UNIVERSITY OF THESSALY

DIPLOMA THESIS

A Study on the Computational Exploitation of Remote Virtualized Graphics Cards

Author:
Christos Konstantinos Matzoros

Supervisor:
Spyros Lalis
Co-Supervisors:
Nikolaos Bellas
Panagiota Tsompanopoulou

A thesis submitted in fulfillment of the requirements for the degree of Diploma Thesis

in the

Department of Electrical and Computer Engineering

Volos, August 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Μελέτη υπολογιστικής αξιοποίησης απομακρυσμένων εικονικοποιημένων καρτών γραφικών.

Χρήστος Κωνσταντίνος Ματζώρος

Η εικονικοποίηση συστημάτων αξιοποιείται για τη μείωση των δαπανών και τη βελτίωση της αποτελεσματικότητας των υπολογιστικών εγκαταστάσεων. Η ίδια ιδέα μπορεί να επεκταθεί και για τις κάρτες γραφικών. Η εικονικοποίηση των GPUs είναι ένας σχετικά καινοτόμος τομέας μελέτης και παραμένει μια απαιτητική προσπάθεια, καθώς οι πληροφορίες για τους drivers των GPUs δεν είναι προσβάσιμες για τροποποίηση λόγω προστασίας της πνευματικής ιδιοκτησίας. Η προσέγγιση του API Remoting, η οποία εφαρμόζεται σε υψηλότερο επίπεδο από τον driver, ξεπερνά τον παραπάνω περιορισμό. Το rCUDA είναι ένα από τα πιο καθιερωμένα frameworks εικονικοποίησης απομακρυσμένων καρτών γραφικών. Σε αυτή τη διπλωματική εργασία, διερευνούμε την απόδοση και τους περιορισμούς του rCUDA σε σύγκριση με την εγγενή εκτέλεση με CUDA, σε διαφορετικά σενάρια και εφαρμογές με ξεχωριστά χαρακτηριστικά. Κατασκευάσαμε μια εφαρμογή πολλαπλασιασμού πινάκων που κάνει ταυτόχρονη χρήση πολλαπλών GPUs, για να αναδείξουμε τα πλεονεκτήματα και τους περιορισμούς της ταυτόχρονης εκτέλεσης σε πολλαπλές απομακρυσμένες συσκευές με την χρήση του rCUDA. Η προαναφερόμενη έρευνα αποκαλύπτει την ανάγκη ανάπτυξης εργαλείων που θα μας επιτρέψουν να ελέγξουμε καλύτερα την απομακρυσμένη εκτέλεση κλήσεων. Προτείνουμε, σχεδιάζουμε και υλοποιούμε ένα απλό middleware που στοχεύει να παρέχει στον rCUDA client μία απόφαση σχετικά με την επιλογή απομακρυσμένου rCUDA server και συσκευής την στιγμή της εκτέλεσης.Πολλά προγράμματα CUDA έχουν την δυνατότητα να εκτελούνται ταυτόχρονα σε πολλές GPUs. Αυτό οδηγεί σε προβλήματα χρονοπρογραμματισμού, επειδή πρέπει να λάβουμε υπόψη με ποιό τρόπο τα μέρη του κώδικα που εκτελούνται σε διαφορετικές GPUs, αλληλεπιδρούν μεταξύ τους. Για να αποκτήσουμε γνώση σχετικά με αυτήν την αλληλεπίδραση, πρέπει να λάβουμε πληροφορίες σχετικά με τις κλήσεις CUDA. Δημιουργούμε ένα μηχανισμό που βασίζεται στην διαμεσολάβηση του middleware κατά την εκτέλεση των CUDA Runtime API κλήσεων. Με αυτό τον μηχανισμό μπορούμε να αποθηκεύουμε τα δεδομένα των κλήσεων και να καθυστερήσουμε την πραγματική εκτέλεση τους. Η ανάληση αυτών των δεδομένων θα μας επιτρέψει να εφαρμόσουμε περαιτέρω πολιτικές χρονοδρομολόγησης που αφορούν την διαχείριση πόρων, ενώ επίσης πετυχαίνουμε μεγαλύτερη διαφάνεια για τον χρήστη.

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma Thesis

A Study on the Computational Exploitation of Remote Virtualized Graphics Cards

by Christos Konstantinos Matzoros

Systems virtualization is used to reduce costs and improve the efficiency of computer systems. The same idea can be extended to virtualize GPUs. Virtualizing graphic processing units is a relatively innovative area of study and remains a challenging endeavor as the implementations of GPU drivers are not accessible for modification due to intellectual property protection reasons, and the information about them is limited. The API remoting approach, which runs at a level higher than the driver, overcomes the limitations mentioned above and is now the most common approach to GPU virtualization. rCUDA is one of the most well established remote GPU virtualization frameworks. In this Thesis, we investigate the performance and limitations of rCUDA compared with native execution with CUDA on different scenarios and distinct applications. We create a multi-GPU matrix multiplication CUDA application to highlight the advantages and identify the limitations of executing on multiple remote devices with rCUDA simultaneously. The aforementioned characterization reveals the need to develop tools that will enable us to control better the remote execution of remote CUDA Runtime API calls. We propose an implementation of a simple client/server middleware that aims to provide the rCUDA client with a decision considering the selection of remote rCUDA server and device at that current time. Also, many CUDA programs are capable of running concurrently on multiple GPUs. This leads to scheduling problems because we need to take into account the way parts of the code executed on different GPUs affect each other. To gain insight on this interaction, we need to acquire information about the CUDA calls. We create a mechanism to intercept the CUDA Runtime API calls and store the their information, while delaying the actual execution of the calls. That mechanism will enable us to implement further support resource management policies and greater transparency for the user.

Acknowledgements

I would first like to thank my thesis advisor, Prof. Christos D. Antonopoulos, for giving me the chance to work on intriguing projects in the last years of my undergraduate studies. I am thankful for his help and guidance throughout this work and I acknowledge that his high standards and unconditional support vastly improved the quality of the presented work. I would also wish to express my gratitude to Manolis Maroudas, for his valuable comments and general direction for the completion of this thesis. I would also like to thank Professors Spyros Lalis, Nikolaos Bellas, and Panagiota Tsompanopoulou for their participation in the evaluation committee of this thesis.

To my family, thank you for encouraging me and supporting me in all my pursuits. Additionally, I am more than thankful to my friends for all the unforgettable moments we have lived together over the past five years. This accomplishment would not have been possible without them.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the recent decades, many-core specialized processors and accelerators, a popular class of which are graphics processors (GPUs), enjoy increased adoption as an appealing approach to diminish the time-to-solution in domains as diverse as finance and physics simulations, image and video processing, machine learning algorithms and many others. The foremost difference between the CPU and GPU architecture is that CPUs are intended to manage a wide variety of tasks, but are restrained in the concurrency of tasks that can be running. A traditional CPU includes only a small number of computing units. The design of traditional processors is optimized for reducing the execution time of sequential code on every core, hence adding complexity to each core at the cost of offering fewer cores in the processor.

GPUs was primarily intended to render high-resolution images and video quickly by exploiting concurrency. The newer concept of GPGPU (General-Purpose Graphics Processing Unit) computing refers to the relatively modern trend of re-purposing GPUs for executing non-specialized computations in order to achieve higher performance on parallel applications, beyond their traditional purpose of rendering computer graphics. GPUs can perform parallel operations on multiple sets of data very quickly. Composed of thousands of processor cores working concurrently, GPUs facilitate massive parallelism where every core is focused on making efficient calculations. GPUs have a massively parallel array of integer and floating-point units and a dedicated, high-speed memory. This massively parallel architecture is what provides the GPU with its high computing performance. GPUs are based on a throughput-oriented model. This design empowers maximizing the execution throughput of applications with a high level of data parallelism, which are expected to be decomposable into a vast number of threads operating on several points in the program data space. Therefore, applications that produce a high degree of computations per data point could achieve higher performance.

The rapid growth in virtualization technologies has resulted in a paradigm shift towards on-demand computing, cloud computing, and Software as Service models. Smaller institutions lease computing resources from cloud vendors like Google and

Amazon, which use virtualization to facilitate efficient resource sharing among disparate applications/users to maintain high overall system utilization. Virtualization technologies are currently broadly deployed, as their application yields essential advantages such as resource sharing, isolation, and decreased management costs [19]. System virtualization allows several operating systems to run simultaneously on a single physical machine, achieving effective sharing of system resources. A hypervisor virtualizes physical resources such as the CPU, memory, and I/O devices.

We can extend this idea to accelerators, by virtualizing a GPU. With GPU virtualization, infrastructure operators can better manage the GPU accelerated virtualized devices while maintaining existing workflows for users and lowering overall operational expenses. We can transform a physical GPU to create virtual GPUs that can be shared across multiple virtual machines, providing a sense of isolation for every host, and sharing one or more powerful physical GPUs. A fundamental fallacy is often thinking that having more resources will help in getting better results, but this is surprisingly not the case. The reality is that we have to find ways to maximize the utilization of the resources that we already have.

Virtualizing GPUs is a relatively innovative area of study and remains a challenging endeavor. A fundamental limitation to this has been the implementations of GPU drivers, which are not accessible for modification due to intellectual property protection reasons. Furthermore, GPU architectures are not regulated, and GPU companies have introduced architectures with immensely diverse levels of support for virtualization. For these purposes, conventional virtualization techniques are not suitable for virtualizing GPUs. GPU companies do not give the necessary information considering the source code of their GPU drivers. Thus, it is challenging (using reverse engineering), if not impossible, to virtualize GPUs at the driver level. For that reason, the approach of API Remoting is increasingly used. This method aims to virtualize GPUs at a higher level in the GPU execution stack. API Remoting uses an RPC-like model to achieve higher-level GPU virtualization of the desirable API calls execution. There are plenty of frameworks that support GPU API Remoting like rCUDA [17], GVirtuS [12], and DS-CUDA [15]. Recent work discusses the performance efficiency of rCUDA among them [18].

The fundamental advantage of employing a framework like rCUDA is that it allows us to execute CUDA applications on nodes that do not have a GPU. With rCUDA, we can access a GPU even if we do not have one physically connected to our system, expanding our capabilities in computational power by running heterogeneous code for a given application. Furthermore, we have access to more devices that are being offered by a cluster, making more resources available for the user at a given time. Hence, we can run more compute-intensive applications that require more resources and computing units than the ones we have locally, which can be achieved only by a more significant number of devices. We can also increase the potential problem size that we have to solve by scaling it and split it to a higher number of remote

GPUs (scale-out approach). Having more devices available in our arsenal, we could combine our program executions in a way that provides better overall performance. The user can choose which devices the application will be executed on, enjoying more flexibility.

Even if we can utilize the rCUDA framework to execute CUDA applications on remote devices, the entire operation is performed "blindly", without knowing how the client interacts with the server. We do not know what and how much information would be exchanged between the two and how it can affect the overall performance of program execution. As soon as the user chooses the desirable remote device, the whole program will be scheduled to be executed on the specific device. That will happen even if the particular device is already full by other requests and even if other devices are fully available. In that way, it may overload an already congested with requests GPU. GPUs typically do not provide the information on how long a GPU request occupies the GPU, which creates task accounting problems. For that reason, GPU virtualization software faces several challenges in applying GPU scheduling policies.

In our pursuit of finding solutions to the aforementioned problem, the following questions need to be answered:

- What level of performance does the rCUDA framework achieve, relative to the native CUDA execution?
- What are the characteristics and useful information of rCUDA executions, and what are the limitations that impede us from achieving better results?
- What attributes of rCUDA make it useful to us?
- In what ways could we possibly control and automate the selection of the destination devices of the remote executions with rCUDA?
- What mechanisms do we need in order to acquire the critical CUDA call information in order to optimize CUDA job-to-node mapping?
- What mechanisms do we need to allow the implementation of further scheduling policies?

1.1 Contributions

The thesis is focusing on exploiting the relationship between the rCUDA clients and servers and on identifying what conditions can cause possible latencies on our application's remote execution. With that information in mind, we create a mechanism that will enable us to acquire and store CUDA calls information and to redirect rCUDA calls. This mechanism can be used to implement further scheduling policies for the remote execution of rCUDA calls to the desired remote device, based on that information.

The contributions of our work are the following:

- We create several realistic scenarios concerning the use of the rCUDA framework to test the applicability of it to different situations. We characterize the rCUDA framework with several simple applications of the CUDA SDK sample programs. Those programs are executed both with CUDA and rCUDA, to characterize the behavior of rCUDA and detect possible problems.
- After having completed rCUDA characterization, we implement a simple multi-GPU CUDA application to show the potential advantage of rCUDA against the use of CUDA applications that require the use of only local GPUs.
- We implement a middleware that supports the rCUDA framework and aims to control the execution when using many rCUDA clients and many rCUDA servers in our system, by redirecting whole CUDA applications to a selected remote device in our system if needed.
- We design and implement a mechanism that allows us to intercept CUDA Runtime API calls to control our program execution better and access all the information of the CUDA calls before the actual deployment and execution of kernel invocations. By batching the API calls and exploiting inherent synchronization points in the program, we can attain information that allows more educated code-to-device mapping decisions in order to deliver a higher level of utilization during remote execution. Moreover, our mechanisms provides a platform on top of which sophisticated scheduling policies can be implemented.

1.2 Thesis structure

The rest of this Thesis is organized as follows:

Chapter 2 provides background on GPU architectures and on the popular NVIDIA CUDA parallel computing platform. It also provides a quick review of GPU virtualization and API remoting. We close this chapter by explaining the architecture and various characteristics of the rCUDA framework.

Chapter 3 discusses the characterization of the main attributes of rCUDA, which is achieved by various performance evaluations on different setups and scenarios using rCUDA. We also present a Multi-GPU application to provide insight regarding the main benefit of rCUDA, which is its scalability due to being able to use multiple remote devices.

Chapter 4 introduces a middleware that supports the rCUDA framework by letting a central authority/coordinator decide on which remote device the rCUDA client will be deployed based on a specific algorithm. We present the different parts and functionality of the middleware, and we experimentally evaluate it.

Chapter 5 fist discusses how we can intercept a CUDA Runtime API call. We continue by implementing a mechanism that gathers information about the individual calls and delays the actual execution of them up to a specific synchronization point. We adjust the implementation to support multi-GPU applications by either using CUDA streams only, or with the use of POSIX threads as well.

Chapter 6 concludes with a summary of our main findings and presents directions for future work.

Chapter 2

Background

2.1 GPU Architecture

GPUs embrace a fundamentally different model for executing parallel applications in comparison with conventional multi-core processors. GPUs are based on a throughput oriented model and provide thousands of single cores and a high bandwidth memory scheme. In that model, when some of the threads are anticipating for the completion of memory accesses with high latency, different threads can be scheduled by the hardware scheduler to hide this latency. This mechanism may increase the execution time of individual threads, however it significantly improves total execution throughput. On the contrary, conventional processors typically use complex control logic and large cache memories in order to efficiently handle conditional branches, pipeline stalls, and poor data locality [1]. The two models are depicted in Figure 2.1.

Modern GPUs can also manage complicated control flows, have adequately large SRAM-based local memories, and implement some additional features of conventional processors, maintaining the fundamental properties of allowing a higher degree of thread-level parallelism and higher memory bandwidth.

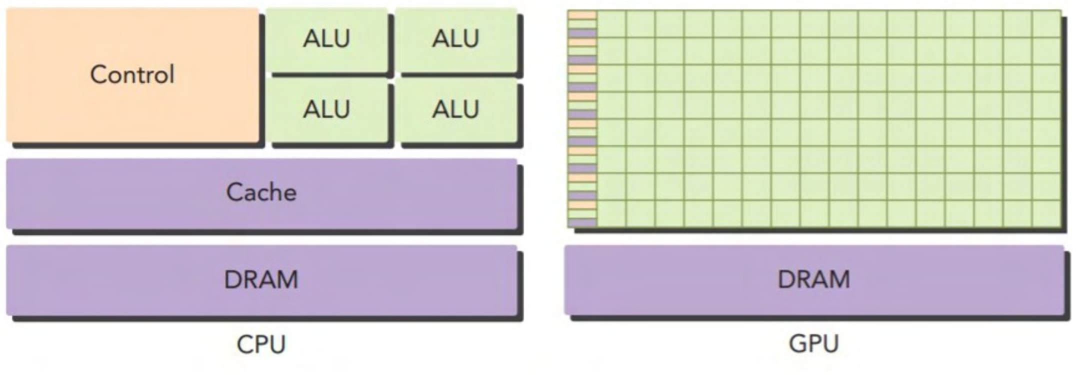


FIGURE 2.1: General model of a conventional CPU versus a conventional GPU [9]

Figure 2.2 depicts the architecture of a traditional heterogeneous system that includes a GPU. The GPU component is based on the basic architecture of NVIDIA

GPUs but is not restricted to NVIDIA architectures, as modern GPUs assume a similar high-level design. A GPU has numerous streaming multiprocessors (SMs), each of which has 32 computing cores. Each SM has an L1 data cache, and a low latency shared memory. Each core owns local registers, an integer arithmetic logic unit (ALU), a floating point unit (FPU) and several "special function units" (SFUs) that perform transcendental functions such as exponential or trigonometric calculations. A GPU memory management unit (MMU) provides virtual address spaces for GPU applications. A GPU memory reference by an application is resolved into a physical address by the MMU using the application's page table. Memory accesses from each application, cannot refer to other applications' address spaces. The host attaches the discrete GPU utilizing the PCI Express (PCIe) interface. Data transfers between the host memory and the GPU device memory can be initiated by the direct memory access (DMA) engine. Heterogeneous systems require substantial programming effort to control data handled by the CPU and the GPU.

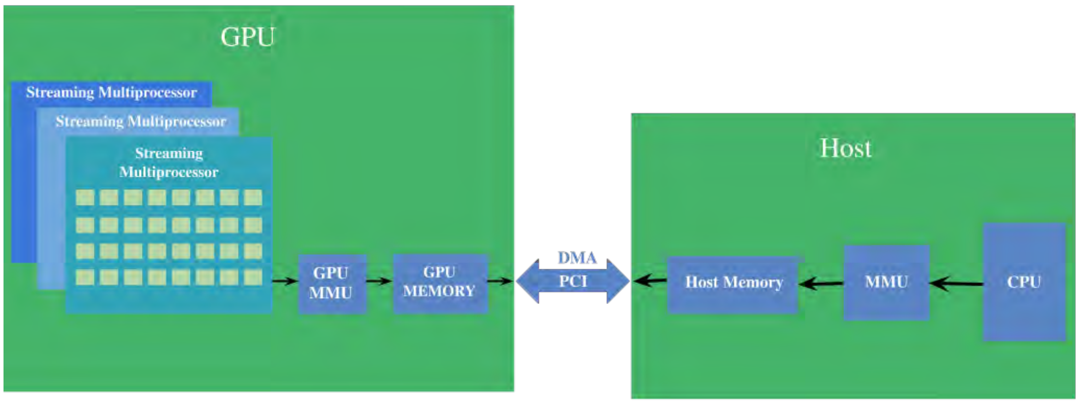


FIGURE 2.2: Architecture of a traditional heterogeneous system that includes a GPU

2.2 CUDA Programming Model

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming model created by Nvidia [23]. CUDA leverages the parallel compute engine in NVIDIA GPUs to solve complex computational problems more efficiently. Using CUDA, we can access the GPU for computation, as has been traditionally done on the CPU. The CUDA platform is available through CUDA-accelerated libraries, compiler directives, application programming interfaces, and extensions to industry-standard programming languages. In this thesis, we use CUDA C/C++ programming. CUDA C is an extension of standard C with some language additions to facilitate heterogeneous systems programming and straightforward APIs to handle devices, memory, and other tasks [9].

CUDA provides two API levels for controlling the GPU device and organizing threads.

The CUDA Driver API and The CUDA Runtime API. The respective structure is depicted in Figure 2.3.

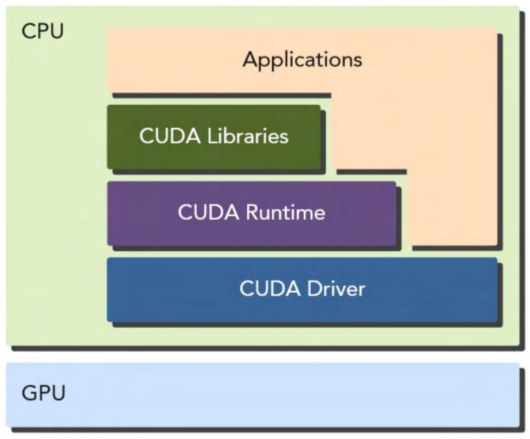


FIGURE 2.3: CUDA API structure [9]

The driver API is a lower level API and is relatively arduous to program, but it provides more control over how the GPU device is used. The runtime API is a higher-level API, implemented on top of the driver API. Each function of the runtime API is broken down into basic operations assigned to the driver API. The Runtime API is used more often by the programmers. There is no visible performance difference between runtime and driver APIs.

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the term host applies to the CPU and its memory, while the device refers to the GPU and its memory. The code that runs on the host can control both host and device memory. Also, it can launch kernels, which are functions executed on the device (kernel invocations). Many GPU threads execute these kernels in parallel. A kernel is defined using the "global" declaration specifier. The number of CUDA threads that execute that kernel for a given kernel call is specified using a the «<arg1,arg2,...»> execution configuration syntax along with some optional arguments. Each thread that executes the kernel is assigned a different thread ID accessible within the kernel through built-in variables. Threads are arranged as a grid of thread blocks, as depicted in Figure 2.4. Different kernels can have different grid/block configuration. Threads from the same block have access to shared memory, and their execution can be synchronized [20].

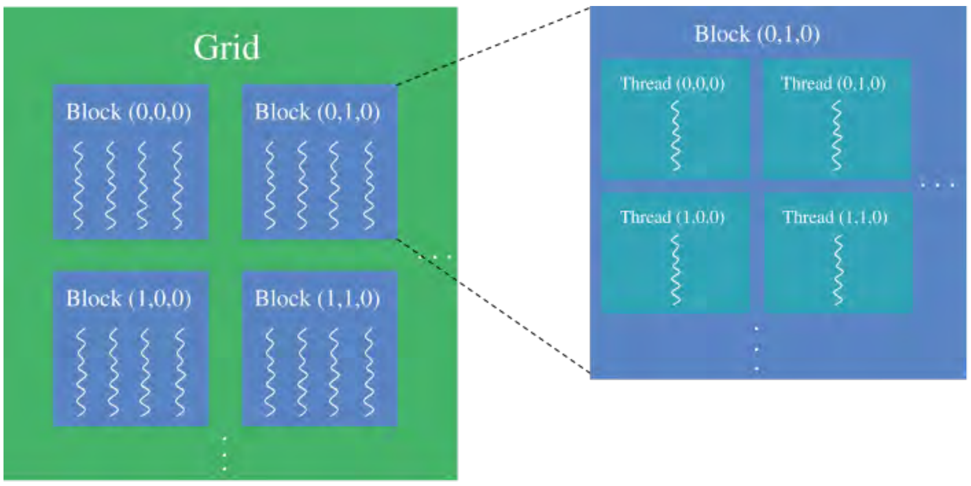


FIGURE 2.4: Grid of thread blocks in CUDA

The CUDA programming model implies a system comprised of a host and a device, each with their distinct memory. Kernels operate out of device memory. Thus, the CUDA runtime provides the appropriate API calls to allocate, deallocate, and copy device memory, as well as transfer data between host and device memory [4].

There are numerous different types of memory that each CUDA application has access to. For each different memory type, some trade-offs must be considered when designing the CUDA kernel for a given algorithm. Global memory offers ample capacity, but the latency to access this memory type is high. Shared memory has low access latency, yet the memory capacity is small compared to Global memory [16].

A typical sequence of operations for a CUDA C program is [6]:

1. Allocate host and device memory.
2. Initialize data to Host memory.
3. Transfer data from the host memory to the device memory.
4. Execute the kernel function (kernel invocation).
5. Transfer the results of the computation from the device to the host memory.

There are two main ways of attaining concurrency in CUDA simultaneously on multiple GPUs. The first one is by using multiple threads from the host environment, each one of them invoking a CUDA kernel. The most common way to do that is using POSIX threads (pthreads) or with the help of OpenMP. The second way is by solely using the CUDA API and with the help of CUDA Streams. CUDA applications manage concurrency by executing asynchronous commands in streams, sequences of commands that are executed in order. CUDA operations are placed within a stream such as kernel launches and memory copies.

The host places work in the queue and continues on immediately without blocking. Operations within the same stream are ordered (FIFO) and cannot overlap. Operations in different streams are unordered and can overlap while Operations in different streams are unordered and can overlap. Unless otherwise specified all calls are placed into a default stream, often referred to as *Stream 0*, is used. Operations in *Stream 0* cannot overlap other streams, thus we must create non-default streams to achieve concurrency.[21] [11] [7].

To also achieve concurrent memory copies using CUDA Streams:

1. CUDA operations must be in non-default streams
2. We must use asynchronous API calls along with pinned (Page-Locked) allocated memory on the host
3. Sufficient resources must be available
    (a) cudaMemcpyAsyncs in different directions

(b) Device resources (SMEM, registers, blocks, etc.)

2.3 Virtualization

Virtualization is a set of techniques that enable more efficient utilization of systems by creating a virtual computing environment, as opposed to a physical environment. The notion of virtualization can be implemented at different levels, as presented below. The virtualization mechanism can be employed at the node level, leading to popular and broadly used virtual machine frameworks. Examples of this technology are solutions such as VMware [22], Xen [24], or KVM [10], where numerous virtual machines can be concurrently executed in a real computer, sharing its resources and, hence improving overall utilization. The concept of virtualization can also be implemented at the device level, providing support to virtual machines. For instance, network adapters for technologies as different as InfiniBand or Ethernet offer virtualization features, which allow the network adapter to be replicated, at the logic level, so that distinctive replicas of the network card are assigned to different virtual machines. Similarly, graphics processing units (GPUs) have also recently been equipped with some virtualization support. For instance, the GRID K2 GPU by NVIDIA [13] can be shared among up to 32 virtual machines (per board), although it is intended for desktop virtualization.

The novel remote GPU virtualization technique also mentioned as API Remoting, implemented in frameworks like rCUDA, enables a set of GPUs to be simultaneously shared among several cluster nodes. In general, CUDA based virtualization frameworks offer the same or similar API as the CUDA Runtime API. As GPU vendors do not provide the source code of their GPU drivers, it is challenging to virtualize GPUs at the driver level. The main concept is that GPUs are logically separated from nodes, and an inventory of available GPUs is created. These GPUs can be accessed from any node in the cluster. Furthermore, a GPU may simultaneously serve more applications at once. This sharing of GPUs increases overall GPU utilization and reduces the total energy required to operate a computing facility, thus partially addressing the significant energy and power consumption concerns of future data centers and other computing facilities. Remote GPU virtualization also enables easier system upgrades, given that a cluster without GPUs can execute GPU-accelerated applications just by attaching one or more GPU servers to the cluster.

Virtualizing GPUs has been considered a more difficult task than virtualizing I/O devices, such as network interfaces or disks. Numerous reasons increase the complexity to the sharing of GPU resources between VMs. Firstly, GPU vendors tend to not reveal the source code and implementation details of their GPU drivers for practical reasons. Such technical specifications are crucial for virtualizing GPUs at the driver level. Secondly, even when driver implementations are revealed, GPU vendors still introduce significant advances with every new generation of GPUs to enhance performance. As a consequence, specifications unveiled by reverse engineering soon become useless. Besides, some OS vendors provide proprietary GPU drivers for virtualization, but these exclusive drivers cannot be used across all OSs. In summary, standard interfaces for accessing GPUs do not exist, which are needed for virtualizing these devices.

The API remoting approach overcomes the limitations mentioned above and is now the most common approach to GPU virtualization. The premise of API remoting is to provide a guest OS with a wrapper library with the equivalent API as the original GPU library. In most cases, a "split-device" scheme is used, where the frontend drivers are inhabited in the guest OS, and the backend drivers are inhabited in the host OS. The wrapper library of the guest OS intercepts a GPU call from the application and delivers it to the frontend driver. The frontend wrapper serializes and batches the GPU call parameters along with other essential information, into a transferable message and sends it to the backend in the host OS via shared memory or through the network. In the host OS, the backend driver parses the information and converts it into the original GPU call. The call handler executes the required action on the GPU through the GPU driver. The call handler delivers the result back to the application via the reverse path. This technique basically implements the logic of an RPC (remote procedure call) with the wrapper libraries being the stubs for both the client and the server side.

The main advantage of this approach is that it can support applications using GPUs without recompilation in most cases. The wrapper library can be dynamically linked to existing applications at runtime. API remoting provides much versatility to the way GPUs are utilized in a cluster, because this mechanism allows them to separately schedule, for a given application, the use of CPUs and GPUs [8]. The application can be assigned CPU cores in some nodes of the cluster while using the GPUs in a different set of nodes. Moreover, GPUs can be concurrently shared among distinct applications.

2.4 rCUDA Framework

rCUDA (or remote CUDA) is one of the commonly used API Remoting frameworks. It is a GPU virtualization framework that enables GPU acceleration by the use of remote CUDA-compatible GPUs. By leveraging the GPU virtualization approach, rCUDA achieves the disassociation of CUDA accelerators from the nodes where they are installed. This middleware supports up to version 9.0 of CUDA, implements the basic CUDA API, and provides support for the libraries included within CUDA, such as cuFFT.

rCUDA focuses on remote GPU-based acceleration, which offloads parts of CUDA computations onto GPUs located in remote hosts. rCUDA claims that previous virtualization research based on emulating local devices is not suitable for HPC applications, because of unacceptable virtualization overhead. Instead of device emulation, rCUDA implements virtual CUDA-compatible devices by adopting remote GPU-based acceleration without the hypervisor layer.

This framework is organized following a client-server model. The applications contact the client middleware, requesting GPGPU services. More concretely, rCUDA provides a CUDA API wrapper library to the client-side. The rCUDA client offers to the application the same interface as the NVIDIA CUDA Runtime and Driver APIs. Upon receiving a call request from the application, the client middleware processes it and forwards the corresponding requests to the rCUDA server. In turn, the server evaluates the requests and performs the expected processing by accessing the physical GPU to execute the corresponding request. Once the GPU has finished with the requested call's execution, the results are collected by the rCUDA server, which packs and sends them back to the client middleware. There, it eventually forwards the results to the application. The middleware's underlying architecture is depicted in Figure 2.5 [17].

To optimize client/server data exchange, rCUDA employs a customized application-level communication protocol and supports several underlying network technologies and interconnects. This is achieved by using a set of runtime-loadable, network-specific communication modules. Currently, three modules are available, one intended for TCP/IP compatible networks, another one designed explicitly for InfiniBand, which uses the RDMA feature of this network, and a third one intended for RoCE networks. The latter also leverages RDMA features [17].

The client-side of the rCUDA middleware is a library of wrappers that replaces the CUDA Toolkit dynamic libraries. Thus, CUDA applications that use rCUDA are not aware of accessing an external device. Also, they do not need any source code (or even binary) modifications. The rCUDA client is distributed in a set of files: $libcuda.so.m.n$, $libcudart.so.x.y$, $libcublas.so.x.y$, $libcut.so.x.y$, $libcusparse.so.x.y$, $libcurand.so.x.y$ and $libcudnn.so.x.y$. These shared libraries should be placed in those machines accessing remote GPGPU services. The user must set the $LD\_LIBRARY\_PATH$ environment variable to point to the location of these files (with priority over the original CUDA libraries). Furthermore, as the nvcc compiler links with CUDA static libraries by default, compilation flags enforcing the use of dynamic libraries are necessary to enable the use of the rCUDA software. The rCUDA server-side is configured as a daemon (rCUDAd) that runs in those nodes offering GPGPU acceleration services.

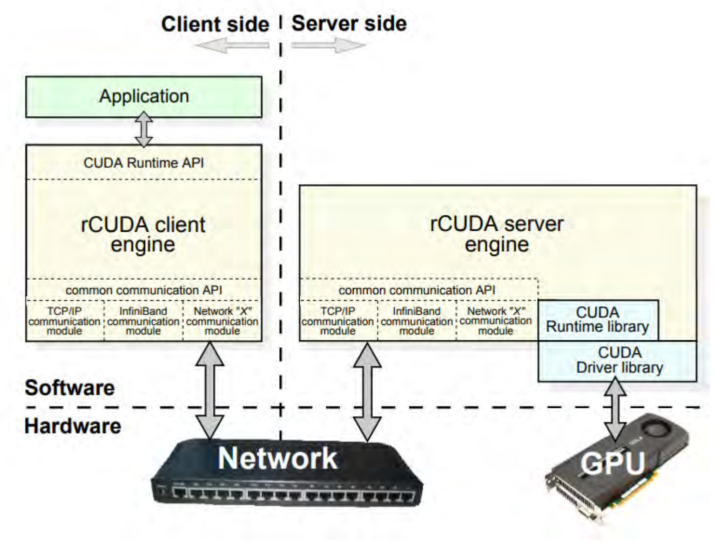


FIGURE 2.5: Basic architecture of rCUDA execution stack [17]

Chapter 3

rCUDA Framework: Experimentation and Performance Characterization

3.1 Overview

In this section, we test various scenarios considering the use of rCUDA, and we evaluate multiple characteristics of the rCUDA framework.

- We begin by outlining the characteristics of the devices that will be used for our experiments.
- We test the rCUDA framework against the CUDA SDK sample programs to recognize the framework's possible limitations considering the use of the CUDA API calls.
- Afterward, we test the functionality of rCUDA against different client/server setups and configurations.
- We expose a potential bug considering the utilization of the rCUDA framework.

Thenceforth, we characterize rCUDA performance. We again use the CUDA SDK sample programs which contain many distinctive CUDA calls and attain deviant behaviors in terms of computations and data transfers.

- We start by quantifying the cost of using the rCUDA framework versus using a GPU directly (overhead of virtualization)
- We discuss the overhead of using the rCUDA framework with the client and server being on the same machine versus being on different machines (overhead of communication).
- We proceeded by examining the impact of the framework on the bandwidth for both pageable and pinned memory.

- We consider scenarios in which we make use of the same GPU simultaneously by many clients and try to reverse-engineer the scheduling performed by the rCUDA server.
- We evaluate the advantages of rCUDA on programs capable of using more than one GPUs concurrently, by creating a simple multi-GPU program that performs a tilled matrix multiplication by splitting the computation to the available devices. We start by explaining the algorithm and its CUDA implementation. We continue by evaluating the performance using the rCUDA framework versus a local native execution, and we quantify the advantage of using rCUDA for multi-GPU applications. We conclude by revealing an rCUDA bug considering the use of CUDA streams and discuss how we can bypass it.

3.2 GPU characteristics and setups for the experiments

We employed four different heterogeneous computing nodes/machines that host one to four GPU devices each. All the nodes can communicate over the same network using TCP/IP. Table 3.1 summarizes the main characteristics of each one of these nodes.

System Code Name	**Artemis**	**Venus**	**Mars 1**	**Mars 2**
Number Of Available (same) Devices	2	2	4	4
Models of Each Device	Tesla K80	Tesla K80	GeForce GTX 690	GeForce GTX 690
CUDA Driver Version	10.2	10.2	10.2	10.1
CUDA Runtime Version	9.0	9.0	9.0	9.0
CUDA Capability	3.7	3.7	3.0	3.0
Number of Multiprocessors	13	13	8	8
CUDA Cores	2496	2496	1536	1536
Max Number of Threads per Multiprocessor	2048	2048	2048	2048
Global Memory (MB)	11441	11441	2000	2000
Max Dimension Size per Block	(1024,1024,64)	(1024,1024,64)	(1024,1024,64)	(1024,1024,64)
Memory Clock Rate (Mhz)	2505	2505	3004	3004

TABLE 3.1: GPU Characteristics

3.3 Testing the capabilities of the rCUDA framework

3.3.1 Functionality testing of the rCUDA framework

The rCUDA middleware aspires to imitate the functionality and behavior of the entire CUDA Runtime API [17]. There are, however, some calls that are not supported yet. The best way to identify unsupported API calls is to execute programs with distinct behaviors, using all calls in the CUDA API. We, therefore, experimented with most of the NVIDIA CUDA Toolkit program samples, which contain a wide variety of different calls. Appendix A summarizes the resulting behavior for the executions with rCUDA. The "Ok" indicates that everything works fine for the specific application and that it is executed as expected, giving the same results that we got by executing directly with CUDA. On the other side, if we deal with unexpected behavior in any program execution, we display the error code or the reason that the particular code does not work correctly.

From the results of the samples execution, we can confirm what is being reported in the documentation, namely that the graphics interoperability is not implemented yet. The following modules are not supported:

- OpenGL extensions (API for rendering 2D and 3D vector graphics)
- Direct3D 9, Direct3D 10, Direct3D 11 (also for supporting 3D rendering)
- VDPAU (Video Decode and Presentation API for Unix)
- Graphics (CUDA calls for graphics interoperability)

Also, as we can see the cudaMallocManaged() call is not supported so Unified Memory Management is not supported as well.

3.3.2 rCUDA Prerequisites and troubleshooting

In order to be able to use the rCUDA framework, we first have to satisfy two requirements. CUDA must be available to the rCUDA server running on the server node and specifically CUDA version 8.0 or CUDA version 9.0. Also, communications must be properly working between the nodes where the rCUDA clients and rCUDA server are employed, either for TCP/IP based communications (Ethernet) or for RDMA-based communications (InfiniBand or RoCE). There are simple steps to verify that everything works as expected. Firstly, we can use CUDA to execute the deviceQuery and bandwidthTest samples included in the CUDA distribution on the nodes where we intend to run the rCUDA servers. To check if communications are working properly between the client and server nodes (in case of RDMA-based communications) use the ib_write_bw and ib_read_bw tests included in the Mellanox OFED.

After the first run with rCUDA we may experience unexpected behavior, with two possible error codes appearing:

Error 1: “mlock error: Cannot allocate memory”

UNIX/Linux operating systems have the ability to limit the amount of various system resources available to a user process. These limitations include how many files a process can have open, how large of a file the user can create, and how much memory can be used by the different components of the process such as the stack, data and text segments. ulimit is the command used to accomplish this. For the ulimits to persists across reboots we need to set the ulimit values in the configuration file

/etc/security/limits.conf. In order to bypass the aforementioned error the rCUDA documentation suggests the following steps.

We must raise the locked memory size without limit. To set unlimited memlock we should add the following lines at the end of file /etc/security/limits.conf in both client and server nodes:

```
* hard memlock unlimited
```

```
* soft memlock unlimited
```

After rebooting the system, we can run the following command to verify that the limits have been changed:

```
$ ulimit -a | grep "max locked memory"
```

We should get an output similar to max locked memory (kbytes, -l) unlimited.

Error 2: “function cuGetExportTable not supported”

This error is caused because the application was compiled to use static libraries. The rCUDA framework needs applications to be compiled to use CUDA as a dynamic library. Therefore, a compilation using CUDA dynamic libraries is needed to allow the use of the rCUDA software. This requirement can be satisfied in two different ways:

1. If nvcc compiler is used, the argument -cudart=shared is needed.

2. If gcc/++ compiler is used, the -lcudart argument is needed.

3.3.3 Cross Execution

To further validate the functionality of the rCUDA framework by testing some more complicated scenarios, we employed two nodes, artemis and venus, enabling the device 0 (default) for each of them (Device 1.0 and Device 2.0). The two nodes are connected to the same network using a TCP/IP connection. We enable one rCUDA server and one rCUDA client on both of them. We made a successful cross-execution of the matrixMul CUDA sample problem, where the rCUDA client of artemis was using the device of venus and vice versa, concurrently, as depicted in Figure 3.1. By cross-execution, we refer to the scenario where node X uses the device of node Y, and node Y uses the device of node X. That experiment proves that we can use multiple rCUDA clients and rCUDA servers simultaneously and complete multiple cross executions of CUDA programs without interfering with one another.

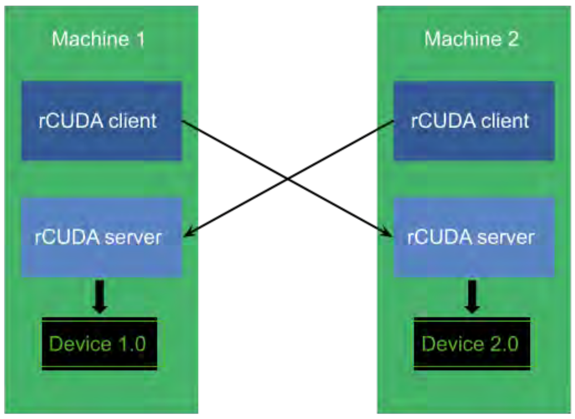


FIGURE 3.1: Cross-execution scenario using rCUDA

3.3.4 Simultaneous execution on the same rCUDA server

As a next step, we simultaneously executed the same program on the same device from multiple rCUDA clients started on different nodes. We enable device 0 (default) and initiate an rCUDA server on artemis. In this setup, we use six clients from three different nodes (venus, mars1 and mars2) with every one of them executing the matrixMul CUDA SDK sample simultaneously, as depicted in Figure 3.2. Every request was completed as expected, without any problem on either the server's or the clients' side. This experiment verifies that the rCUDA server can simultaneously handle multiple requests from different clients and create a point-to-point connection between one rCUDA client and an rCUDA server thread that satisfies the demands from the specific rCUDA client.

To reverse-engineer the potential scheduling applied on the server's side when executing simultaneously kernels from multiple clients on the same device, we created the following experiment: We set up on the same node (artemis), an rCUDA server that uses one device, and four rCUDA clients that run the same program at approximately the same time. The order of kernels' execution on the server side was different and unexpected, with every repetition of the same experiment. In the second phase, we set up four threads that run directly with CUDA, the same program on the same GPU. The behavior was very similar to the first experiment that uses the rCUDA framework.

The conclusion was that the rCUDA framework does not perform yet any kind of scheduling in the rCUDA server but rather follows a First Come First Served (FCFS) approach. When requests from several clients arrive, they are managed by independent processes (forked from rCUDA server) to guarantee isolation among the several clients. Since they are separate processes, the scenario is the same as when we run several CUDA applications using the same device.

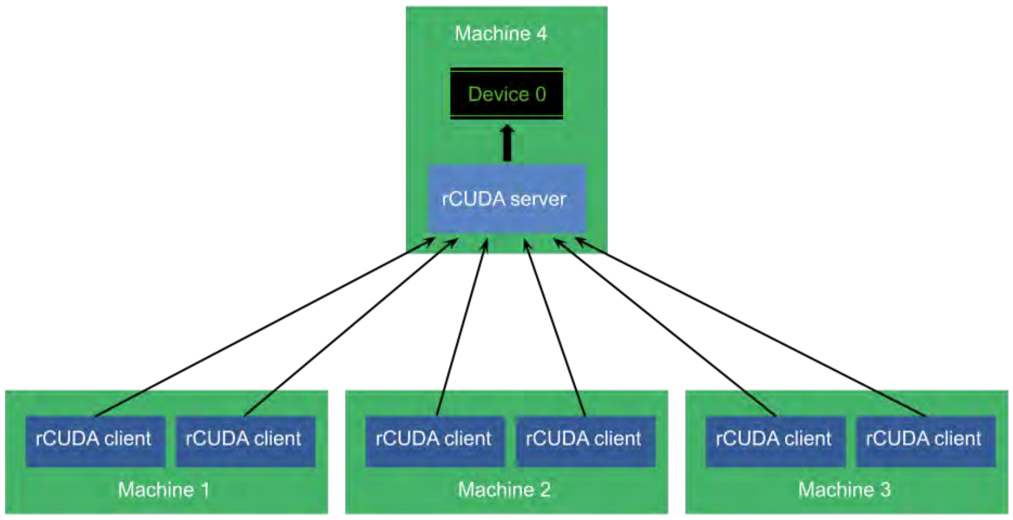


FIGURE 3.2: Simultaneous use of the same device by many clients using rCUDA

3.3.5 Multi-GPU scalability

To take full advantage of the rCUDA framework, we need to execute scalable applications that use more than one GPUs and therefore achieve better performance as we increase the number of available devices. Figure 3.3 depicts a scenario where we use the simpleMultiGPU CUDA SDK sample, which splits a vector addition problem into sub-problems deployed on the available devices. After all the separate computations have completed on the different devices, the host merges the intermediate results to create the final result. For the first experiment, we used two remote servers (on artemis and venus), enabling only device 0 on each one of them, and we executed the program with the rCUDA client being on a different node. For the second experiment, we enabled one more device for each of the servers, making four devices available in total. During the second scenario, we achieved almost half the execution time in comparison to the first experiment as the problem was scaled even further with double the initial devices. This scenario shows the advantage of the rCUDA framework, which is able to use multiple devices to scale a multi-GPU program and achieve better overall performance.

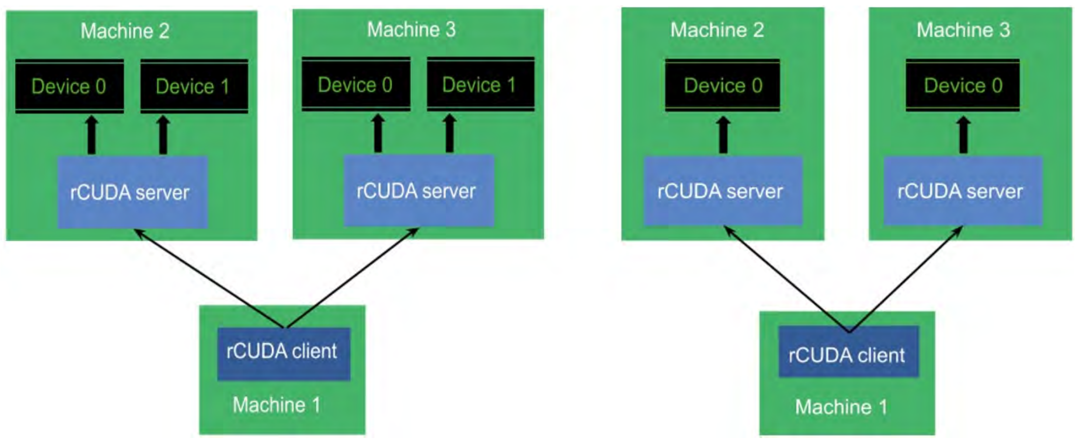


FIGURE 3.3: Multi-GPU program scaling with rCUDA

3.4 Performance Evaluation of rCUDA

3.4.1 Overhead of Virtualization

To evaluate the overhead of virtualization caused by employing the rCUDA framework, we set up an experiment where we compare the average execution time of the NVIDIA CUDA SDK sample programs using the rCUDA framework versus using CUDA directly to a GPU, to execute the same programs. In the first case, we employed the rCUDA server and rCUDA client on the same machine (artemis) to ensure that there is no network overhead involved. We then repeated the same executions using CUDA on the same device and node (artemis). Appendix B reports the average execution time on each of the two scenarios and overhead (%) of average execution time using the rCUDA framework over the simple native execution with CUDA.

From the outcome of the measurements, we can conclude that for all the programs of the CUDA samples, the average execution time is increased when we use the rCUDA framework, which is a very reasonable behavior considering that the rCUDA middleware appends an additional layer to the execution stack.

An unexpected result is that the overhead (even though the client and the server are at the same machine) is substantially higher for some applications. These applications mainly stress data transfers and make use of the CUDA streams (simpleStreams, simpleMultiCopy). We enlist a summary of the measurements in Table 3.2.

CUDA SDK Samples	Average execution time (seconds) executing directly with CUDA	Average execution time (seconds) executing locally with rCUDA	Percentage increase of average execution time of using rCUDA over directly CUDA
matrixMul	0.330	0.365	10.606
simpleStreams	2.787	57.856	1975.923
vectorAdd	12.198	12.208	0.081
simpleMultiCopy	0.502	2.514	400.796
convolutionFFT2D	4.287	4.557	6.298
convolutionSeparable	2.083	2.173	4.320
binomialOptions	31.427	31.503	0.241
BlackScholes	1.334	1.441	8.020

TABLE 3.2: Summary of Appendix B measuring the overhead of virtualization of rCUDA for the sample programs from NVIDIA

3.4.2 Local vs Remote Execution Evaluation

To evaluate the overhead of applying the rCUDA framework remotely, we designed an experiment where we measure the cost of using a remote GPU from an rCUDA server that is being placed on a different node from the one that the rCUDA client is located. In the first setup, we measure the average execution time of the NVIDIA CUDA SDK sample programs using the rCUDA framework with the rCUDA client located on the same machine (artemis) as the rCUDA server. In the second experiment we executed the same programs again, but this time we deployed the rCUDA client and rCUDA client on different nodes (artemis and venus respectively). Appendix C summarizes the average execution time on each of the two scenarios and the overhead (%) of using a remote rCUDA server over a local rCUDA server.

The objective of that experiment was to identify how much the network stack can affect performance. The results suggest that on all the different programs, the average execution time is increased when using the rCUDA server on a remote node versus using an rCUDA server on a local machine. This behavior was expected, as the network overhead (TCP/IP stack) is now a significant part of the execution flow. Once again, we can conclude from the percentage increase of average remote execution time that the programs that achieve lower performance are the ones that undergo more data transfers, as more data transfers are needed to be routed over the network. We enlist a summary of the measurements in Table 3.3.

CUDA SDK Samples	**Average execution time (seconds) executing locally with rCUDA**	**Average execution time (seconds) executing remotely with rCUDA**	**Percentage increase of average execution time of using rCUDA locally over using rCUDA remotely**
matrixMul	0.365	0.367	0.547
simpleStreams	57.856	428.251	640.201
vectorAdd	12.208	12.236	0.229
simpleMultiCopy	2.514	8.043	219.928
convolutionFFT2D	4.557	4.973	9.128
convolutionSeparable	2.173	2.540	16.889
binomialOptions	31.503	31.549	0.146
BlackScholes	1.441	1.912	32.685

TABLE 3.3: Summary of Appendix C measuring the network overhead by executing remotely with rCUDA for the sample programs from NVIDIA

3.4.3 Bandwidth evaluation

To measure the effect of rCUDA framework on memory bandwidth, we set up an experiment where we use the bandwidthTest program from CUDA SDK sample programs. We compare the host to device copy bandwidth for pageable and page-locked (pinned) memory, and the device to host copy bandwidth for pageable and page-locked memory (pinned).

Allocated host memory is by default pageable (it is mapped as pages), which means that it can be swapped out by other processes or the OS. Virtual memory offers the illusion of much more main memory than is physically available. The GPU cannot safely access data in pageable host memory because it has no control over when the host operating system may choose to transfer that data physically. When moving data from pageable host memory to device memory, the CUDA driver initially allocates temporary page-locked or pinned host memory, copies the source host data to pinned memory, and then transfers the data from pinned memory to device memory, as illustrated on the left side of Figure 3.4. The CUDA runtime allows us to allocate and use pinned memory directly. In general, Pinned memory is more expensive to allocate and deallocate than pageable memory, but it provides higher transfer throughput for large data transfers [9].

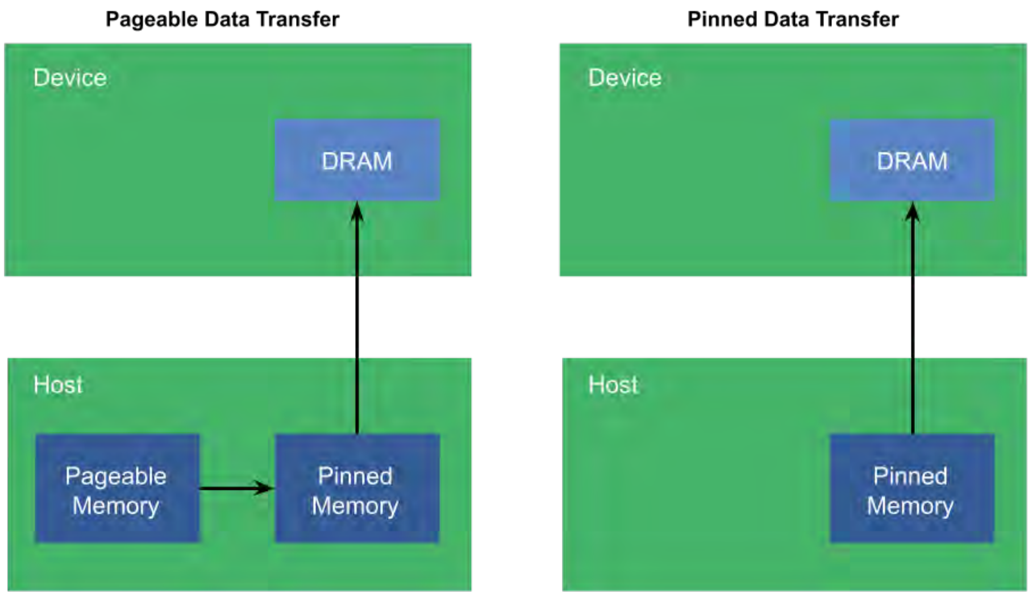


FIGURE 3.4: Pageable memory data transfer versus pinned memory data transfer illustration

For this experiment, we used device 0 (default) at the server side on artemis. In Setup 1, we measure the bandwidth in the case of executing directly with CUDA. n Setup 2, we run the bandwidthTest program using the rCUDA framework with the rCUDA client and the rCUDA server being on the same node (artemis). n Setup 3, we again use the rCUDA framework, with the rCUDA client on a different computer (venus) from the rCUDA server (artemis). For both the pinned and pageable memory, we transfer the default bytes (33554432 bytes) using quick mode (which performs a quick measurement of the bandwidth) [2].

For the pinned memory we execute for the host to device (–htod) and device to host (–dtoh) bandwidth:

```
./bandwidthTest –memory=pinned –mode=quick –htod
```

```
./bandwidthTest –memory=pinned –mode=quick –dtoh
```

As we can see in Figures 3.5 and 3.6, the bandwidth is decreased from Setup 1 to Setup 2. This is caused due to the overhead of virtualization and the internal functionality of rCUDA. We have a decrease in bandwidth from Setup 2 to Setup 3 due to the network stack overhead. We present a summary of the measurements in Table 3.4.

	Average host to device memory transfers bandwidth (MB/sec) using pinned memory	**Average device to host memory transfers bandwidth (MB/sec) using pinned memory**
Setup 1	5789.30	6440.65
Setup 2	2802.84	186.14
Setup 3	114.88	75.27

TABLE 3.4: Summary of pinned memory bandwidth measurements

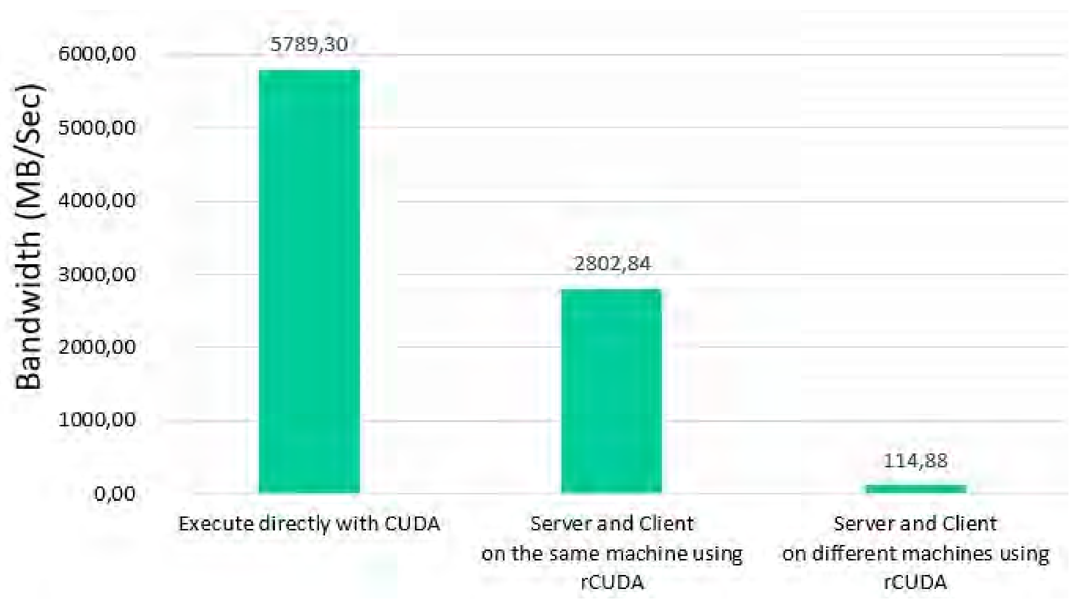


FIGURE 3.5: Average host to device Bandwidth for different setups using pinned memory

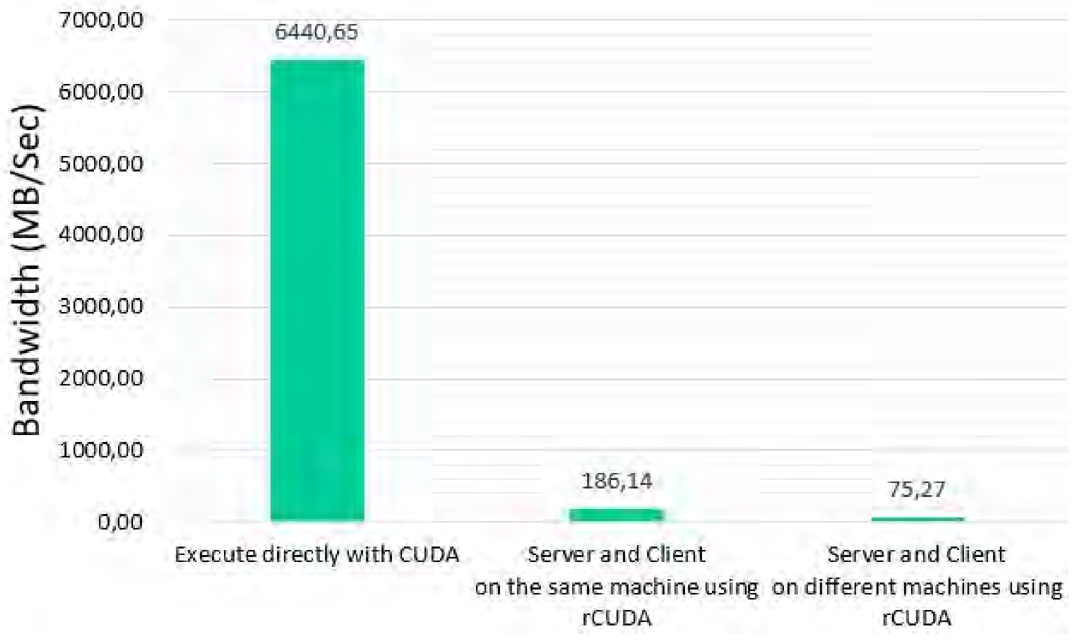


FIGURE 3.6: Average device to host Bandwidth for different setups using pinned memory

Interestingly, in Figure 3.7 we can see a significant difference for the second setup. When we use the rCUDA framework, the Device to Host bandwidth reduction is in the order of 97.11%, significantly worse than the respective penalty in Host to Device bandwidth, which equals 51.58%. This behavior can only be explained by the way in which the framework is designed internally.

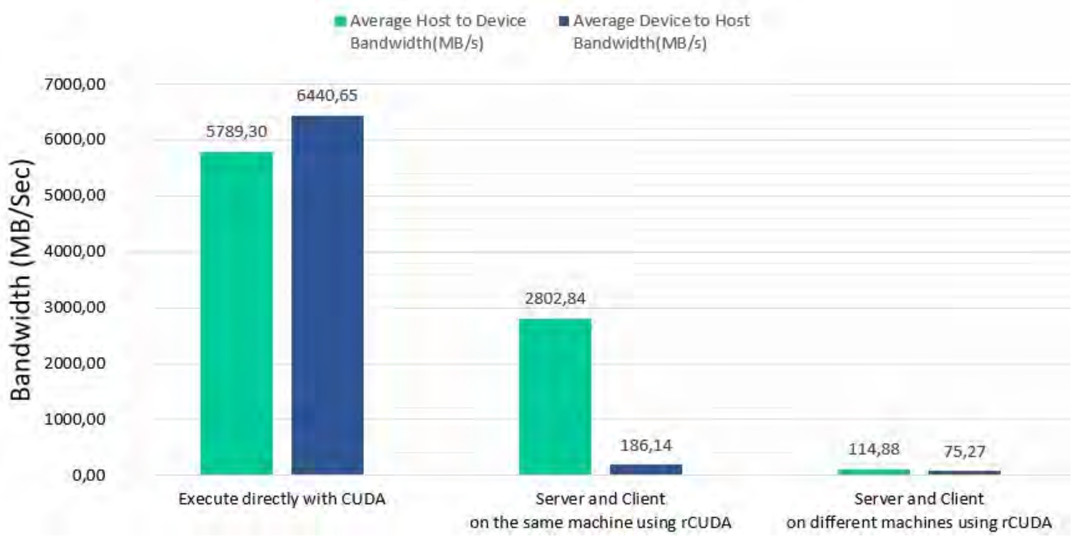


FIGURE 3.7: Comparison between average host to device bandwidth and device to host Bandwidth using pinned memory

For the pageable memory we execute for the host to device(–htod) and device to host(–dtoh) bandwidth:

```
./bandwidthTest –memory=pageable –mode=quick –htod
```

```
./bandwidthTest –memory=pageable –mode=quick –dtoh
```

Figures 3.8 and 3.9 show that the bandwidth is reduced from Setup 1 to Setup 2 due to the overhead of virtualization. We have also a bandwidth reduction from Setup 2 to Setup 3 due to the network stack overhead.

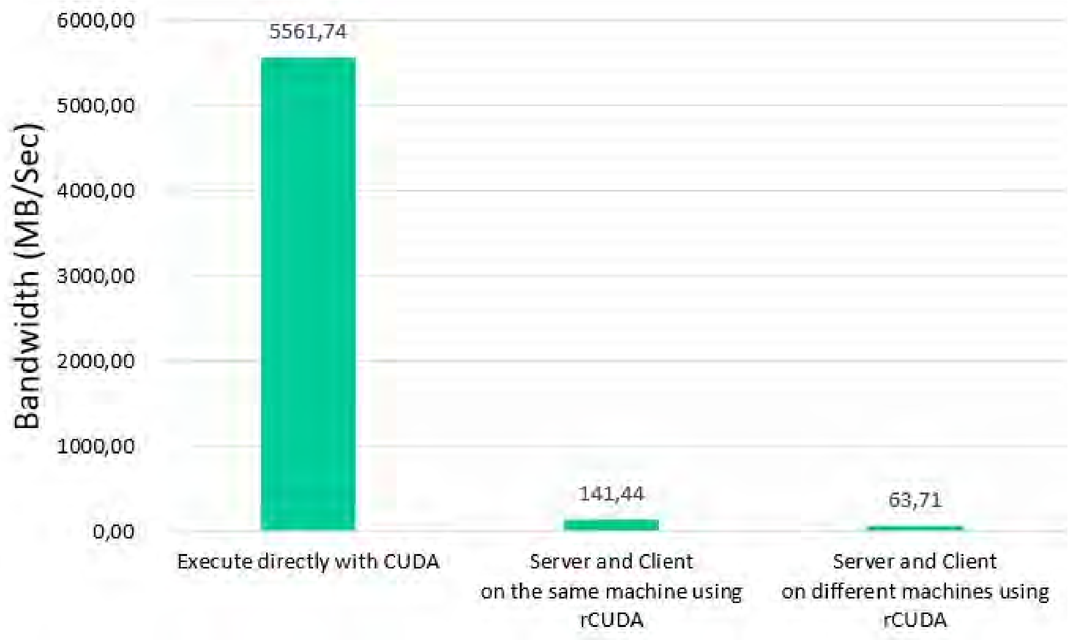


FIGURE 3.8: Average host to device Bandwidth for different setups using pageable memory

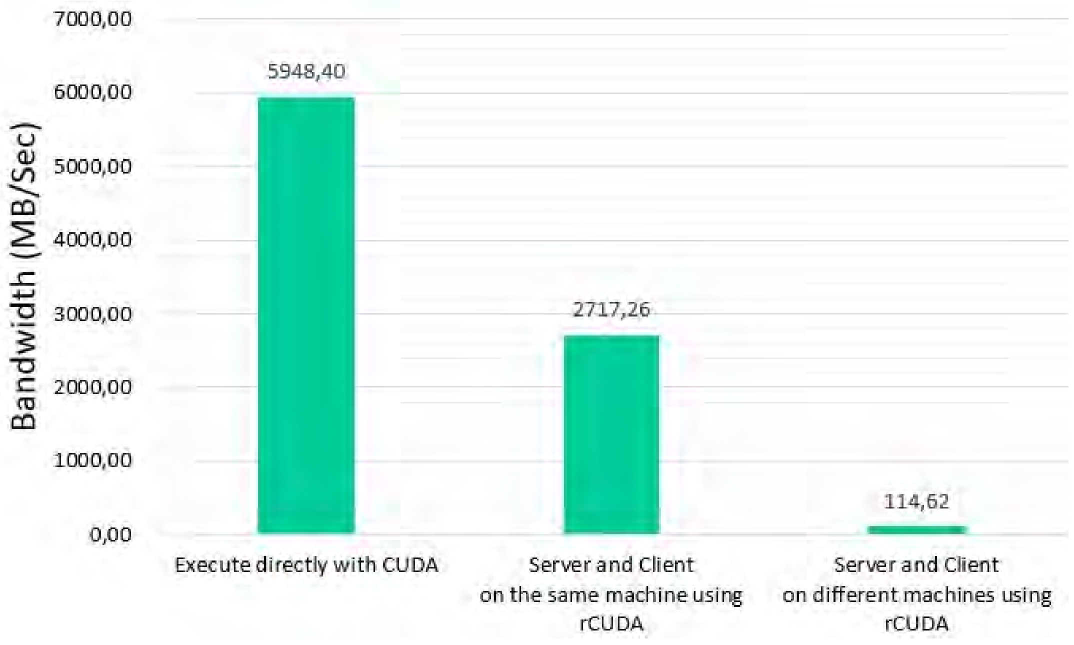


FIGURE 3.9: Average device to host Bandwidth for different setups using pageable memory

Figure 3.10 depicts that when we use the rCUDA framework, the reduction in Host to Device bandwidth which equals 97.45%, is significantly worse than the reduction in Device to Host bandwidth, which equals 54.31%. That is exacttly the reverse behavior compared to the pinned memory experiment. There is no obvious reason for this behavior and again it can only be explained by the way in which the framework is functioning internally.

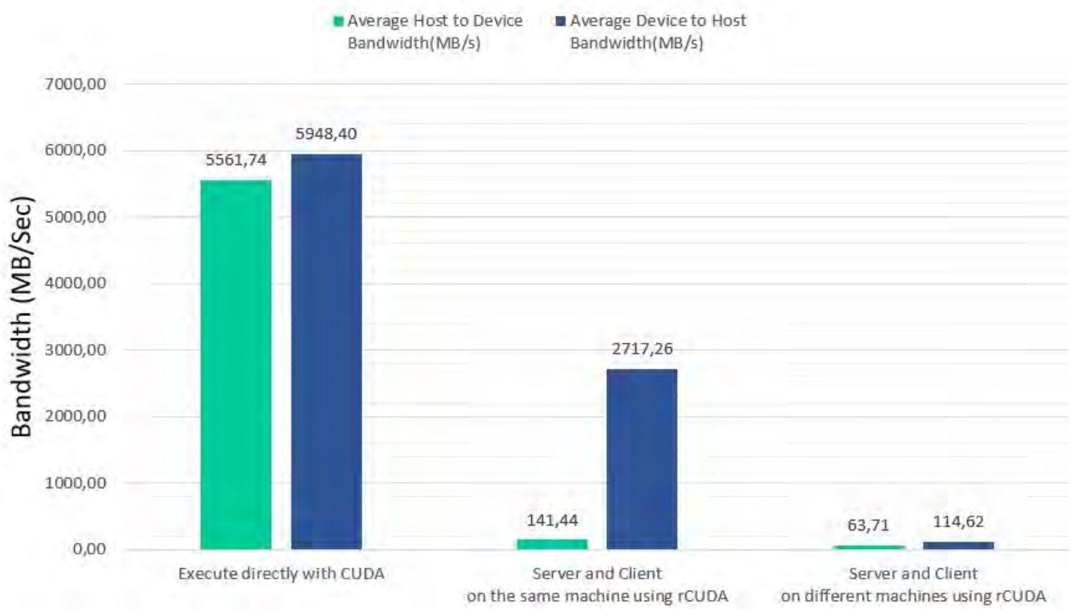


FIGURE 3.10: Comparison between average host to device bandwidth and device to host Bandwidth using pageable memory

3.5 Implementation of a multi-GPU matrix multiplication application in CUDA

We can take advantage of the rCUDA framework by using scalable applications that take less execution time by combining multiple GPUs. rCUDA can use all the GPUs of the cluster, instead of using the devices that are directly connected to a node which is the case when using CUDA natively. In some cases, rCUDA can achieve better results than executing only with CUDA. To expose and test the scalability that rCUDA can offer, we created a program that performs a matrix-matrix multiplication on multiple GPUs.

3.5.1 Algorithm Explanation

Before analyzing the multi-GPU implementation, it is helpful to recap how a matrix-matrix multiplication is computed. If we have two matrices, matrix A and matrix B, A is an $n \cdot k$ matrix (n rows and m columns), and B is a $k \cdot m$ matrix. The result of the multiplication $A \cdot B$ is an $n \cdot m$ matrix which we call C. To calculate the cell $c_1 1$ (on the first row and first column) of matrix C, we have to calculate the inner product of the elements of row 1 in matrix A, with the elements of column 1 in matrix B, as depicted in Figure 3.11.

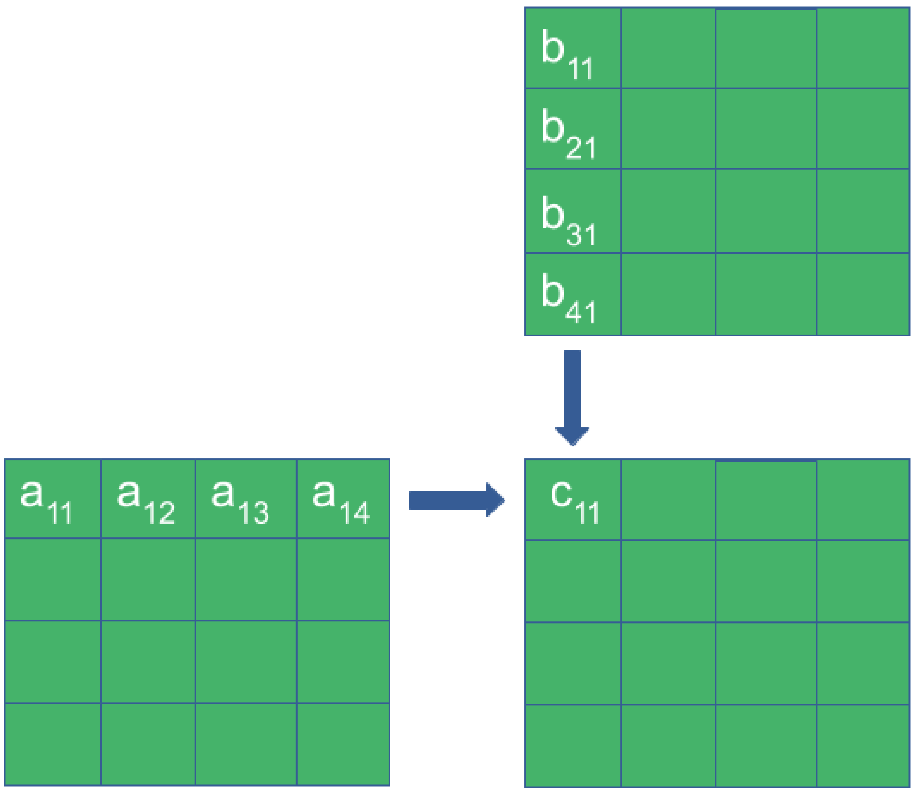


FIGURE 3.11: Illustration of matrix-matrix multiplication algorithm's inputs and output

So, in order to compute the cell c_{11} (where n=4 and m=4 for matrix C), we proceed with the following equation:

$$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} + a_{14} \cdot b_{41}$$

3.5.2 Multi-GPU implementation in CUDA

There is a steadily growing interest in using multiple GPUs concurrently to overcome the memory limitations of the single device and to further reduce execution times. The idea is to divide the problem further into individual computations that will be conducted simultaneously on different GPUs. The calculation of matrix-matrix multiplication will be divided into independent computations that will produce different parts of the final matrix, which will eventually be merged into the final matrix.

Our implementation divides the matrices A and B into four parts, allowing us to perform eight distinct computations in eight different GPUs. More precisely, to perform these computations, the host must first create the appropriate sub-matrices for the matrices A (A1, A2, A3, A4) and B (B1, B2, B3, B4) as shown in Figure 3.12. The ratio r $\in$ [0,1] selects the right number of rows and columns for the submatrices and is given by the user. For example, to divide the matrices into four submatrices with equal dimensions, the ratio should be equal to r = 0.5. After the creation of the submatrices, the computation is mapped to the different devices available. Figure 3.13 shows how the computation will be split. The equations for the resulting submatrices will be the following:

- $C1 = A1 \cdot B1 + A2 \cdot B3$
- $C2 = A1 \cdot B2 + A2 \cdot B4$
- $C3 = A3 \cdot B1 + A4 \cdot B3$
- $C4 = A3 \cdot B2 + A4 \cdot B4$

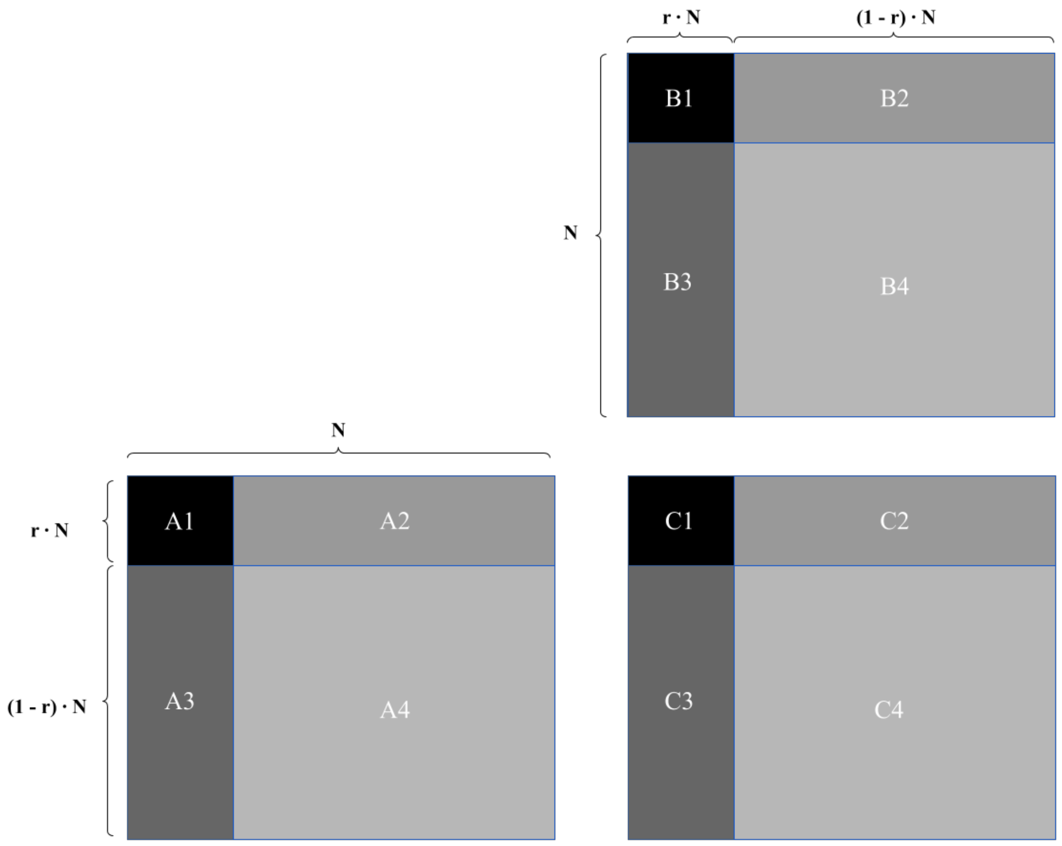


FIGURE 3.12: Division of matrices to distinct sub-matrices based on ratio r

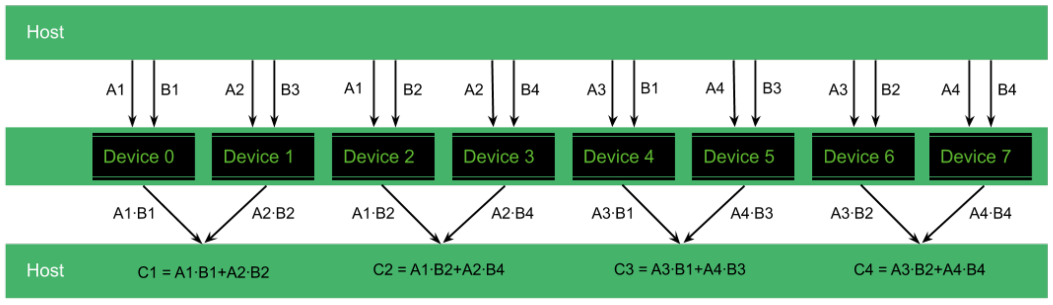


FIGURE 3.13: Flow of computations and data transfers for the algorithm running with CUDA

To implement this functionality in CUDA, we used the command cudaSetDevice(int device) [3], which specifies the device on which the active host thread executes the device code. We also use the concept of CUDA Streams to invoke the kernels simultaneously on many streams rather than the default. In that way, we can achieve concurrency.

The simplified CUDA code in Appendix D showcases the use of streams to achieve the desirable concurrency. The main functions that are being used are the cudaStreamCreateWithFlags() which creates an asynchronous stream, cudaMalloc() which allocates device memory, cudaMemcpyAsync() which copies data from the host to device memory and back, and cudaStreamSynchronize() which waits for a specific stream's tasks to complete. Listing 3.1 outlines the kernel function.

```
__global__ void matrixMultiply(double * A, double * B, double * C, int
    numARows, int numAColumns, int numBRows, int numBColumns, int
    numCColumns) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((Row < numARows) && (Col < numBColumns)){
        float Cvalue = 0;
        for (int k = 0 ; k < numAColumns ; ++k )
            Cvalue += A[Row*numAColumns + k] * B[k * numBColumns + Col];
        C[Row*numCColumns + Col] = Cvalue;
    }
}
```

LISTING 3.1: Matrix-Matrix Multiplication kernel implementation in CUDA

3.5.3 Evaluation using CUDA

As a next step, we executed this application and evaluated its performance on many GPUs. To assess the scalability of our application, we run it with 1,2 and 4 devices on mars1. Figure 3.14 shows the computation performance of the application executed with CUDA natively for different matrix sizes. As "Matrix Size" we assume to be the number of elements of the final matrix (C), that is equal to the size of the starting matrices (both A and B).

The results indicate a definite improvement in average execution time when we use more GPUs. For example, for problem size 6000x6000, the speedup in average execution time when we use 4 Devices is 3.804 times the one we get with only one Device. From Figure 3.14 it is evident that when we use more GPUs, the amount of problem size that can support is getting bigger since we have more available resources (such as memory). For example when we use 1 GPU we can only support problem size up to 6000x6000 (equals to $36 \cdot 10^6$ elements on the final matrix), while when we use 4 GPUs we can reach a problem size of 12000x12000 (equals to $144 \cdot 10^6$ elements on the final matrix).

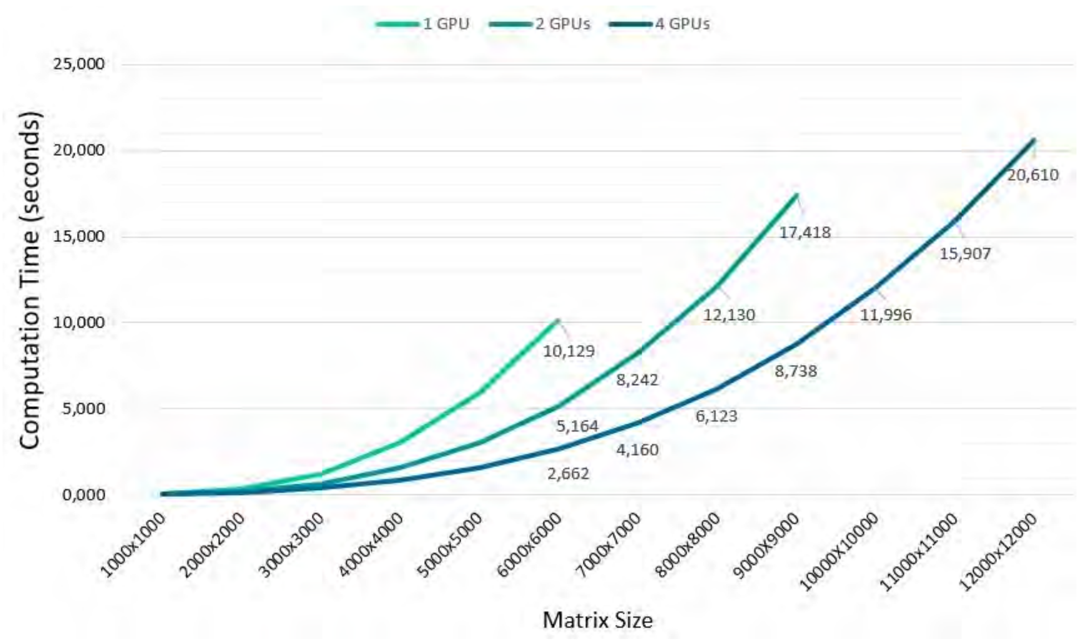


FIGURE 3.14: Computation performance of the multi-GPU matrix multiplication application using CUDA (natively)

3.5.4 Evaluation using rCUDA

To evaluate the performance that we achieve with the use of rCUDA, we performed the following experiment. We first created an rCUDA server (on mars1) with up to 4 devices at its disposal and one rCUDA client to execute our application. We tested our program's performance for 1,2 and 4 devices, respectively, and got the program's average computation time over different problem sizes. Again more remote devices allow us to compute bigger problem sizes. Figure 3.15 summarizes the execution flow when using rCUDA. Here we use all four available devices, and we see that the eight different sub-computations are distributed and executed simultaneously. When the computation is completed for the two corresponding submatrices such as the submatrices $A1 \cdot B1$ and $A2 \cdot B2$, we add these to create the submatrix C1, a piece of the final matrix C. Finally, we merge the submatrices to form the final matrix C.

In Figure 3.16, it is apparent that the computation time decreases as we increase the available devices in our system. For e problem size equal to 6000x6000 the speedup in average execution time when we use 4 devices is 3.610 times the one we get with only one device.

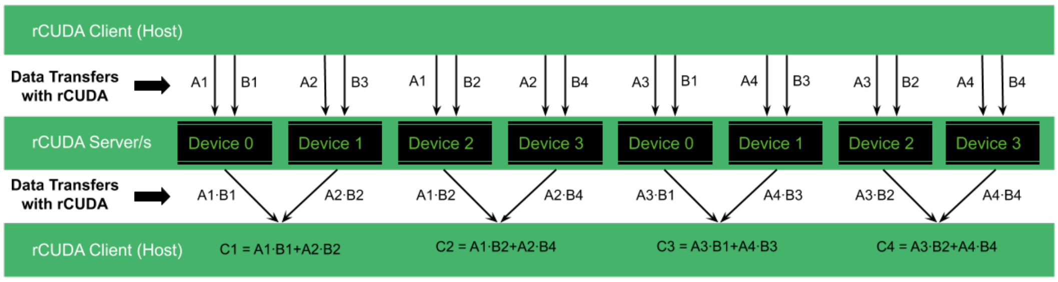

rCUDA Client (Host)
Data Transfers with rCUDA
A1 B1 A2 B3 A1 B2 A2 B4 A3 B1 A4 B3 A3 B2 A4 B4
rCUDA Server/s
Device 0 Device 1 Device 2 Device 3 Device 0 Device 1 Device 2 Device 3
Data Transfers with rCUDA
A1·B1 A2·B2 A1·B2 A2·B4 A3·B1 A4·B3 A3·B2 A4·B4
rCUDA Client (Host)
C1 = A1·B1+A2·B2
C2 = A1·B2+A2·B4
C3 = A3·B1+A4·B3
C4 = A3·B2+A4·B4

FIGURE 3.15: Flow of computations and data transfers for the algorithm running with rCUDA

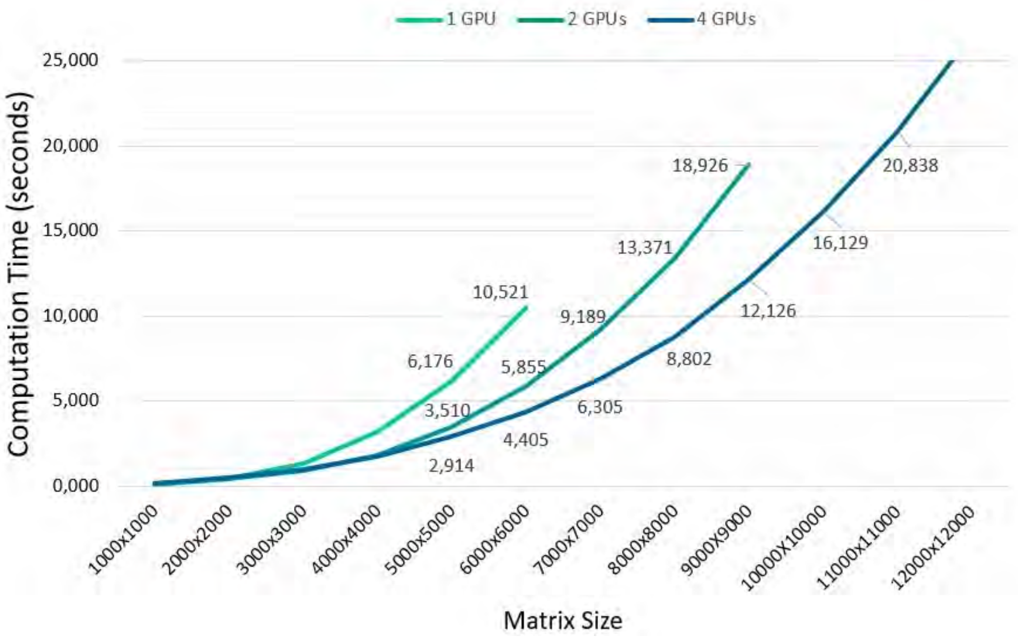


FIGURE 3.16: Computation performance of the multi-GPU matrix multiplication application using rCUDA (locally)

3.5.5 Comparison between CUDA and rCUDA performance

We continued by comparing the performance of using CUDA directly versus the scenario where we use the rCUDA framework for the aforementioned application. Based on the results that are depicted in Table 3.5 we can conclude that when we use rCUDA, the execution time is increased than the one we got by using CUDA natively. More precisely, by executing the program with only one available device, the execution time may increase. However, the speedup of executing with rCUDA is similar to the speedup of direct execution with CUDA, especially when we evaluate weak scaling (increase the problem size together with the number of GPUs). This is due to the fact that when scaling problem size, the part of execution time associated with computation scales faster than the part of the execution time related to data transfers. Similar behavior is observed when we use four devices for the same application, as depicted in Table 3.6. When we use rCUDA, the execution time is higher than the one we got by using CUDA natively. The execution time of using 4 GPUs are also smaller than using 1 GPU, as expected.

Finally, we examine the case where the use of rCUDA would provide higher performance. At first, we assume that we can execute the application directly with CUDA, considering that we have only one device available locally on our system. With the help of rCUDA, we can use four devices being offered by a cluster of devices. The measurements depicted in Table 3.7 suggest that when we use rCUDA with four available devices, the execution time is lower than when using only one device natively after a specific problem input size(approximately over 2500x2500), which gives us a ratio of executing with rCUDA versus executing natively with CUDA which is higher than one. For example the ratio is equals 1.281 when we use a problem size equal to 3000x3000. That ratio continues to increase further as we increase the problem input size. This is caused because when we use more devices, the delays due to the rCUDA data transfers do not affect us as much. In this scenario, we can scale our problem and split the computation into many devices, thus achieving concurrency that we could not acquire in any other way. Figure 3.16 displays the difference between the two setups.

	2000x2000	**3000x3000**	**4000x4000**	**5000x5000**	**6000x6000**
Executed natively on 1 GPU	0.378	1.253	3.013	5.904	10.128
Executed with rCUDA on 1 GPU	0.426	1.362	3.170	6.175	10.520
Ratio of $\frac{\textbf{executing with rCUDA}}{\textbf{executing natively with CUDA}}$	0.888	0.919	0.950	0.956	0.962

TABLE 3.5: Comparison between native execution using one device and rCUDA execution using one device

	2000x2000	**3000x3000**	**4000x4000**	**5000x5000**	**6000x6000**
Executed natively on 4 GPU	0.161	0.388	0.828	1.577	2.662
Executed with rCUDA on 4 GPU	0.496	0.977	1.770	2.913	4.404
Ratio of $\frac{\textbf{executing with rCUDA}}{\textbf{executing natively with CUDA}}$	0.324	0.397	0.467	0.541	0.604

TABLE 3.6: Comparison between native execution using four devices versus rCUDA execution using four devices

	2000x2000	**3000x3000**	**4000x4000**	**5000x5000**	**6000x6000**
Executed natively on 1 GPU	0.378	1.253	3.013	5.904	10.128
Executed with rCUDA on 4 GPUs	0.496	0.977	1.770	2.913	4.404
Time reduction (%)	**-30.931**	**21.986**	**41.259**	**50.652**	**56.510**
Ratio of $\frac{\textbf{executing with rCUDA}}{\textbf{executing natively with CUDA}}$	**0.763**	**1.281**	**1.702**	**2.026**	**2.299**

TABLE 3.7: Comparison between native execution with one device versus rCUDA execution using four devices

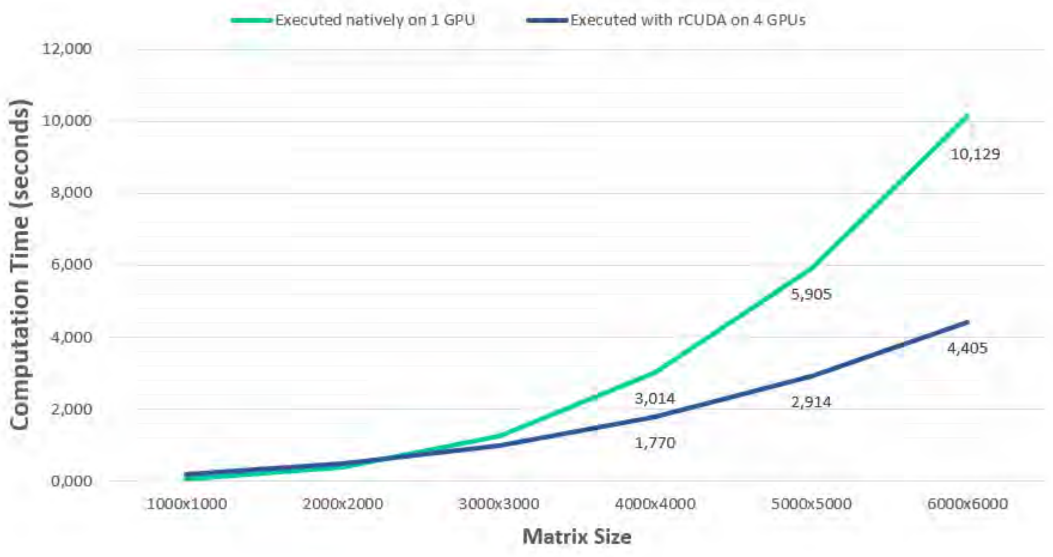


FIGURE 3.17: Comparison on computational performance of the multi-GPU matrix multiplication application executed natively with CUDA on 1 GPU versus executed with rCUDA on 4 GPUs

3.5.6 Unexpected behavior of rCUDA streams implementation

Throughout the implementation of the matrix multiplication application, we encountered an unexpected behavior considering the use of CUDA streams when we used rCUDA versus when we used the native execution with CUDA, which worked as expected.

The expected behavior, when we only use CUDA natively, is everything to be executed asynchronously. Therefore, all the data transfers and kernel invocations should be executed asynchronously, leading to the simultaneous execution of the kernel code among the available devices. The single point of synchronization in our code is when we use the call cudaStreamSynchronize (streamId) for every stream at the end of the program, where we wait for all the data transfers between the GPU and the host to be completed for each stream. When we execute the program, nothing blocks until this point, as expected.

On the contrary, when we use the rCUDA framework, the entire program blocks on cudaMemcpyAsync (DeviceToHost) without any reason, producing a serialized execution of our program. This is caused because we use that particular call between several kernel invocations for every stream. The symptom / result is an unexpected delay for the cudaMemcpyAsync (DeviceToHost) requests and blocking of each of them. This is an unexpected behavior that is not documented into any official document of rCUDA. We reported the specific bug to the rCUDA team.

To bypass this problem, we make sure to execute all the cudaMemcpyAsync (DeviceToHost) calls at the end of our program, after all the kernel invocations and before the cudaStreamSynchronize (streamId) calls. In that way, we achieve the desired parallelism among the multiple devices.

Chapter 4

Implementation of middleware for supporting rCUDA execution

4.1 General middleware design

A weakness of the rCUDA framework is that the rCUDA client (or the system administrator) needs to be aware of the architecture of the system and needs to explicitly specify which GPU each application is going to execute on. At the same time, rCUDA does not provide the clients with information on the status of remote servers / devices to allow them make educated decisions. We discuss an implementation of a simple middleware (in C++) that is employed to support rCUDA executions and aims to solve the aforementioned problem at execution time.

We propose a general architecture, as depicted in Figure 4.1. The system consists of a set of clients, a collection of servers, and a single broker/coordinator. The middleware client is employed at the same machine as the rCUDA client. he same applies for the middlewars server, which is employed at the same machines that the rCUDA servers are employed. The coordinator may be placed on any machine. The coordinator maintains the global view of the system state. It is responsible for communicating with all the servers to collect information for their individual state and for taking code-to-device mapping decisions on behalf of the clients. Each client communicates directly with the coordinator based on a simple request-reply protocol.

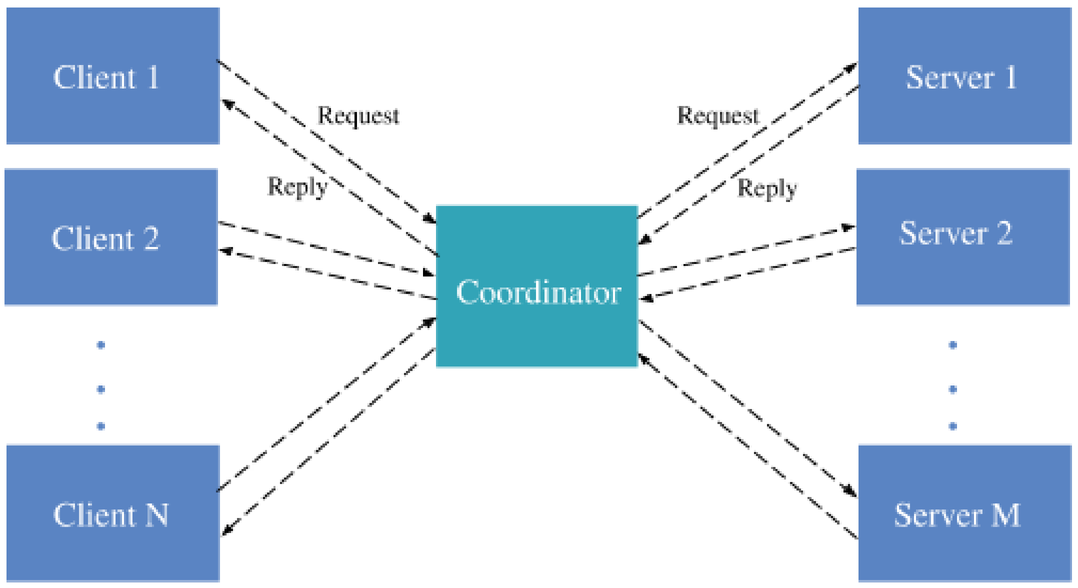


FIGURE 4.1: Basic middleware design

4.2 Middleware client

At the middleware client side, we start by executing the client program, which creates a new Linux Shell. The client's design is depicted in Figure 4.2. The users have different options and can use specific commands to proceed with the desired action. The environment waits for the user to enter a command. The user's two main options are detailed below:

- $ native /absolute/path/to/CUDA/program
This command will execute the CUDA program specified by the user locally on the same node, directly with CUDA if there is any available device.

- $ remote /absolute/path/to/CUDA/program
With this command, the middleware will first communicate with the coordinator with a request message asking for information about the address and identifier of a remote device. The coordinator will eventually respond with a reply message containing information about the chosen device. With that information, the client process creates a new process that executes a bash script. This script is responsible to set the required environment variables for rCUDA execution such as LD_LIBRARY_PATH=../rCUDA/lib and RCUDA_DEVICE_X=server_ip_address: deviceId. The same bash script will initiate the CUDA program will be executed using the rCUDA framework. The communication protocol that is used here between the client and the coordinator is an at-least-once delivery protocol, which means that for each request handed to the mechanism, multiple attempts are made at delivering it, such that at least one succeeds. In more casual terms, messages may be duplicated but not lost. That requires maintaining state at the sending end and having an acknowledgment mechanism at the receiving end.

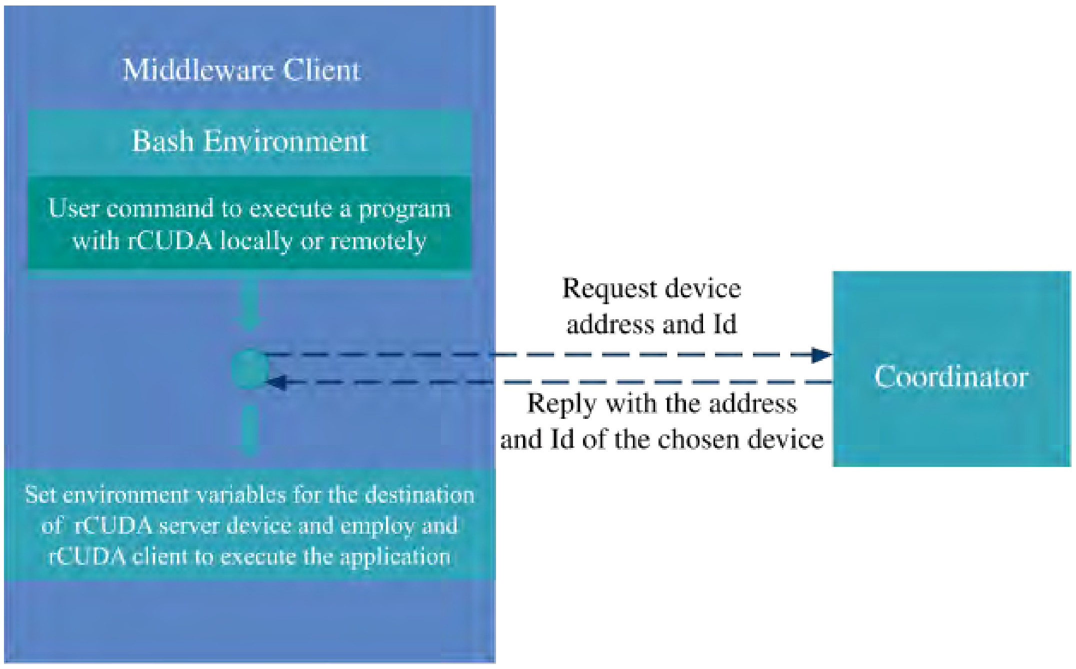

Middleware Client
Bash Environment
User command to execute a program with rCUDA locally or remotely
Request device address and Id
Reply with the address and Id of the chosen device
Coordinator
Set environment variables for the destination of rCUDA server device and employ and rCUDA client to execute the application

FIGURE 4.2: Middleware client design

Some other useful command options are the following:

- [H] or [h]

It provides a simple explanation for the user on how to execute a simple program as well as some command examples for both native and remote execution.

- [C] or [c]

It gives the user the ability to change the coordinator's address information (IP address and port).

4.3 Middleware server

On the server side, the middleware server's primary responsibility is to provide the coordinator with useful information about the status of the available devices at the particular machine. The functionality is depicted in Figure 4.3. To launch our server, we must first initiate the main program by giving a specific IP address and port on which a thread is waiting to receive request messages from the coordinator. Simultaneously, another thread collects information about the available devices of the system by executing a bash script periodically every second.

This script uses the nvidia-smi [14] command-line utility tool, which monitors and manages NVIDIA devices such as Tesla, Quadro, GRID, and GeForce. It is installed along with the CUDA toolkit and provides us with meaningful insights, such as the "Memory-Usage", which reports the memory allocation on GPU out of total memory and the "GPU-Util", which indicates the percent of GPU utilization (percent of the time during which a specific device was occupied executing CUDA kernels).

When we assemble this information from all the server's devices, we can identify the device with the minimum utilization at that specific time. This information is stored in a global structure. When a new request is received, coming from the coordinator, the receiving thread fetches the data from the global structure and responds to the coordinator by sending back a reply message, which includes information about the device with the lowest utilization. The coordinator can then use this information to drive application-to-device allocation decisions.

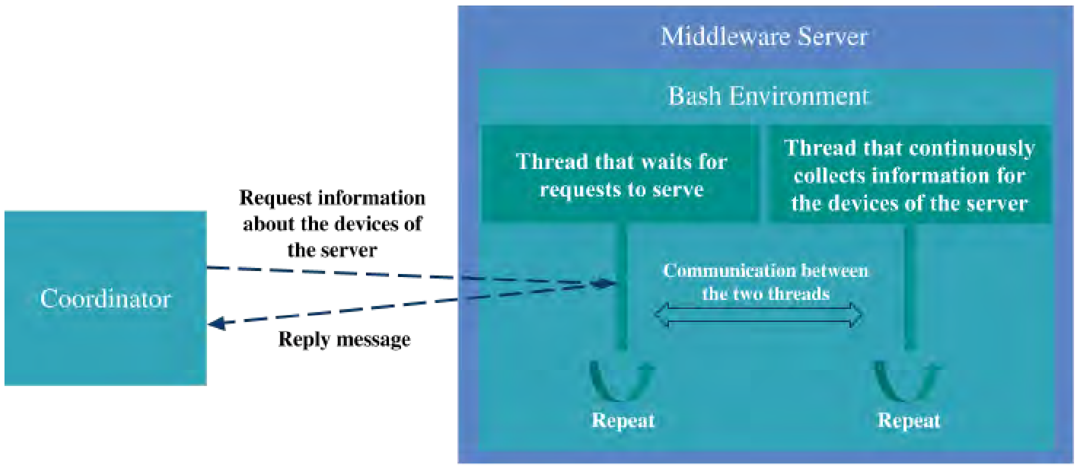

Middleware Server
Bash Environment
Thread that waits for requests to serve
Thread that continuously collects information for the devices of the server
Request information about the devices of the server
Coordinator
Communication between the two threads
Reply message
Repeat
Repeat

FIGURE 4.3: Middleware server design

4.4 Middleware coordinator

The coordinator's primary responsibility is to decide, on behalf of the clients, the server and device client rCUDA code should be executed on. To do that, it uses information continuously collected from rCUDA server nodes. Its main functionality design is depicted in Figure 4.4.

At first, we create a thread that anticipates request messages from clients. These messages ask the coordinator to specify a target device for rCUDA code execution, based on the scheduling algorithm the coordinator applies and its overview of system status. When the algorithm finds a suitable device, it sends a reply message back to the client, which contains information about the IP address and port of the chosen server and the device id of the selected device. In the current implementation, the coordinator assigns devices to the different clients in a round-robin way, without considering the information we have from the servers. However, the coordinator is an excellent platform for implementing adaptive scheduling policies based on runtime information.

Secondly, the coordinator can collect valuable information from the servers. This functionality is optional, based on the needs of the scheduling algorithm. The scheduling algorithm may apply this information to reach more sophisticated scheduling decisions, according to system status. If we want to collect and use the data from the servers, we spawn a second thread. This thread has the responsibility of periodically asking all the servers for information considering their local devices. In our implementation, the coordinator sends a request message to all the servers. Each server responds with a reply message that contains the device with the lowest utilization on that server at a particular time[1].

If one or more servers are not available or if there is any connection error, then the coordinator does not consider the respective servers and devices as part of the system. Moreover, it notifies the administrator so that corrective action can be taken.

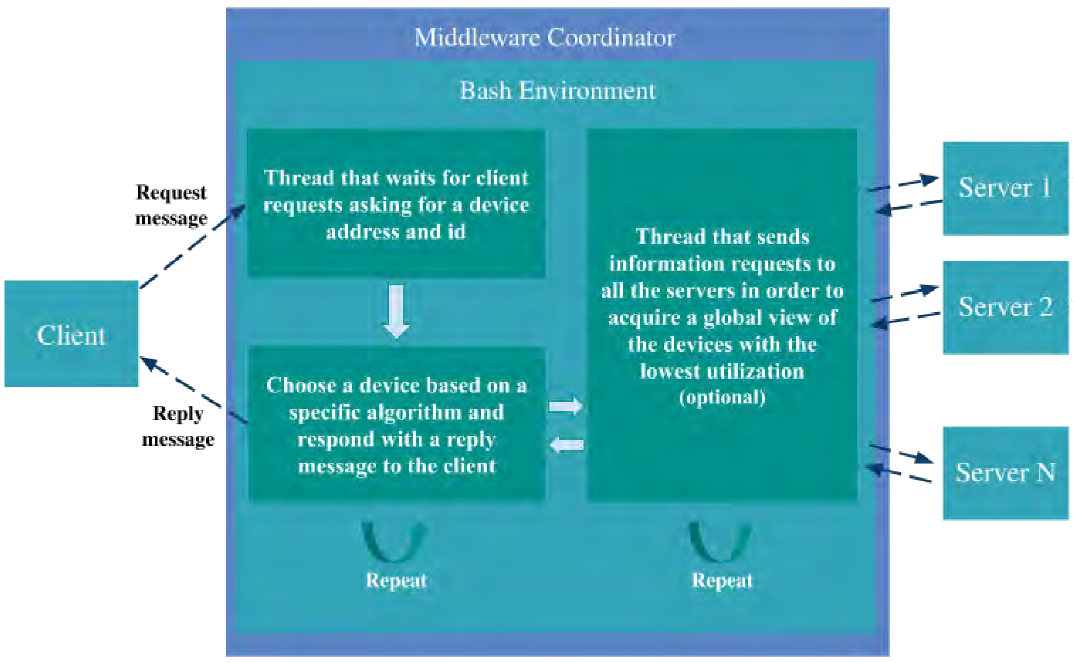

Middleware Coordinator
Bash Environment
Thread that waits for client requests asking for a device address and id
Choose a device based on a specific algorithm and respond with a reply message to the client
Thread that sends information requests to all the servers in order to acquire a global view of the devices with the lowest utilization (optional)
Repeat
Repeat
Request message
Reply message
Client
Server 1
Server 2
Server N

FIGURE 4.4: Middleware coordinator design

4.5 Middleware evaluation

We executed an experimental scenario in which we quantify the performance of multiple executions of a CUDA application using two different setups. In the first experiment, we use rCUDA to execute all instances on the same remote device. On the second one, we use multiple remote devices, and we let the middleware select one of the available remote devices[2]. In both cases, we execute the FDT3D CUDA

[1]It is straightforward to adapt the server- and the coordinator-part of the middleware to collect additional information.

[2]As we discussed earlier, at this point the scheduling algorithm is a naive, non-adaptive round-robin implementation.

SDK sample program. This program applies a finite differences time domain progression stencil on a 3D surface. We concurrently run this program ten times, and we measure the overall execution time on both experiments.

For the first experiment, we use the rCUDA framework, as depicted in Figure 4.5. We set an rCUDA server at artemis, and we execute the FDTD3D program ten times from mars1, employing ten rCUDA clients at the same time. The execution is taking place concurrently, and the only scheduling performed is being done by the CUDA driver internally on the rCUDA server's device. The total execution time is 86.71 seconds.

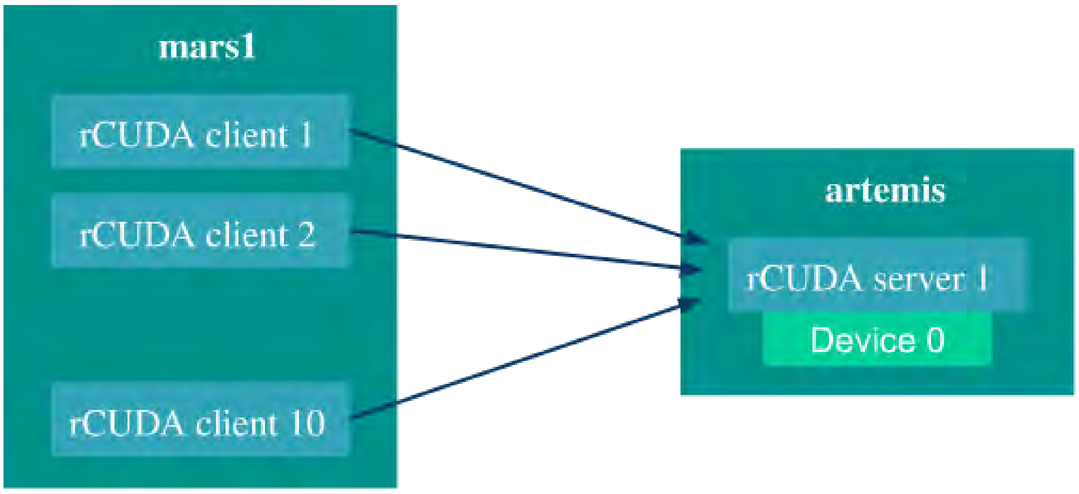


FIGURE 4.5: Experiment setup using directly rCUDA

For the second experiment, we use the middleware, as depicted in Figure 4.6. We set two rCUDA servers, on artemis and venus (one on each node), each of them controlling two GPU devices. We also deploy the coordinator on artemis. We deploy the middleware client on mars1, and after entering the necessary information about the coordinator, we execute the FDTD3D program ten times, using the "remote" keyword. All the executions start at the same time as we spawn a new thread for every execution to avoid blocking. For every remote program execution, the client requests the coordinator for a device address and id, based on the protocol discussed earlier. In our case, the coordinator responds to the clients with a device id using the round-robin algorithm. The result is for the multiple executions to be distributed on all the available devices rather than being executed on a single GPU. Thus, this approach achieves a better overall performance as the needs for resources are distributed among the different devices, and we also achieve concurrency as we use more than one GPUs at the same time. The total execution time equals to 67.31 seconds, reaching a speedup of 1.29 over the total execution time observed during the first experiment.

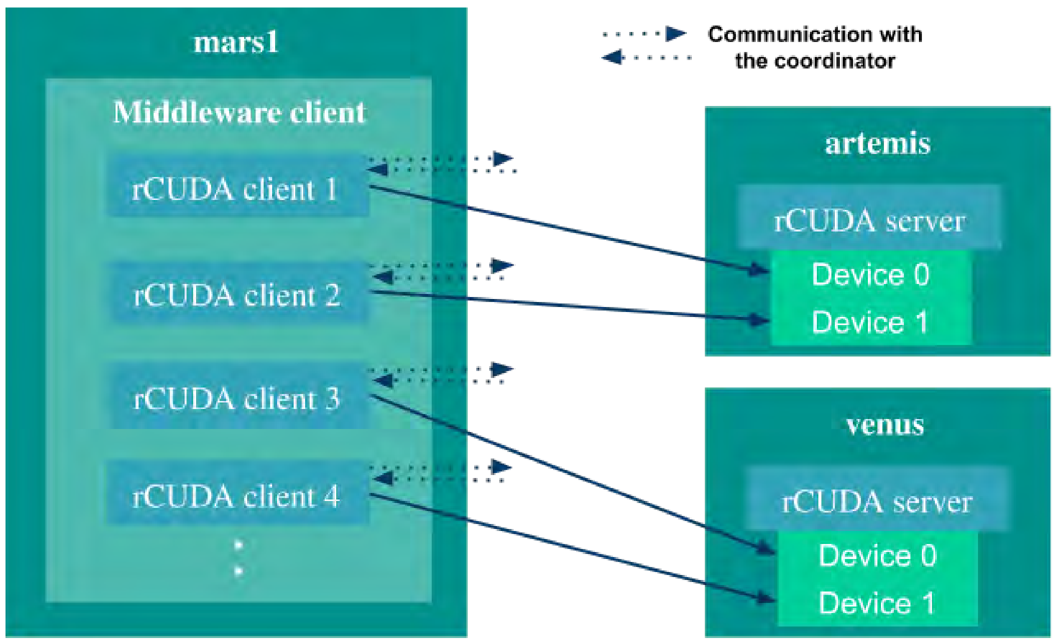


FIGURE 4.6: Experiment setup using the middleware

Chapter 5

Implementation of a CUDA Runtime API handling mechanism

5.1 Overview

The only approach to execute a CUDA program using the rCUDA framework, even when using the middleware introduced in the previous chapter, is by running the whole program remotely as a monolithic entity. The only approach to execute a CUDA program using the rCUDA framework, even when using the middleware introduced in the previous chapter, is by running the whole program remotely as a monolithic entity. Many CUDA programs are capable of running concurrently on multiple GPUs, which introduces inner interactions to take place between the executions of the individual calls on these GPUs. This perplexes the scheduling problem because one needs to take into account the way parts of the code executed on different GPUs affect each other. In order to gain insight into this interaction, we need to at least have information considering the memory footprint of kernels, the thread geometries, and the degree of data sharing among kernels. This information is only available if someone intercepts CUDA calls. For that reason, we introduce a mechanism to intercept the calls and collect this information so that it can be used to drive scheduling policies. In this chapter, we will cover:

- How to intercept and modify a simple function call of the standard C library.
- How to intercept and modify CUDA Runtime API calls.
- The design and implementation of a mechanism to delay, store and process lazily the actual execution of CUDA API calls.
- The design and implementation of a mechanism to execute a batch of CUDA Runtime API calls on a selected device.
- The extensions of these mechanisms to support multi-GPU programs that use CUDA streams and asynchronous calls.

- The extensions to support multithreaded programs that run CUDA from several POSIX threads.
- The possible limitations of the aforementioned mechanisms and extensions.

5.2 CUDA Runtime API calls interception

On the following section we provide a review on how to intercept API calls to a dynamic library. We begin by showing how to intercept standard C library functions by providing an example. Then, we present how we can expand this concept to intercept CUDA Runtime API calls.

5.2.1 Call interception in C/C++

As an example, we use a simple C program that opens a socket. The program creates a socket without binding it to an address, as displayed in Listing 5.1. We want to intercept the socket() function to modify the behavior of the call. We start with the shared library socket_hook.so that will override the socket() function, as presented in Listing 5.2.

LD_PRELOAD is an optional environment variable, containing one or more paths to shared libraries or shared objects that the loader will load before any other shared library, including the C runtime library (libc.so). This is called preloading a library. With the use of LD_PRELOAD, we can easily have our socket_hook.so library loaded before the standard C libraries. The first occurrence of the socket() function is in the shared library that we have already created. This enables library functions to be intercepted and replaced (overwritten.) As a result, program behavior can be modified in a non-invasive manner. Up to this point, we discussed how to override a standard socket() call with our function from our shared library. In the next step, we will also call the original socket() function from our shared library. This will enable us to easily add additional functionality to existing standard C libraries without modifying or rewriting the original libraries. Additionally, when implemented correctly, function call interception is completely transparent to user software that calls the original function.

To achieve that, we will use the dlsym function (void* dlsym(void* handle, const char* name)). The dlsym() function lets a process obtain the address of the symbol specified by name defined in a shared object. The dlsym is available only to dynamically linked processes. If the handle is equal to RTLD_NEXT, dlsym() searches the objects loaded after the object calling dlsym(). We use the dlsym function with RTLD_NEXT (from <dlfcn.h>) to find the next occurrence of the socket() function and store the location in o_socket. Hence, in our example, o_socket can from then on be used as a pointer to call the original socket() function. _GNU_SOURCE has to be specified to be able to use RTLD_NEXT [5].

We compile as:

```
$ gcc -Wall -fPIC -shared socket_hook.c -o socket_hook.so -ldl
```

and using our new shared library gives us the following:

```
LD_PRELOAD=./socket_hook.so ./simple_client
socket() call have been intercepted
Socket successfully created
```

```
#include <stdio.h>
#include <sys/socket.h>

int main(int argc, char *argv[]){
    int sockfd;
    // Create socket and check for error
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("Error : Could not create socket\n");
        return 1;
    }else{
        printf("Socket successfully created\n");
    }
    return 0;
}
```

LISTING 5.1: A simple client that opens a socket in C

```
#define _GNU_SOURCE
#include <stdio.h>
#include <sys/socket.h>
#include <dlfcn.h>

int (*o_socket)(int,int,int);  // return_type     int  , type_name
    o_socket, arguments     (int,int,int)

int socket(int domain, int type, int protocol){
    // find the next occurrence of the socket() function
    o_socket = dlsym(RTLD_NEXT, "socket");

    if(o_socket == NULL){
        printf("Could not find next socket() function occurrence");
        return -1;
    }
    printf("socket() call have been intercepted\n");

    // return the result of the call to the original C socket() function
    return o_socket(domain,type,protocol);
}
```

LISTING 5.2: A simple socket interception using a dynamic library

5.2.2 Call interception of CUDA Runtime API

The next step is to apply the aforementioned call interception technique to intercept the CUDA Runtime API calls. In the same way as above, we begin by creating a shared library "lib_cuda_intercept.so" that will override every supported function call of the API. Listing 5.3 shows the interception of cudaMalloc(void ** devPtr, size_t size). In the same way as previously, we extract a function pointer to the original function call. Here we use typedef to alias types such as "cudaMalloc_t" by using the following syntax:

```
typedef  void  (*cudaMalloc_t)  ( );
            ^             ^          ^
       return_type  ptr_name   arguments
```

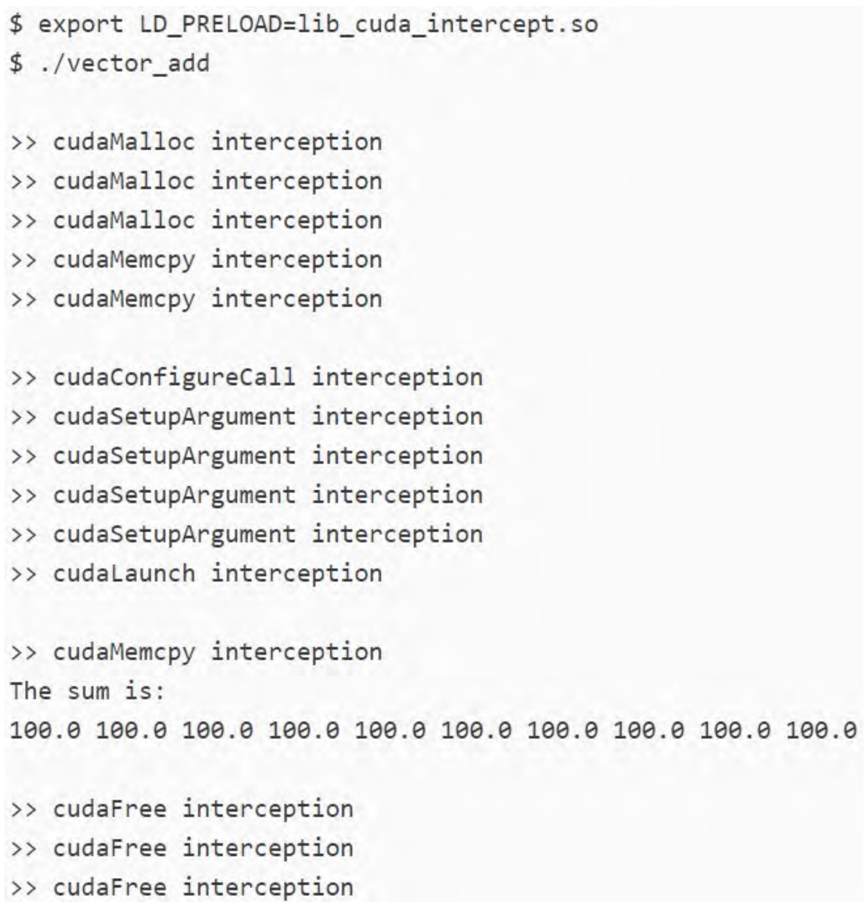
When the call is intercepted, we can modify its behavior. We can access and store the arguments of each call and control its execution. We can either execute the original function, trigger other functions to be executed, or even skip the execution of the original function alltogether.

```
typedef cudaError_t (*cudaMalloc_t)(void ** devPtr, size_t size);

static cudaMalloc_t native_cudaMalloc = NULL;

extern "C" cudaError_t cudaMalloc(void ** devPtr, size_t size) {

//Here we can modify its behavior

// return(cudaSuccess)  If we want nothing to happen

// Call the actual cudaMalloc() function
if (native_cudaMalloc == NULL) {
    native_cudaMalloc = (cudaMalloc_t)dlsym(RTLD_NEXT,"cudaMalloc");
}

assert(native_cudaMalloc != NULL);
return native_cudaMalloc(devPtr,size);

}
```

LISTING 5.3: cudaMalloc call interception code in C++

To test the functionality of our library, we used a simple "toy program" in CUDA that performs a vector addition of 10 element arrays. The CUDA Runtime API calls that are used by the program are three calls to cudaMalloc(), three calls to cudaMemcpy(), one kernel invocation, and three calls to cudaFree().

When we launch a kernel with the <<<>>> notation the compiler injects three different calls to the binary. Firstly, it runs the cudaConfigureCall(), which specifies the grid and block dimensions for the kernel call. The second call is cudaSetupArguments(arg, size, offset) and is executed N times, where N is the number of arguments passed to the kernel function. It pushes size bytes of the argument pointed to by arg at offset bytes from the start of the parameter passing area, which starts at offset 0.



```
$ export LD_PRELOAD=lib_cuda_intercept.so
$ ./vector_add

>> cudaMalloc interception
>> cudaMalloc interception
>> cudaMalloc interception
>> cudaMemcpy interception
>> cudaMemcpy interception

>> cudaConfigureCall interception
>> cudaSetupArgument interception
>> cudaSetupArgument interception
>> cudaSetupArgument interception
>> cudaSetupArgument interception
>> cudaLaunch interception

>> cudaMemcpy interception
The sum is:
100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0

>> cudaFree interception
>> cudaFree interception
>> cudaFree interception
```

FIGURE 5.1: Execution results after the interception of a simple vector addition CUDA program

The arguments are stored in the top of the execution stack. The third call is cudaLaunch(entry), which launches the function entry on the device. The parameter entry must be a character string naming a function that can execute on the device. The parameter specified by entry must be declared as a __global__ function. In Figure 5.1, we can see the execution and results of the interception of this program. Every time we intercept a call, we print " » call_name " to the screen, and we execute the call without any further modification.

5.3 Handling of CUDA Runtime API calls interception

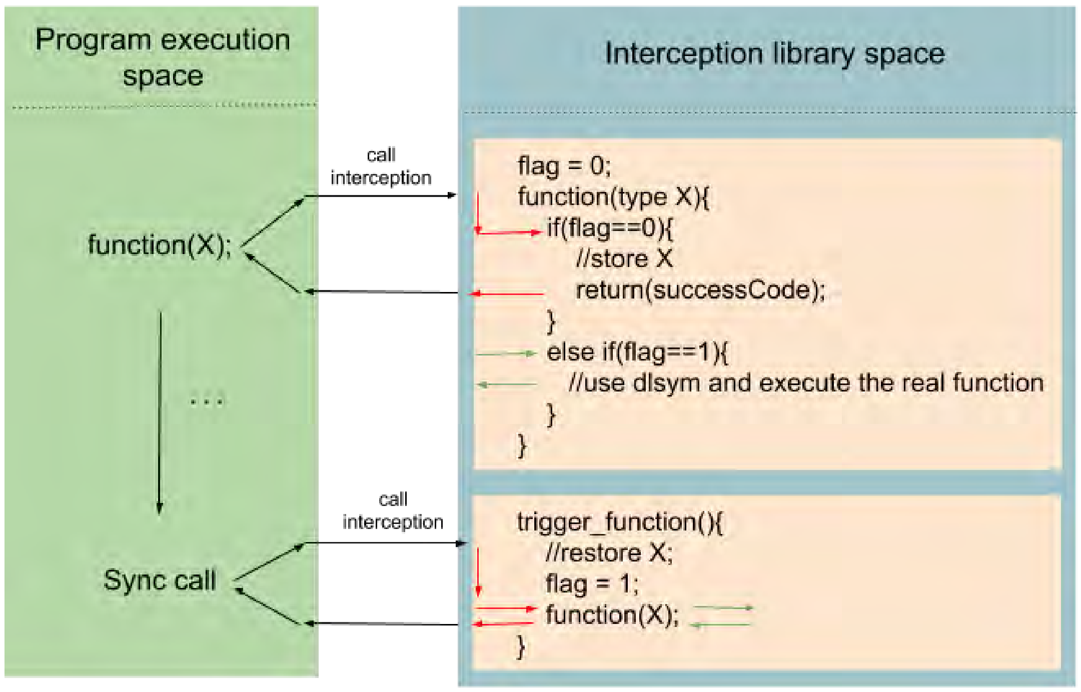
In this section we will discuss how we use the aforementioned function call interception mechanisms to intercept CUDA calls, analyze their arguments to extract useful information for scheduling policies, potentially rewrite the arguments, and lazily execute those CUDA function calls later. Figure 5.2 depicts the general lazy execution mechanism supported by our implementation. Whenever we intercept a CUDA call, we store information about the type of that call and the value of its arguments. The data structure used to store this information is discussed in Section 5.3.1. We return a success code to the user program, without executing the real function without any delay. A more sophisticated way to produce return values, especially for functions returning pointers, is discussed in Section 5.3.1. The program does not know when the actual execution of the call is completed and considers that it has been executed successfully, even if it its execution has actually been delayed.

We execute lazily postponed CUDA functions when we intercept special CUDA calls that have synchronization point semantics. To achieve execution, we have to restore the information about the actual functions and change a flag that indicates that the functions must be executed and not recorded and postponed.

Another feature that we use is the '__attribute__(constructor)' in C++. This constructor runs when a shared library is loaded during program startup. Its destructor runs when the shared library is unloaded, typically on program exit. As presented in Listing 5.4, we use this constructor to create a thread that begins to run when we first load the dynamic library. This thread can be used for accounting and scheduling purposes as it has a global view. The thread is terminated on program execution.

```
void *func(void *data){
    while(1){
        // Do something useful
    }
    pthread_exit(NULL);
}

__attribute__((constructor))  static void init_functon (void) {
    pthread_t handle;

    if (!pthread_create(&handle, NULL, func, NULL)){
        //Init thread create successfully
        if (!pthread_detach(handle)){
            //Init thread detached successfully
        }
    }
    return;
}
```

LISTING 5.4: Creating a thread from the constructor when the shared library is loaded



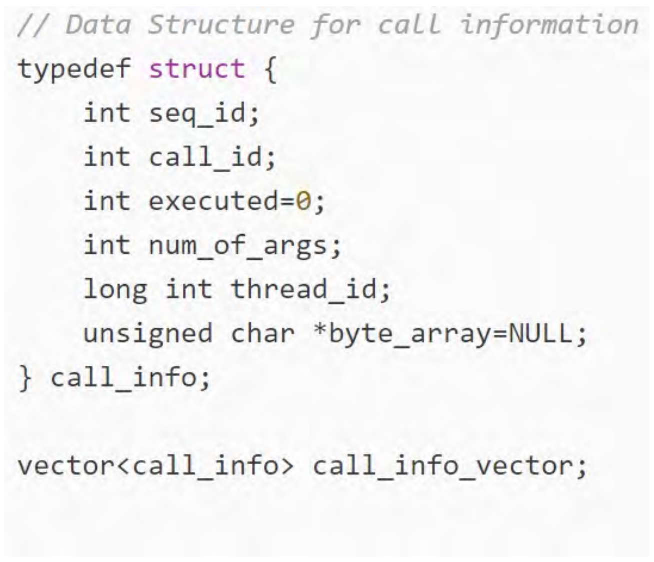
```
flag = 0;
function(type X){
  if(flag==0){
    //store X
    return(successCode);
  }
  else if(flag==1){
    //use dlsym and execute the real function
  }
}
```

```
trigger_function(){
  //restore X;
  flag = 1;
  function(X);
}
```

FIGURE 5.2: General mechanism of storing call information and delay real execution of the call

5.3.1 Data Structure used for storing CUDA call information

In order to analyze the information from CUDA API calls, we need a data structure where we store information for each of the calls that get intercepted. Figure 5.3 depicts the central data structure for the calls, which is a vector of structures in C++. Each element of the vector represents a CUDA call. Each member includes some type of information about the particular CUDA call.



```
// Data Structure for call information
typedef struct {
    int seq_id;
    int call_id;
    int executed=0;
    int num_of_args;
    long int thread_id;
    unsigned char *byte_array=NULL;
} call_info;

vector<call_info> call_info_vector;
```

FIGURE 5.3: Data structure to store CUDA calls information

More specifically, *seq_id* designates the sequence number of this call that is called from a particular thread (*thread_id*). We obtain the identifier of the thread of which we intercepted a call, with pthread_self() function. The member *call_id* is an identifier with distinct value for every type of call and is used to recognize which is the actual CUDA call name/type later. The member *executed* is an integer value, which shows if a particular call has been truly executed already or not. The *num_of_args* member is equal to the number of arguments of the particular call. The member *byte_array* is an array where we store the actual values of arguments as bytes (or characters). This type of structure allows us to store and acquire the arguments of the calls flexibly and quickly, regardless of their type.

Every time we intercept a CUDA call, we create a new instance of *call_info* structure, we store the appropriate information in the members of the structure based on the information about the actual call, and we push the element to the vector of calls. If we later need to access the information about the particular call, we can access it in constant time if we know its position in the vector.

5.3.2 Lazy, batch execution of delayed CUDA calls at synchronization points

The objective of collecting the arguments of all the CUDA calls is to acquire knowledge on the required resources prior to the actual execution of the calls. This *a priori* knowledge is a prerequisite for the design of sophisticated CUDA code to (local or remote) GPU mapping policies. To achieve that, we must delay the actual execution of the calls until a predefined synchronization point, in order to gather enough information before executing the calls. On the synchronization points, however, we are obliged to execute the previous calls, in order to maintain consistency with programmer view up to this point. When we eventually intercept a synchronization call, the scheduling policy in effect can analyze the information collected from previously intercepted calls to select a (local or remote) GPU device based on that information. After selecting a device, our framework injects a cudaSetDevice() function call to select the new device, and executes the set of calls that have been delayed up to the synchronization point. Figure 5.4 depicts the concept described above, where we store the information of the calls and we return a success code without executing the code, until we reach a synchronization point. The user is not aware of this process. The CUDA calls that carry synchronization semantics are the following:

- cudaLaunch()
- cudaDeviceSynchronize()
- cudaStreamSynchronize()

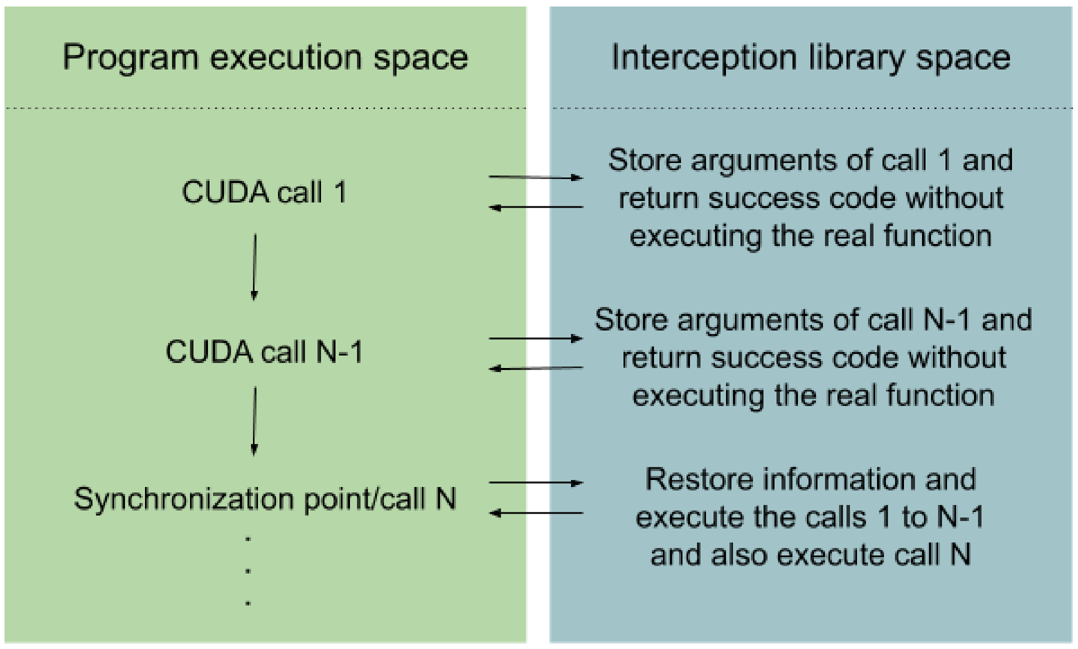

Program execution space
Interception library space
CUDA call 1
Store arguments of call 1 and return success code without executing the real function
CUDA call N-1
Store arguments of call N-1 and return success code without executing the real function
Synchronization point/call N
Restore information and execute the calls 1 to N-1 and also execute call N

FIGURE 5.4: Concept of synchronization points/calls

There is one main obstacle that we had to overcome to implement this idea. In CUDA, some function calls pass their arguments by reference as a pointer and expect a value to be stored as a value in this pointer. The problem is that when the user executes this kind of calls, it assumes that after the completion of the call, the content of the pointer is an actual value. That value may be used later on in other calls. A typical CUDA call with that behavior is the cudaMalloc (void** devPtr, size_t size) function. The user expects *devPtr to point to the newly allocated device memory area after the call. The value of *devPtr will be then used by functions such cudaMemcpy(void *Ptr, . . .). When we delay the actual execution, we have to return a value for this pointer, even though we have not actually executed the call because we have not met a synchronization point yet.

To solve this problem, we introduce another layer of virtualization. Our framework returns values to these pointers. The user has only knowledge of these virtual addresses over the actual virttual addresses of the GPU, and continue to use them in other function calls. This concept is depicted in Figure 5.5. When we first intercept a CUDA call such as cudaMalloc, we create a unique handle returned to devPtr as the "pointer" to the newly allocated area. That value actually indicates a position on a mapping array that stores 64-bit addresses, called virtual_addr. This array will be used later, when the memory allocation call is actually executed, to maintain the association between the handle and the actual virtual address. The programmer treats these handles as actual virtual addresses (the whole process is transparent to the programmer). When execution reaches a synchronization point and the cudaMalloc function is eventually executed, we acquire the actual virtual address on the device, and we match the handle with the actual address.

We use the handle as an index to the position of the virtual address in the mapping array.

The following CUDA calls that have handle value as a virtual address argument will be rewritten with the actual virtual address at the time of execution. The search for the handle to virtual address mapping is performed in constant time as we use the handle as an index in the mapping array. When we meet a function such as the cudaFree(void* devPtr) that frees memory on the device using a handle, we both free the memory area starting from the corresponding virtual address and we also erase the handle to virtual address mapping in the table.

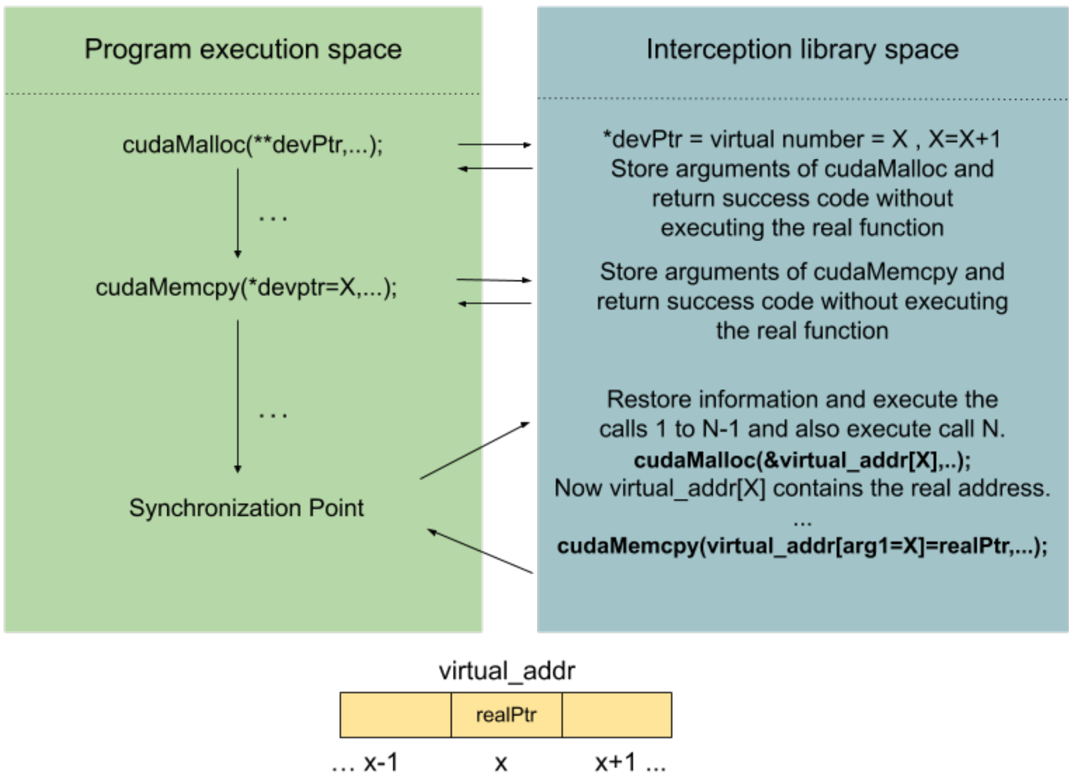

Program execution space
cudaMalloc(**devPtr,...);
. . .
cudaMemcpy(*devptr=X,...);
. . .
Synchronization Point
Interception library space
*devPtr = virtual number = X , X=X+1
Store arguments of cudaMalloc and return success code without executing the real function
Store arguments of cudaMemcpy and return success code without executing the real function
Restore information and execute the calls 1 to N-1 and also execute call N.
cudaMalloc(&virtual_addr[X],..);
Now virtual_addr[X] contains the real address.
...
cudaMemcpy(virtual_addr[arg1=X]=realPtr,...);
virtual_addr
realPtr
... x-1
x
x+1 ...

FIGURE 5.5: Use of virtual address handles

5.3.3 Support for multi-GPU CUDA applications with CUDA streams

We can apply the aforementioned mechanism to support the scheduling of the CUDA function calls on different devices when using multiple GPUs. One way to execute CUDA programs on multiple GPUs is by using specific CUDA API calls. The main calls that are used are the cudaSetDevice(int device) function, functions for streams such as the cudaStreamCreate() function and asynchronous functions such as the cudaMemcpyAsync() call. The user sends a set of functions to be executed on a specific device after using the cudaSetDevice() function. By intercepting the arguments of the calls prior to any synchronization point, we can make more educated decisions on the (remote or local) device to use for executing these calls. The main idea is depicted in Figure 5.6. After the interception of all the calls, we have gathered information about that batch of CUDA calls. When launching the kernel function, we decide on which device to execute these calls and overwrite the destination device value that has been given by the user.

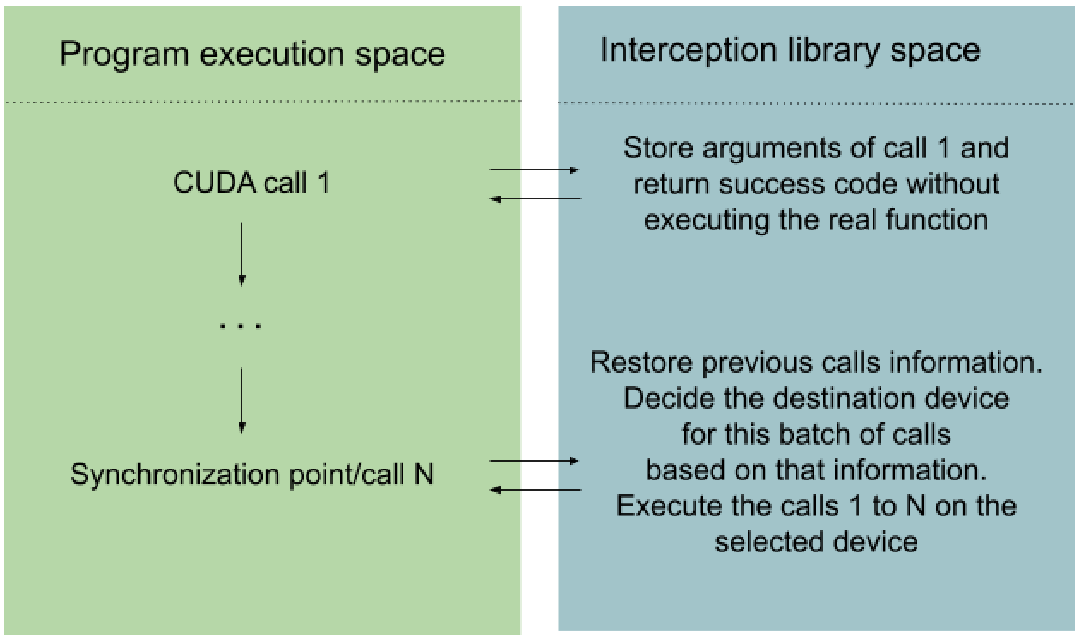

Program execution space
Interception library space
CUDA call 1
Store arguments of call 1 and return success code without executing the real function
. . .
Synchronization point/call N
Restore previous calls information. Decide the destination device for this batch of calls based on that information. Execute the calls 1 to N on the selected device

FIGURE 5.6: Concept of selecting a device to execute a set of delayed CUDA calls

There is an obstacle that we have to workaround to implement the aforementioned idea. First of all, the user designates a sequence of calls to be executed on a specific device by calling the cudaSetDevice(deviceId) function before any call in the sequence. Therefore, we cannot just redirect every CUDA call to every device we desire. That would change the semantics and result in unexpected behavior as some of the calls that must be executed on the same device would be split on different devices. The single parameter that we can control and rewrite is deviceId (in a manner similar to handles and virtual addresses discussed earlier). We can, thus, transparently substitute (rewrite) the deviceId with another device that best fits the execution of that set of calls, based on the information that we have collected and the scheduling policy in effect.

This idea is depicted in Figure 5.7. We use an array where we store the association between the selected device and the deviceId. We use the same technique as before, where the value deviceId indicates the position on the array in which we store the value of the selected device. Hence, the deviceId value is used as an indexing value.

The same process will be repeated if we meet another cudaSetDevice function with different deviceId value. If the user ever reuses the same deviceId in a cudaSetDevice function, then we match the deviceId with the previously selected device and execute it with that device. The association between the deviceId values, and the values of the selected device can not change until the end of the program execution, in order to avoid breaking the semantics (and thus compromizing the correctness) of the original CUDA program.

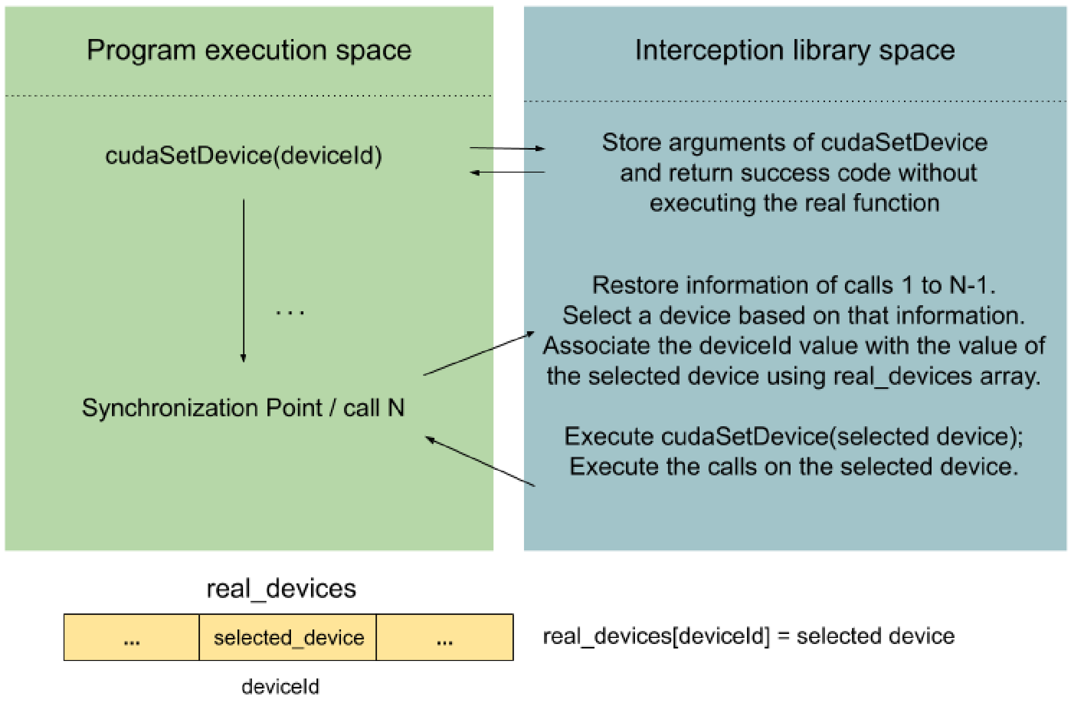

Program execution space
Interception library space
cudaSetDevice(deviceId)
Store arguments of cudaSetDevice and return success code without executing the real function
...
Restore information of calls 1 to N-1. Select a device based on that information. Associate the deviceId value with the value of the selected device using real_devices array.
Synchronization Point / call N
Execute cudaSetDevice(selected device); Execute the calls on the selected device.
real_devices
...
selected_device
...
real_devices[deviceId] = selected device
deviceId

FIGURE 5.7: Use of virtual device identifiers

We have to address a similar problem to the one discussed in Section 5.3.2, with the creation of a CUDA stream. When creating a stream with either cudaStream-Create(cudaStream_t * pstream) or cudaStreamCreate WithFlags (cudaStream_t * pstream, ...), we expect a new stream identifier to be stored in pstream after the completion of the call. In our case, as we do not execute the actual function right away, we return a stream identifier handle to that address. The programmer will be only aware of the handle, and she will use it as argument to other functions such as cudaMemcpyAsync(..., cudaStream_t stream). When we reach a synchronization point, we have to execute these calls. When we get the actual stream identifier, we map it with the stream handle. After this process, at execution time, when we detect a stream handle as an argument, we substitute it with the actual stream identifier. We remove the information about a particular identifier when we execute the cudaStreamDestroy() call.

5.3.4 Support for multi-GPU CUDA applications that use POSIX threads

The second most popular approach to execute CUDA programs on multiple GPUs is to use many different threads, as depicted in Figure 5.8. Each thread has, by default its own GPU context, and the CUDA calls issued by the thread are associated with the respective context. Therefore, it is trivial to support simultaneous, multi-GPU execution by simply calling cudaSetDevice by each thread to associate its GPU context with a specific GPU device. If a thread does not use the cudaSetDevice call, it will execute its commands on the default device.

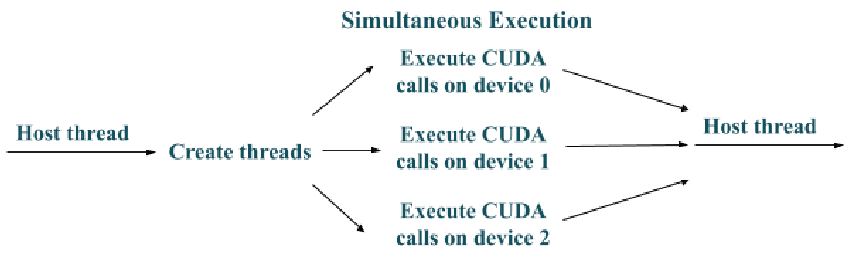

Simultaneous Execution
Host thread
Create threads
Execute CUDA calls on device 0
Execute CUDA calls on device 1
Execute CUDA calls on device 2
Host thread

FIGURE 5.8: Basic flow of a multi-GPU CUDA program execution using POSIX threads

In order to design our implementation correctly, we must have in mind the concept of CUDA Contexts. Every device is accessed through a construct called its context and is distinct for every device. It encapsulates all CUDA resources and actions such as the Streams, Memory objects, and Kernels. It occupies a separate address space. It is created by runtime on the first Runtime API call during initialization, and it is shared among all threads in the same CPU process. Each thread accesses a device through its context, and a single thread can swap amongst different contexts using the cudaSetDevice() call, as depicted in Figure 5.9. A multi-threaded application can hold multiple CUDA contexts simultaneously on the same GPU, but these contexts cannot perform operations concurrently. When active, each context has sole use of the GPU and must yield before another context access the GPU. We can only have one context on a GPU at a time. We design our implementation without affecting the semantics of CUDA contexts, and the programmer has the responsibility to follow the rules correctly to achieve the desired outcome [25].

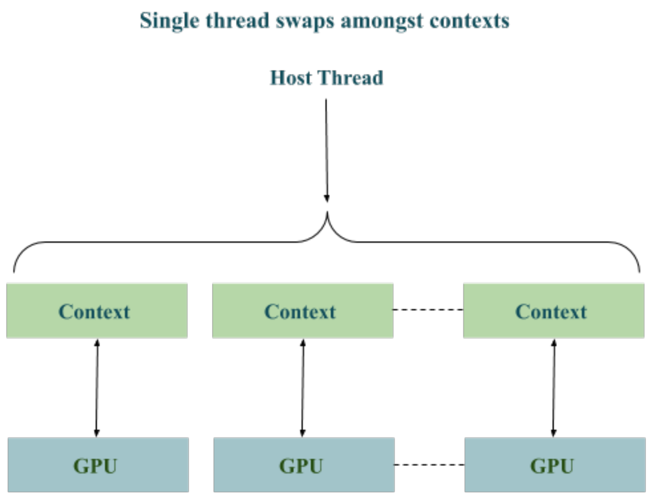
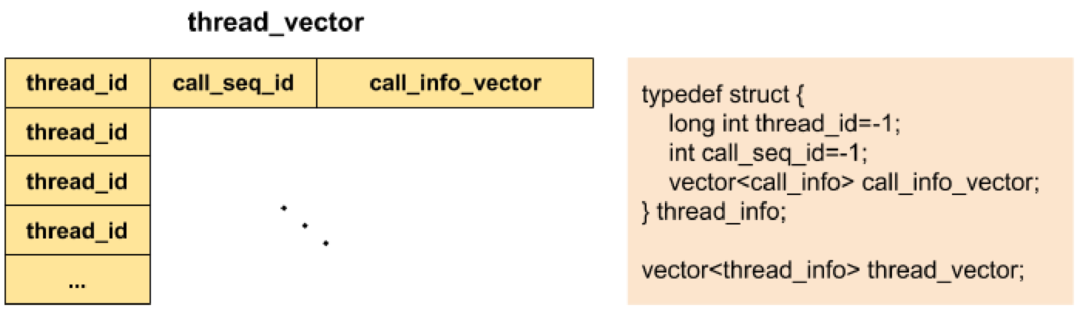


FIGURE 5.9: CUDA device contexts viewed by a thread

To support these types of multi-threaded programs, we need to extend the implementation of our interception mechanism and our data structures. First of all, we add another dimension (host thread id) to all data structures, so that each thread can work on an isolated subset of the data structures. In that way, we limit the use of locking and achieve better performance. As depicted in Figure 5.10, this data structure is a vector of structs in C++. Each position of this vector refers to a specific thread. The struct includes a member value which is equal to the thread identifier. It also includes a member of type call_info_vector, which is used to store information about the calls that have been intercepted from the particular thread. Finally, the structure also contains a member that maintains the sequence id of the calls for the specific thread.

Every time we intercept a new CUDA call, we identify the thread that issued the call. If it is the first call for this particular thread, then we create a new instance of the structure thread_info, and we push it to the thread_vector vector. We follow the same process as before, by applying the concept of batch execution of delayed CUDA calls at synchronization points, at the granularity of each thread.



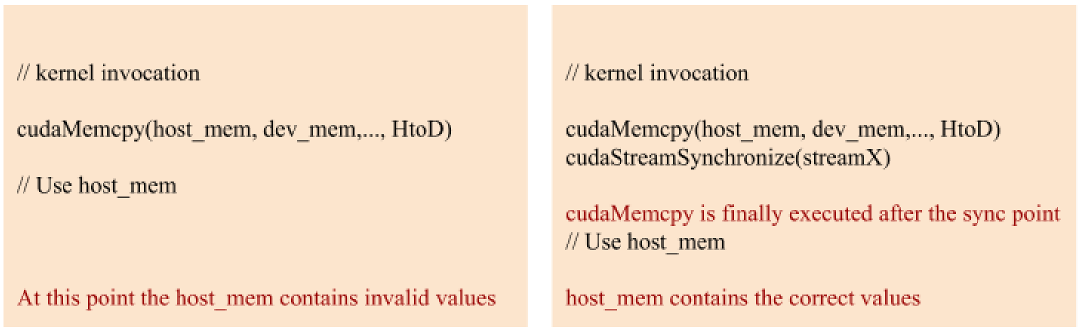
```
typedef struct {
    long int thread_id=-1;
    int call_seq_id=-1;
    vector<call_info> call_info_vector;
} thread_info;

vector<thread_info> thread_vector;
```

FIGURE 5.10: Data structure to support multiple threads

5.3.5 Limitations

One limitation that we have to consider when employing this library is that the programmer must explicitly indicate with a synchronization point/call, that the memory transfers between the device and the host, must be forced to be completed up to that point, in order for the host to use these data. Figure 5.11 depicts the problematic situation where, after a synchronous cudaMemcpy (DtoH), we expect that the memory transfer is completed after the call is performed. Thus, we proceed to use the host memory. The problem is that we have not yet truly executed the cudaMemcpy up to this point, as we haven't met any synchronization point after that the cudaMemcpy call. This would lead us to an unexpected error (even worse, potentially a silent data corruption), since the host memory does not (yet) contain the right data.

The solution to that problem is to explicitly indicate that the execution of the call that is responsible for the memory transfer to the host is being actually executed before the host use the host data. This can only be achieved by using a synchronization point. In our case, after the use of cudaMemcpy, we must use either a cudaDeviceSynchronize() call or a cudaStreamSynchronize() call based on the CUDA program implementation.



```
// kernel invocation

cudaMemcpy(host_mem, dev_mem,..., HtoD)

// Use host_mem


At this point the host_mem contains invalid values
```

```
// kernel invocation

cudaMemcpy(host_mem, dev_mem,..., HtoD)
cudaStreamSynchronize(streamX)

cudaMemcpy is finally executed after the sync point
// Use host_mem

host_mem contains the correct values
```

FIGURE 5.11: Explicit use of synchronization points for correctness

In our current implementation, we support the most common CUDA Runtime API functions. These calls are the following:

- cudaMalloc
- cudaMemcpy
- cudaMemcpyAsync
- cudaMemset
- cudaFree
- cudaConfigureCall
- cudaSetupArgument
- cudaLaunch
- cudaStreamCreate
- CudaStreamCreateWithFlags
- cudaStreamSynchronize
- cudaSetDevice
- cudaDeviceSynchronize
- cudaGetErrorName
- cudaGetErrorString
- cudaGetLastError
- cudaPeekAtLastError

Chapter 6

Conclusions and future work

6.1 Conclusions

In this Thesis we started by characterizing the rCUDA API remoting GPU virtualization framework to gain insight on its performance and functionality in multiple scenarios, when executing a multitude of CUDA calls on virtualized GPU resources. We created a middleware which monitors the utilization of all GPU resources in the system and alleviates programmers from having to statically define which GPU resources (local, remote, on which node) are going to be used at execution time. Based on the characterization, we also created mechanisms to collect and analyze performance-critical information about the CUDA calls before the actual remote execution of them. CUDA calls are executed lazily after the analysis, which identifies the appropriate mapping of kernels to devices. These mechanisms can be used as the basis to build adaptive policies for execution on clusters of virtualized GPUs.

Our characterization confirmed that rCUDA does not support the unified memory management API calls and graphics libraries yet. We experimentally confirmed that rCUDA supports multiple programs simultaneously on the same remote device. It also enables the concurrent use of many remote devices for scalability and better performance. We then evaluated the performance overhead of rCUDA due to virtualization, as well as the effect of network (for execution to remote nodes). Data-intensive applications suffer more overhead compared with the direct use of CUDA in the local node. A less expected result was that applications using CUDA streams are effected the most. Another unexpected result was that the bandwidth between the host and the GPU is significantly decreased when we use the rCUDA framework, even if we deploy both the rCUDA client and the rCUDA server locally, on the same node. Also, when we use pinned memory with rCUDA, the average device to host bandwidth is much lower than the average host to device bandwidth. On the other side, when we use pageable memory, the average host to device bandwidth is lower than the average device to host bandwidth, which is exactly the opposite. By using a multi-GPU-capable application we observed that rCUDA does not implement any scheduling policy by itself (beyond any scheduling performed by the CUDA driver). We also revealed an unexpected behavior (which we reported as a bug to rCUDA developers) concerning the use of asynchronous calls such as cudaMemcpyAsync() in CUDA streams under rCUDA.

We continued by discussing the design and functionality of a middleware that supports the rCUDA framework. The purpose of this middleware is to offer suggestions to rCUDA clients at runtime, concerning the device in which it should execute a CUDA program, based on continuously monitoring of server / device status and on the decision of a central authority.

In order to make educated scheduling decisions one should have insight into application characteristics, beyond just monitoring server / device status. We followed the approach of intercepting CUDA calls in order to gain this additional information. We intercept CUDA API Runtime function calls to acquire knowledge by observing the parameters of CUDA functions and making scheduling decisions before their actual invocation. To achieve that, we designed and implemented a mechanism which delays the calls' execution until a synchronization point. When we intercept a synchronization point/call, we analyze the collected data, and we can feed the results of the analysis to the scheduler to make intelligent scheduling decisions. Then we can lazily execute these calls on the selected device, after performing rewriting of CUDA call parameters wherever necessary. We also extended our mechanism to support multi-GPU-capable CUDA programs, which either use CUDA streams or Pthreads to achieve concurrency.

6.2 Future work

Future work will explore the following directions:

First, we plan characterize rCUDA on top of high-bandwidth, low-latency network interconnects, such as InfiniBand. Using such high-end interconnects will reduce network overhead, shifting the focus to the effects of the virtualization overhead of rCUDA and motivating the development of methods to reduce it.

In the middleware that supports rCUDA execution, we plan to implement and compare different scheduling / redsource management algorithms in the coordinator. These algorithms may make use of machine learning models, beyond conventional scheduling techniques.

The interception mechanism can be extended to support all the CUDA Runtime API calls. We also plan to design, implement and evaluate scheduling policies capable of exploiting the additional information collected by CUDA call interception. Finally, we will combine the middleware along with the interception mechanism to support more intelligent resource management policies and to enable end-to-end transparency for the programmer.

Bibliography

[1] *CPU vs GPU Comparison*. https://www.omnisci.com/technical-glossary/cpu-vs-gpu.

[2] *CUDA Demo Suite, Demos, bandwidthTest*. v11.0.3. NVIDIA. Aug. 2020. URL: https://docs.nvidia.com/cuda/demo-suite/index.html.

[3] *cudaSetDevice, Device Management documentation*. NVIDIA. 2020. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g159587909ffa0791bbe4b40187a4c6bb.

[4] *Device Memory,CUDA Runtime,CUDA TOOLKIT DOCUMENTATION*. NVIDIA. Aug. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory.

[5] *dlsym documentation*. IEEE Open Group. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/dlsym.html.

[6] Mark Harris. *An Easy Introduction to CUDA C and C++*. Oct. 2012. URL: https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/.

[7] Mark Harris. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. Jan. 2015. URL: https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[8] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. “GPU Virtualization and Scheduling Methods: A Comprehensive Survey”. In: *ACM Comput. Surv.* 50.3 (June 2017). ISSN: 0360-0300. DOI: 10.1145/3068281.

[9] T. McKercher J. Cheng M. Grossman. *Professional CUDA C Programming*. 1st ed. John Wiley Sons, 2014. ISBN: 1118739329.

[10] *KVM Linux*. URL: https://www.linux-kvm.org/page/Main_Page.

[11] Justin Luitjens. *CUDA STREAMS, BEST PRACTICES AND COMMON PITFALLS*. Tech. rep. NVIDIA. URL: https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[12] Raffaele Montella et al. “On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework”. In: *Int. J. Parallel Program.* 45.5 (Oct. 2017), 1142–1163. ISSN: 0885-7458. DOI: 10.1007/s10766-016-0462-1.

[13] *NVIDIA GRID K2 GRAPHICS BOARD*. URL: https://www.nvidia.com/content/grid/pdf/grid_k2_bd-06580-001_v02.pdf.

[14] *NVIDIA System Management Interface*. NVIDIA. URL: https://developer.nvidia.com/nvidia-system-management-interface.

[15] M. Oikawa et al. "DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, pp. 1207–1214.

[16] 3D Game Engine Programming. *CUDA Memory Model*. Nov. 2011.

[17] *rCUDA v18.10beta User's Guide documentation*. v18.10beta. rCUDA. July 2020.

[18] C. Reaño and F. Silla. "A Performance Comparison of CUDA Remote GPU Virtualization Frameworks". In: *2015 IEEE International Conference on Cluster Computing*. 2015, pp. 488–489.

[19] Carlos Reaño et al. "Improving the user experience of the rCUDA remote GPU virtualization framework". In: *Concurrency and Computation: Practice and Experience* 27.14 (2015), pp. 3746–3770. DOI: 10.1002/cpe.3409.

[20] *Thread Hierarchy, Programming Model, CUDA TOOLKIT DOCUMENTATION*. NVIDIA. Aug. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[21] Cornell University. *Introduction to GPGPU and CUDA Programming: Stream and Synchronization*. Tech. rep. Cornell University Center for Advanced Computing.

[22] *VMware*. URL: https://www.vmware.com/.

[23] Wikipedia. *CUDA*. URL: https://en.wikipedia.org/wiki/CUDA.

[24] *Xen project, THE LINUX FOUNDATION PROJECTS*. URL: https://xenproject.org/.

[25] S. Yalamanchili and J. Wang. *CUDA kernel Execution*. Tech. rep. Georgia Institute of Technology. URL: https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/0/550/files/2018/02/Micro-I-KernelExecution.pdf.

Chapter 7

Appendices

Appendix A: Eligibility of rCUDA for programs with distinct behaviors

NVIDIA CUDA SDK Sample Programs	Availability or Possible Problem
0_Simple	
matrixMul	Ok
cdpSimpleQuicksort	Ok
simpleStreams	Ok
vectorAdd	Ok
simpleVoteIntrinsics	Ok
matrixMulCUBLAS	Ok
simpleMultiCopy	Ok
UnifiedMemoryStreams	cudaMallocManaged not supported
simpleOccupancy	Ok
simpleTemplates	Ok
simpleP2P	Ok
simplePitchLinearTexture	Ok
1_Utilities	
bandwidthTest	Ok
deviceQuery	Ok
deviceQueryDrv	Ok
p2pBandwidthLatencyTest	Ok
topologyQuery	Ok
2_Graphics	
	openGL is not supported
3_Imaging	
convolutionFFT2D	Ok
convolutionSeparable	Ok
convolutionTexture	Ok

NVIDIA CUDA SDK Sample Programs	Availability or Possible Problem
dct8x8	Ok
dwtHaar1D	Ok
dxtc	Ok
histogram	Ok
HSOpticalFlow	Ok
4_Finance	
binomialOptions	Ok
BlackScholes	Ok
quasirandomGenerator	Ok
5_Simulations	
	openGL is not supported
6_Advanced	
alignedTypes	Ok
c++11_cuda	Ok
cdpAdvancedQuicksort	Ok
cdpBezierTessellation	Ok
cdpLUDecomposition	Ok
concurrentKernels	Ok
eigenvalues	Ok
fastWalshTransform	Ok
FDTD3d	Ok
interval	Ok
mergeSort	Ok
newdelete	Ok
reduction	Ok
scalarProd	Ok
scan	Ok
shfl_scan	Ok
simpleHyperQ	Ok
sortingNetworks	Ok
StreamPriorities	Ok
threadFenceReduction	Ok
transpose	Ok
warpAggregatedAtomicsCG	Ok

Appendix B: Characterization of the overhead of virtualization of rCUDA framework for the CUDA SDK sample programs

CUDA SDK Samples	Average execution time (seconds) executing locally with rCUDA	Average execution time (seconds) executing remotely with rCUDA	Percentage increase(%) of average execution time by using rCUDA over natively CUDA
matrixMul	0.330	0.365	10.606
transpose	0.400	0.479	19.750
simpleStreams	2.787	57.856	1975.923
vectorAdd	12.198	12.208	0.081
simpleVoteIntrinsics	0.203	0.208	2.463
matrixMulCUBLAS	0.842	0.887	5.344
simpleMultiCopy	0.502	2.514	400.796
simpleOccupancy	0.233	0.278	19.313
simpleTemplates	0.206	0.253	22.815
simplePitchLinearTexture	0.383	0.424	10.704
deviceQuery	0.097	0.152	56.701
deviceQueryDrv	0.092	0.160	73.913
convolutionFFT2D	4.287	4.557	6.298
convolutionSeparable	2.083	2.173	4.320
convolutionTexture	1.186	1.269	6.998
dct8x8	0.665	0.692	4.060
dwtHaar1D	0.221	0.268	21.266
dxtc	0.252	0.270	7.142
histogram	1.912	2.698	41.108
HSOpticalFlow	14.251	14.487	1.656
binomialOptions	31.427	31.503	0.241
BlackScholes	1.334	1.441	8.020
quasirandomGenerator	1.063	1.130	6.302
alignedTypes	2.672	3.248	21.556
c++11_cuda	0.515	0.516	0.194
cdpAdvancedQuicksort	0.259	0.273	5.405
cdpBezierTessellation	0.236	0.286	21.186
cdpLUDecomposition	0.856	1.028	20.093
concurrentKernels	0.222	0.253	13.963
eigenvalues	2.416	2.408	0.041
fastWalshTransform	3.849	4.037	4.884
FDTD3d	14.527	14.656	0.888
interval	1.835	1.944	5.940
mergeSort	0.509	0.541	6.286
newdelete	0.217	0.219	0.921
reduction	0.805	0.885	9.937
scalarProd	0.277	0.283	2.166
shfl_scan	0.232	0.266	14.655
simpleHyperQ	0.277	0.432	55.956
sortingNetworks	7.286	7.536	3.431
StreamPriorities	1.488	2.142	43.951
threadFenceReduction	0.259	0.261	0.772
warpAggregated AtomicsGC	0.442	0.575	30.090

Appendix C: Characterization of the network overhead of using the rCUDA framework over TCP/IP for the CUDA SDK sample programs

CUDA SDK Samples	Average execution time (seconds) executing directly with CUDA	Average execution time (seconds) executing directly with rCUDA	Percentage increase (%) of average execution time by using rCUDA over natively CUDA
matrixMul	0.365	0.367	0.547
transpose	0.479	1.107	131.106
simpleStreams	57.856	428.251	640.201
vectorAdd	12.208	12.236	0.229
simpleVoteIntrinsics	0.208	0.241	15.865
matrixMulCUBLAS	0.887	0.896	1.014
simpleMultiCopy	2.514	8.043	219.928
simpleOccupancy	0.278	0.317	14.028
simpleTemplates	0.253	0.267	5.533
simplePitchLinearTexture	0.424	0.891	110.141
deviceQuery	0.152	0.165	8.552
deviceQueryDrv	0.160	0.163	1.875
convolutionFFT2D	4.557	4.973	9.128
convolutionSeparable	2.173	2.540	16.889
convolutionTexture	1.269	1.387	9.298
dct8x8	0.692	0.720	4.046
dwtHaar1D	0.268	0.272	1.492
dxtc	0.270	0.280	3.703
histogram	2.698	2.990	10.822
HSOpticalFlow	14.487	14.644	1.083
binomialOptions	31.503	31.549	0.146
BlackScholes	1.441	1.912	32.685
quasirandomGenerator	1.130	1.141	0.973
alignedTypes	3.248	7.931	144.18
c++11_cuda	0.516	0.519	0.581
cdpAdvancedQuicksort	0.273	0.305	11.721
cdpBezierTessellation	0.286	0.292	2.097
cdpLUDecomposition	1.028	1.690	64.396
concurrentKernels	0.253	0.263	3.952
eigenvalues	2.408	2.634	9.385
fastWalshTransform	4.037	4.121	2.080
FDTD3d	14.656	18.954	29.325
interval	1.944	2.826	45.370
mergeSort	0.541	1.005	85.767
newdelete	0.219	0.268	22.374
reduction	0.885	1.457	64.632
scalarProd	0.283	0.286	1.060
shfl_scan	0.266	0.349	31.203
simpleHyperQ	0.432	0.521	20.601
sortingNetworks	7.536	7.787	3.330
StreamPriorities	2.142	10.337	382.586
threadFenceReduction	0.261	0.359	37.547
warpAggregatedAtomicsCG	0.575	1.112	93.391

Appendix D: Part of the Multi-GPU matrix-matrix multiplication implementation of the host code in CUDA

```
int N = 1000;    //default number of elements (A dims:1000x1000, B dims: 1000x1000)
double r = 0.5; //default r
double inv_r = (1-r);            //default r_inv
cudaGetDeviceCount(&ndev);       //finds the number of available devices in our
    system

/// To compute  A 1 B1
id=0;
cudaSetDevice((int)(id%ndev));
cudaStreamCreateWithFlags(&streams[id],cudaStreamNonBlocking);

cudaMalloc((void**)&dA1,(int)(N*N*r*r*sizeof(double)));
cudaMalloc((void**)&dB1,(int)(N*N*r*r*sizeof(double)));
cudaMalloc((void**)&dC11,(int)(N*N*r*r*sizeof(double)));

cudaMemcpyAsync(dA1,hA1,(int)(N*N*r*r*sizeof(double)),cudaMemcpyHostToDevice,streams
    [id]);
cudaMemcpyAsync(dB1,hB1,(int)(N*N*r*r*sizeof(double)),cudaMemcpyHostToDevice,streams
    [id]);

m = (int)(N*r); n = (int)(N*r); // determines size of output matrix

matrixMultiply<<< dimGrid,dimBlock,0,streams[id]>>> (dA1,dB1,dC11,m,(int)(int)(r*N)
    ,(int)(int)(r*N),n,(int)(r*N));

cudaMemcpyAsync(hC11,dC11,(int)(N*N*r*r*sizeof(double)),cudaMemcpyDeviceToHost,
    streams[id]);

// To compute  A 2 B2
id=1;
cudaSetDevice((int)(id%ndev));
cudaStreamCreateWithFlags(&streams[id],cudaStreamNonBlocking);

cudaMalloc((void**)&dA2,(int)(N*N*r*inv_r*sizeof(double))); cudaMalloc((void**)&dB3
    ,(int)(N*N*r*inv_r*sizeof(double)));
cudaMalloc((void**)&dC12,(int)(N*N*r*r*sizeof(double)));

cudaMemcpyAsync(dA2,hA2,(int)(N*N*r*inv_r*sizeof(double)),cudaMemcpyHostToDevice,
    streams[id]);
cudaMemcpyAsync(dB3,hB3,(int)(N*N*r*inv_r*sizeof(double)),cudaMemcpyHostToDevice,
    streams[id]);

matrixMultiply<<< dimGrid,dimBlock,0,streams[id]>>>(dA2,dB3,dC12,m,(int)(N*inv_r),(
    int)(N*inv_r),n,(int)(r*N));

cudaMemcpyAsync(hC12,dC12,(int)(N*N*r*r*sizeof(double)),cudaMemcpyDeviceToHost,
    streams[id]);
 ...
id=0;
cudaSetDevice((int)(id%ndev));
cudaStreamSynchronize(streams[id]);
id=1;
cudaSetDevice((int)(id%ndev));
cudaStreamSynchronize(streams[id]);
...
```