine-Grained Task-Based Execution on Multi-GPU Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 386–394, 2011.

[105] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. Towards Multi-tenant GPGPU: Event-driven Programming Model for System-wide Scheduling on Shared GPUs. In *Proceedings of the Workshop on Multicore and Rack-scale Systems*, 2016.

[106] G. A. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*, pages 33–44, 2013.

[107] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–30, 2011.

[108] M. Butler, K. Sajjapongse, and M. Becchi. Improving Application Concurrency on GPUs by Managing Implicit and Explicit Synchronizations. In *Proceedings of the 21st IEEE International Conference on Parallel and Distributed Systems*, pages 535–544, 2015.

[109] Priyanka Sah. Improving GPU utilization with Multi-Process Service (MPS). In *GPU Technology Conference*, ID S5584, 2015.

[110] Florian Wende, Thomas Steinke, and Frank Cordes. Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture. In *Zuse Institute Berlin Report*, pages 1–17, June 2014.

[111] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *International Conference on High Performance Computing and Simulation*, pages 24–32, 2011.

[112] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. Fine-Grained Resource Sharing for Concurrent GPGPU Kernels. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2012.

[113] NVIDIA. *CUDA Multi-Process Service*, May 2015. URL https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[114] Thomas Bradley. *Hyper-Q Example.* NVIDIA, 2013. URL https://www.ecse.rpi.edu/~wrf/wiki/ParallelComputingSpring2014/cuda-samples/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf.

[115] Bill Nitzberg, Jennifer M. Schopf, and James Patton Jones. PBS Pro: Grid Computing and Scheduling Attributes. In *Grid Resource Management*, pages 183–190, 2004.

[116] NVIDIA. *CUDA C Programming Guide 8.0*, 2016.

[117] Lee Howes. *OpenCL 2.1 Specification.* Khronos OpenCL Working Group, 2015.

[118] Patricia K. Immich, Ravi S. Bhagavatula, and Ravi Pendse. Performance Analysis of Five Interprocess Communication Mechanisms Across UNIX Operating Systems. *Journal of Systems and Software*, 68(1):27–43, 2003.

[119] Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos. Adaptive Scheduling Under Memory Constraints on Non-dedicated Computational Farms. *Future Generation Computing Systems*, 19(4):505–519, May 2003.

[120] James Patton Jones and Bill Nitzberg. Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–16, 1999.

[121] V. K. Naik, M. S. Squillante, and S. K. Setia. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 824–833, 1993.

[122] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The Impact of Operating System Scheduling Policies and Synchronization Methods of Performance of Parallel Applications. *SIGMETRICS Performance Evaluation Review*, 19(1):120–132, 1991.

[123] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-Ri Choi, and Jaehyuk Huh. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, 2012.

[124] Aman Kumar Bahl, Oliver Baltzer, Andrew Rau-Chaplin, and Blesson Varghese. Parallel Simulations for Analysing Portfolios of Catastrophic Event Risk. In *SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1176–1184, 2012.

[125] Cole Trapnell and Michael C. Schatz. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing*, 35(8-9):429–440, 2009.

[126] Mellanox. *RoCE in the Data Center*, 2014. URL http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf.

[127] Berkeley Vision and Learning Center (BVLC). Caffe Deep Learning Framework, 2017. URL http://caffe.berkeleyvision.org/.

[128] Google Brain Team. TensorFlow: An open-source software library for Machine Intelligence, 2017. URL https://www.tensorflow.org/.

[129] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, 2016.

[130] A*STAR. The Agency for Science, Technology and Research, 2017. URL http://www.a-star.edu.sg.

[131] Obsidian Strategics Inc. InfiniCortex transatlantic InfiniBand link, 2017. URL http://www.obsidianresearch.com.

[132] Blender Foundation. Blender: The Free and Open Source 3D Creation Suite, 2017. URL https://www.blender.org/.

[133] OTOY Inc. OctaneRender: Real-time 3D Rendering, 2017. URL https://home.otoy.com/render/octane-render/.

[134] The MathWorks, Inc. MATLAB: MATrix LABoratory, 2017. URL https://www.mathworks.com/products/matlab.html.

Graphics Processing Units (GPUs) are being adopted in many computing facilities given their extraordinary computing power, which makes it possible to accelerate many general purpose applications from different domains. However, GPUs also present several side effects, such as increased acquisition costs as well as larger space requirements. They also require more powerful energy supplies. Furthermore, GPUs still consume some amount of energy while idle and their utilization is usually low for most workloads. In a similar way to virtual machines, the use of virtual GPUs may address the aforementioned concerns. In this regard, the remote GPU virtualization mechanism allows an application being executed in a node of the cluster to transparently use the GPUs installed at other nodes. Moreover, this technique allows to share the GPUs present in the computing facility among the applications being executed in the cluster. In this way, several applications being executed in different (or the same) cluster nodes can share one or more GPUs located in other nodes of the cluster. Sharing GPUs should increase overall GPU utilization, thus reducing the negative impact of the side effects mentioned before. Reducing the total amount of GPUs installed in the cluster may also be possible.

In this dissertation we enhance one framework offering remote GPU virtualization capabilities, referred to as rCUDA, for its use in high-performance clusters. While the initial prototype version of rCUDA demonstrated its functionality, it also revealed concerns with respect to usability, performance, and support for new GPU features, which prevented its use in production environments. These issues motivated this thesis, in which all the research is primarily conducted with the aim of turning rCUDA into a production-ready solution for eventually transferring it to industry.

www.gap.upv.es/carregon **carregon@gap.upv.es**