UNIVERSITÀ DEGLI STUDI DI NAPOLI "PARTHENOPE"
SCUOLA INTERDIPARTIMENTALE DELLE SCIENZE, DELL'INGEGNERIA E DELLA SALUTE
DIPARTIMENTO DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA IN INFORMATICA


TESI DI LAUREA TRIENNALE

# GPGPU virtualization: design, development, and evaluation of HPC, IoT, and edge computing middleware

RELATORE:
Prof. Raffaele Montella
CORRELATORE:
Prof. Sokol Kosta

CANDIDATO:
Mariano Aponte
Mat. 0124002088

Anno Accademico 2022-2023

*To my loving grandma and family.*

# Abstract

Hardware virtualization is vital in High-Performance Computing and Cloud Computing scenarios. It allows an easy and cost-effective way to address the challenges of resource allocation, scalability, and performance isolation in shared environments. Since the rise of GPGPU-accelerated HPC Clusters, the research in the hardware virtualization field has focused more and more on GPGPU virtualization, and different solutions have sprung to life. In this work, we will explore the details of the Generalized Virtualization Service "GVirtuS," a transparent virtualization solution with the advantage of a framework designed in a plug-in style that allows easy development and choice of communicators and stub libraries. Firstly, we will briefly discuss GVirtuS's implementation and design choices. Furthermore, we will analyze the differences with similar solutions found by other researchers. Moreover, we will show how GVirtuS's performance was improved through the development and optimization of a novel RDMA-based communicator. We will then evaluate the performances of the communicator through the CUDA implementation of two foundational operations that underpin AI calculations: SAXPY and Matrix Multiplication. As we will see, our novel RDMA Communicator was able to achieve a 35% performance boost over the preexisting TCP Communicator run over an Infiniband fabric and a 55% performance boost over the same TCP Communicator run over a traditional Ethernet network. Finally, we will conclude and explore some future development goals.

La virtualizzazione dell'hardware è vitale negli scenari di High Performance Computing e Cloud Computing. Consente di affrontare in modo semplice ed economico le sfide legate all'allocazione delle risorse, alla scalabilità e all'isolamento delle prestazioni negli ambienti condivisi. Dall'avvento dei cluster HPC accelerati da GPGPU, la ricerca nel campo della virtualizzazione dell'hardware si è concentrata sempre di più sulla virtualizzazione GPGPU e sono nate diverse soluzioni. In questo lavoro esploreremo i dettagli del servizio di virtualizzazione generalizzato "GVirtuS", una soluzione di virtualizzazione trasparente con il vantaggio di un framework progettato in stile plug-in che consente un facile sviluppo e scelta di comunicatori e librerie stub. Innanzitutto discuteremo brevemente dell'implementazione e delle scelte progettuali di GVirtuS. Inoltre, analizzeremo le differenze con soluzioni simili trovate da altri ricercatori. Inoltre, mostreremo come le prestazioni di GVirtuS sono state migliorate attraverso lo sviluppo e l'ottimizzazione di un nuovo comunicatore basato su RDMA. Valuteremo quindi le prestazioni del comunicatore attraverso l'implementazione CUDA di due operazioni fondamentali che sono alla base dei calcoli dell'intelligenza artificiale: SAXPY e moltiplicazione di matrici. Come vedremo, il nostro nuovo comunicatore RDMA è stato in grado di ottenere un aumento delle prestazioni del 35% rispetto al comunicatore TCP preesistente eseguito su una struttura Infiniband e un aumento delle prestazioni del 55% rispetto allo stesso comunicatore TCP eseguito su una rete Ethernet tradizionale. Infine, concluderemo ed esploreremo alcuni obiettivi di sviluppo futuri.

# Contents

# List of Figures

# Listings

# 1 Introduction

## 1.1 GPGPUs in Scientific Computing and HPC

In modern times, Scientific Computing has become increasingly compute-intensive: from physics and mathematics to finance and humanities, and even in health, the need for high-performance infrastructures to solve complex problems has significantly risen. General Purpose Graphic Processing Units (GPGPUs) are widely used in this scenario as accelerators, thanks to their highly parallel architecture and high memory bandwidth. Resource-intensive computations can, therefore, be parallelized and executed on the GPGPUs, gaining massive performance boosts in different scenarios [1]. GPGPUs' exceptional capabilities make them useful in High-Performance Computing Clusters (HPC Clusters) and Cloud Computing.

## 1.2 Cloud Computing

Cloud Computing enables the user to access a shared pool of computing resources and software in a ubiquitous and on-demand fashion through a network, usually on a pay-per-use or charge-per-use basis [2]. Through hardware virtualization and software-as-a-service (SaaS) solutions, cloud computing represents an easy and usually cost-effective way to provide a computationally capable infrastructure to users. It is easy to realize how this approach can be convenient for scientific computing financially and from an ease-of-use point of view.

## 1.3 Hardware Virtualization, CUDA and gVirtuS

GPGPUs for HPC Clusters are mainly provided by NVIDIA Corporation. The company also provides a proprietary toolkit for GPGPUs programming called CUDA [3]. As mentioned in Section 1.2, multiple users can access the shared resources pool through virtualization. However, this approach does not allow a transparent use of CUDA-based GPUs and causes poor message-passing performances between virtual machine instances. An attempt to address these

limitations was made by gVirtuS (with lowercase g), which was a GPGPU Virtualization Service that allowed the execution of CUDA application in a private computing cloud for High-Performance Scientific Computing [4]. The middleware enabled VMs to see and exploit GPUs in HPC nodes without any need even to have a CUDA-capable device on the actual machine and even without the need to develop the CUDA applications with gVirtuS in mind (more on this in Section 3).

## 1.4 GVirtuS: General Virtualization Service

The capabilities of gVirtuS were extended and generalized by its successor, GVirtuS (with upper case G). As mentioned in Section 1.3, the main reason behind gVirtuS development was to allow a transparent use of CUDA-based GPUs in virtualization environments. GVirtuS extends gVirtuS by making it technology-agnostic: a split-driver approach allows now to develop interfaces in an effortless "plug-in style," not only for acceleration libraries (such as CUDA, OpenGL, and OpenCL) but also for file systems or communication libraries [5]. GVirtuS uses a modular approach to perform data transmission: Communicators can be developed in a plug-in fashion, similar to the split-driver development we just discussed. As for the basic communication needs, GVirtuS implements a TCP/IP Communicator: data is transmitted in a classic client-server fashion using sockets and read/write system calls.

## 1.5 Why TCP/IP is slow

Even if TCP/IP is a reliable means of transmission, its performance in GPGPU remoting is not astonishing. Transmission speed is severely impacted due to many context switches and data copies over the TCP/IP layers to perform even the most miniature transmission. The payload is copied through each layer multiple times on both peers for encapsulating and de-encapsulating the data. For apparent reasons, all these copies are costly. Furthermore, when the packet arrives at the Network Interface Controller (NIC), an interrupt is raised

and must be dealt with by the Interrupt Service Routine (ISR), causing another context switch and, therefore, more overhead. Even a non-technician could quickly grasp how significant the performance degradation of this approach is.

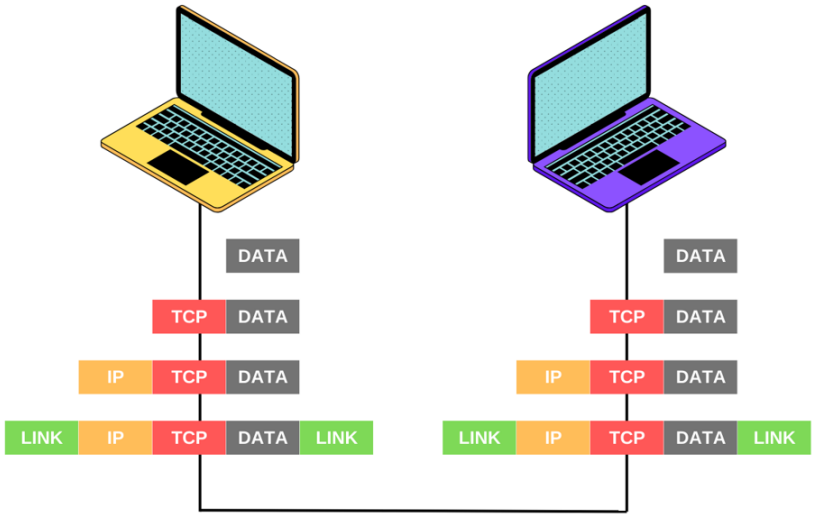


Figure 1: TCP/IP headers

Moreover, that is where Remote Direct Memory Access (RDMA) comes in handy. RDMA is a technology that allows a process to perform data transmission without any system call. Therefore, the context switch count performed by RDMA data transmission could be more manageable.

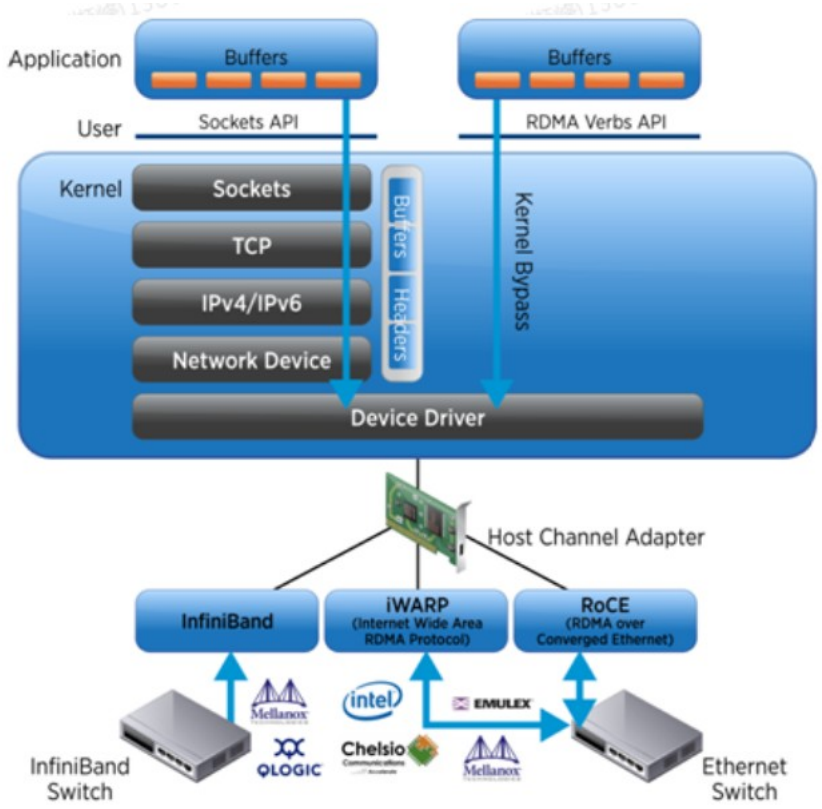


Figure 2: TCP/IP architecture (left) and RDMA architecture (right)

In addition, RDMA does couple greatly with Infiniband, which can offer data transfer rates up to 400Gb/s and ultra-low latency of up to 600ns end-to-end [6]. For this reason, we tried to develop an RDMA over Infiniband Communicator for GVirtuS [7].

## 1.6 This work

In this work, we showcase our attempt at developing an RDMA over Infiniband Communicator for GVirtuS. We implement the Communicator, considering the modular approach of the whole GVirtuS project, and test it with a Matrix Multiplication Algorithm and a SAXPY Algorithm. The rest of the thesis is organized as follows: in Section 2, we describe both related work to

GVirtuS and other RDMA Communicator implementation attempts; in Section 3, we dig deeper into GVirtuS architecture; in Section 4 and Section 5 we present how the RDMA over Infiniband Communicator was designed and implemented; in Section 6 we analyze the benchmarks of the Communicator and in Section 7 we draw some conclusions, with a brief discussion about what gone wrong and what not, and the future developments.

# 2 Related Work

## 2.1 GPGPU Remoting Works

The efficiency of GPGPUs for scientific computing applications and the rise of cloud computing lead to several researches in the GPGPU Remoting field. In particular two research projects relied on similar GPU virtualization solution to GVirtuS: rCUDA [8][9] and Distributed-Shared CUDA (DS-CUDA) [10]. These works use a similar approach to GVirtuS: they provide CUDA API wrappers to the frontend, in order to intercept calls to CUDA functions and offload the execution on a remote GPGPU. Even if the goal is the same, these three works differ in some aspects:

- GVirtuS does provide a transparent virtualization solution. Recompiling the source code of the CUDA Application is not needed if the latter was compiled with CUDA shared libraries. In contrast, DS-CUDA needs recompilation.
- GVirtuS provides a plug-in architecture. Component such as STUB libraries and communicators are easy to implement. GVirtuS is also general purpose, and both CUDA and OpenCL modules can be developed and leveraged, and different GPUs can be used. In contrast, DS-CUDA and rCUDA only work within the nVidia ecosystem.
- GVirtuS provides an hypervisor-independent and technology-agnostic communicator interface. Communicators for different scenarios can be easily developed and plugged in. In contrast, rCUDA and DS-CUDA only provide Infiniband and TCP sockets communication.
- GVirtuS and DS-CUDA are open-source, while rCUDA is closed source.

Other noteworthy projects are vCUDA [11] and GViM [12]. vCUDA implements a client-server model consisting of three user space modules. The first one is the "vCUDA library", that resides in the guestOS and act as a substitute for the original libcudart.so. It works like the GVirtuS Cuda STUB library,

intercepting and redirecting CUDA API calls. The second module is the so called "vCUDA STUB library". This module resides in the hostOS, and its purpose is to receive and execute the remote requests made by the guestOS. The third module is the "vGPU". It is used by the vCUDA library and it provides some abstraction and syncronization functionalities. GViM is a Xen-based solution for virtualizing GPGPUs. It offers an efficient accelerator virtualization, but its deployment requires some modification to the guest VMs running on the virtualized HPC platform. It is another example of a split-driver approach to virtualization, since the offloading is performed through frontend and backend CUDA wrapper libraries. It's clear that the split-driver model and an Infiniband-based connection are not newcomers into the GPGPUs virtualization landscape. While GVirtuS leverages the split-driver model, it is indeed not "RDMA over Infiniband ready" yet. And that's the purpose of this work.

## 2.2 RDMA and ROcE works

As previously mentioned, our work aims to leverage RDMA and Infiniband technologies to improve GVirtuS message passing performances. A student from Virginia Tech already made an attempt to implement an RDMA Communicator in GVirtuS, leveraging RDMA over Converged Ethernet (ROcE) technology to perform the data transmission between the frontend and backend modules of GVirtuS. Their RDMA Communicator was able to decrease execution times of a matrix multiplication algorithm by 12% to even 50% [13]. There are some differences with our work:

- **Libraries and abstractions**: Our work is implemented using librdmacm and libibverbs, while Virginia Tech's work is implemented using a project called libibverbscpp [14], which is a C++ binding of libibverbs. Both implementation actually refer to the Infiniband/RDMA verbs, but it was a noteworthy difference anyway;
- **Optimizations**: Both our work and Virginia Tech's work rely on memory region preregistration, since it's an hidden cost of RDMA that could

(and actually did, more on that in Section 6) remove any performance advantage of using RDMA and Infiniband. We also rely on polling-based work completion instead of event-based work completion. In addition Virginia Tech's communicator also relies on cache-aligning the buffers, RDMA-receive pre-posting (or so called mailbox approach), pipelined and batched network transfers and interception of cuda memory management function calls. We didn't implement these kind of optimization in our code since the scope of our work is to develop a straightforward way to transmit data between the frontend and the backend. In any case, all of the implementation choices are discussed in Section 5;

- **Testing**: While both works share the idea of using RDMA to perform data transmission, our work is aimed to exploit the whole Infiniband infrastructure. We tested our communicator on an Infiniband network, while Virginia Tech's communicator was tested only on Ethernet networks. The results of our tests are available in Section 6.

# 3 GVirtuS Architecture

*The GPU Virtualization Service (GVirtuS) [...] allows an instanced virtual machine to access GPGPUs in a transparent and hypervisor independent way, with an overhead slightly greater than a real machine/GPGPU setup* [4]. In this Section we dig deeper into the implementation details of GVirtuS, from the CUDA Stub Library to the Frontend, Backend and Communicators modules split-driver approach.

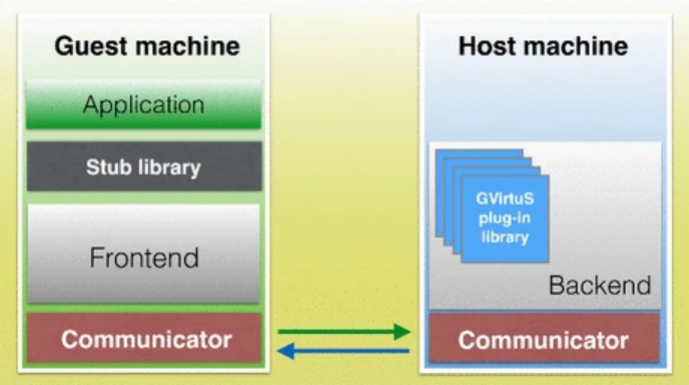


Figure 3: Split-driver architecture of GVirtuS

## 3.1 GVirtuS CUDA Frontend STUB Library

GVirtuS allows to access a remote GPGPU without the need to develop the CUDA application with the virtualization in mind. This is achieved by exploiting shared libraries and imitating the original CUDA library to intercept each function call. A CUDA Stub Library is developed with the same function headers of the original CUDA library, but the STUB functions actually sends the data to a remote machine that performs the requested functions and sends back the processed data. With this approach the virtualization is completely transparent. The user does not have to recompile the code, the only requirement is the dynamic linking of the library. The STUB library will be used on the client device, and allows non-GPGPU-capable machines to be accelerated by

CUDA kernels. More on how the communication happens is discussed in the next Sections.

## 3.2 GVirtuS Frontend

As already said, GVirtuS uses a split-driver approach. The frontend is responsible to allow CUDA applications to be virtualized through the STUB Library discussed in Section 3.1. The library actually invokes the frontend. It will perform the actual data transmission and pointers management [15].

## 3.3 GVirtuS Backend

The backend is a server application that runs on the CUDA-capable machine. It's responsible to actually perform the CUDA function calls or CUDA kernels on behalf of the client. When the backend is launched, it will dynamically load all the requested endpoints and their communicators. It will wait for connections and tasks to execute. When a client submits a function, it performs the actual computation, and eventually sends back the results.

## 3.4 GVirtuS Communicators

The communicator is the actual mean of data transmission of GVirtuS. It performs high-performance data transfers between the frontend and the backend. It is a pluggable module with a well defined API (check Section 4.1 for API specifications), in order to allow an easy development of different kinds of communicators to match diverse use-cases or network ecosystems. In Section 4 and Section 5 we discuss how a new Communicator module based on RDMA over Infiniband was developed.

## 3.5 GVirtuS Execution

We just discussed the duties of frontend, backend, STUB Library and communicators modules. Now let's take a look on what actually happens under

the hood when a CUDA routine is called. The frontend execution flow is the following:

1. The CUDA Application is executed on the non-CUDA-capable machine;
2. The CUDA Application calls a CUDA function, but it's actually a call to a wrapper function in the GVirtuS CUDA STUB Library instead;
3. The wrapper function retrieves the Frontend. If none is instantiated it creates a new Frontend, it initializes it with the Init method instance and adds it into a map containing all the Frontend instances, associated with the Thread ID of the thread where a specific Frontend instance is running. The Init method retrieves the properties.json configuration file path, passes it to the get_endpoint method of the Endpoint Factory class, and gets the needed endpoint to generate a communicator. Next it calls the get_frontend method of the Communicator Factory class, passing the newly instantiated endpoint as parameter and getting a Communicator as a result. It finally calls the Connect method of the communicator.
4. The wrapper function prepares the execution of the requested routine using the Prepare method of the Frontend;
5. The wrapper function adds all the input parameters needed for the execution using the different Add methods provided by the Frontend;
6. The wrapper function requests the execution of a named routine with the Execute method provided by the Frontend, passing as parameters the string representing the requested routine;
7. The Execute method retrieves the communicator and sends the routine to the backend;
8. When the result is ready, it is handled into the wrapper function code and then the cuda exit code is retrieved and returned.

Something interesting happens here: the frontend does not have a main. The actual main of the client side is the CUDA Application itself. The frontend is just a class used by the CUDA STUB Library to handle the connection and other memory management operations (see Section 3.6).

Regarding the backend, this is the execution flow:

1. The backend is executed on the GPGPU-accelerated machine;
2. The main creates a new backend instance, passing the properties.json path to the constructor;
3. The properties.json file is parsed by the backend constructor: all the requested plugins are loaded at runtime and all the required communicators are created. To create a communicator, the properties.json path is passed to the Endpoint Factory, the newly created endpoint is then passed to the Communicator Factory, and finally a Process object holding the communicator is saved into an array;
4. Once the backend is successfully created, all the Process object we just created are started using the Start method of the Process class after doing a fork into the backend object. Each process will handle one kind of connection.
5. The Start method of the Process class dynamically loads all the required plugins;
6. The communicator associated with the Process object is put in serving state through the Serve method;
7. Once a new connection is incoming, it will be accepted and handled in a thread;
8. In the thread, the getstring function is called to read the client's requested function;

9. If the function is known, it is executed through the GVirtuS plug-in library;

10. Once the function execution is completed, the result is finally sent back to the client.

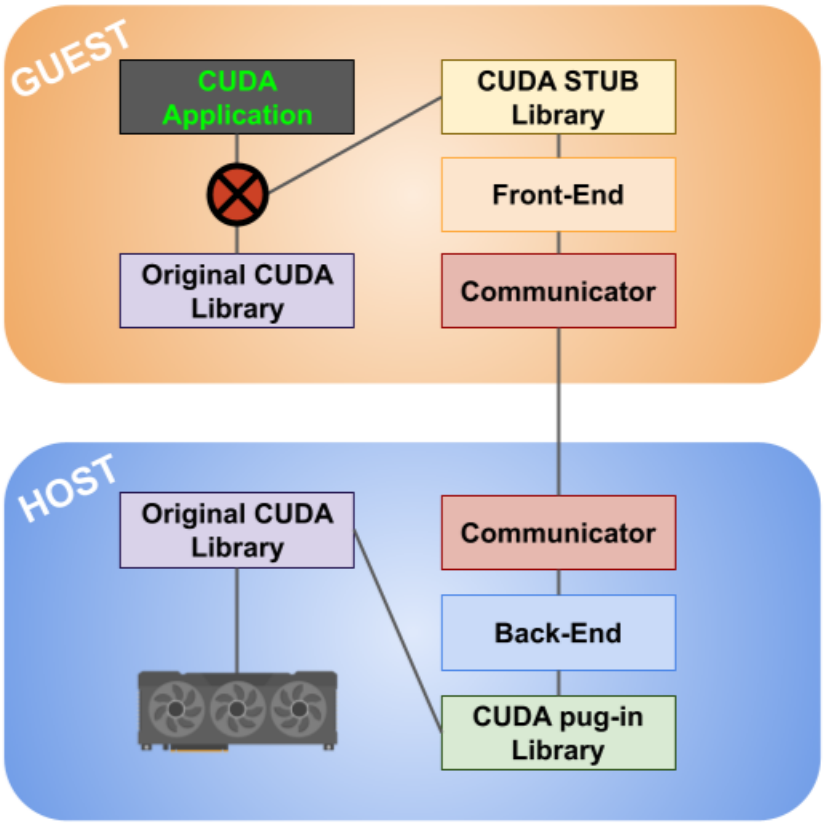


Figure 4: GVirtuS data flow

## 3.6 GVirtuS Unified Virtual Addressing (UVA) Support

CUDA 6.x introduces Unified Virtual Addressing support. It simplifies the programming of GPU-accelerated applications by allowing the user to allocate the memory with a single pointer accessible from both the CPU and the GPU

(Fig. 5). This approach is undoubtedly programmer-friendly, since calls to memory management functions like cudaMalloc, cudaMallocManaged work with a single unified memory space thanks to pointer mappings [16].

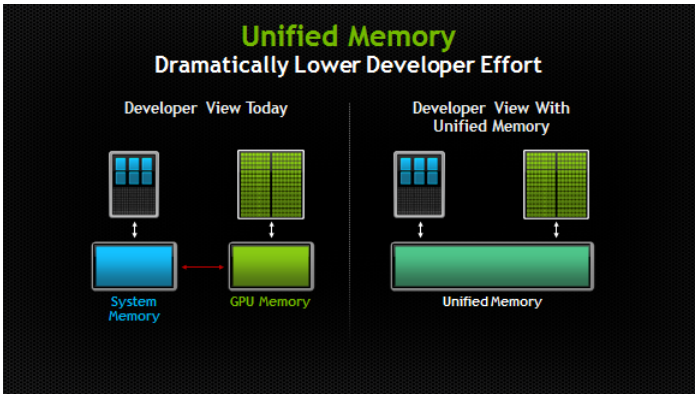


Figure 5: CUDA 6.x Unified Memory comparison

GVirtuS does support CUDA-UVA. To achieve it, the developers found a clever solution: the pointer resulting from calls to cudaMalloc family functions are stored into a list in order to easily identify the nature of the pointer. In addition, anytime a managed pointer is allocated, it will be stored in a map with its size and a host pointer allocated through a regular glibc malloc. In this way, when the managed pointer is used, GVirtuS can therefore search for a match in the map, in order to ensure coherence between the virtualized and the remote address spaces (Fig. 6). After the computation, the pointer mappings are transferred from the backend to the frontend to ensure that the processed data is available on the frontend (Fig. 7) [17].

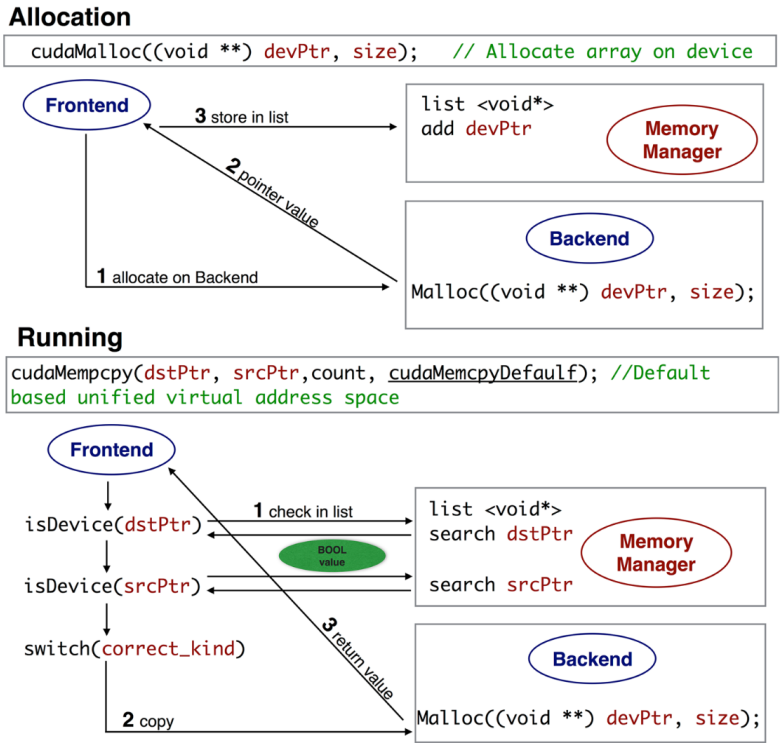


Figure 6: GVirtuS' automatic memory management

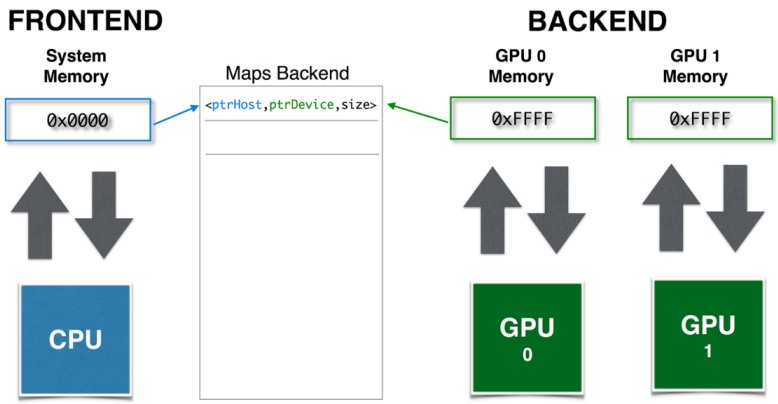


Figure 7: Backend pointer mapping

# 4 Design

In this section we will discuss how we designed a novel RDMA Communicator for GVirtuS.

## 4.1 GVirtuS Communicator Interface

One of the strenghts of GVirtuS is its modularity. New plug-ins and components can be easily developed because of the modular nature of GVirtuS. As to the communicator development, GVirtuS offers a Communicator Interface that exposes all the methods that a communicator must have in order to be compatible with GVirtuS (Fig. 8).

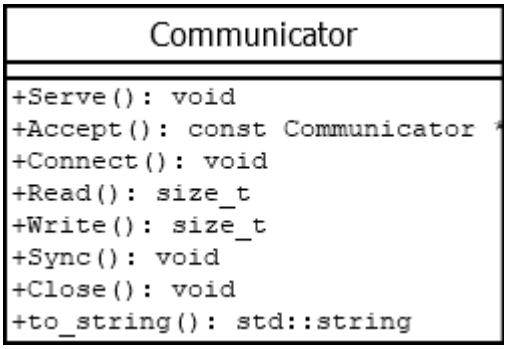


Figure 8: Communicator Interface of GVirtuS

These methods are:

- **Serve**: Sets the communicator as a server;
- **Accept**: Accepts an incoming connection and returns the communicator associated with it;
- **Connect**: Sets the communicator as a clent and tries to connect to an endpoint;
- **Read**: Simply reads the data sent from the peer;
- **Write**: Simply sends data to the peer;

- **Sync**: Actually useful in stream based communicators. It's used to flush communication streams, if no streams are used, the method is expected to do nothing;
- **Close**: Closes the connection with the endpoint;
- **to_string**: Returns a string that represents the communicator.

The Communicator methods must be used in a specific order. For instance, before using Read and Write calls to perform the data transmission, a reliable connection must be established using Serve, Accept and Connect, but Accept must be used after setting the Communicator in serving state using Serve, and so on.

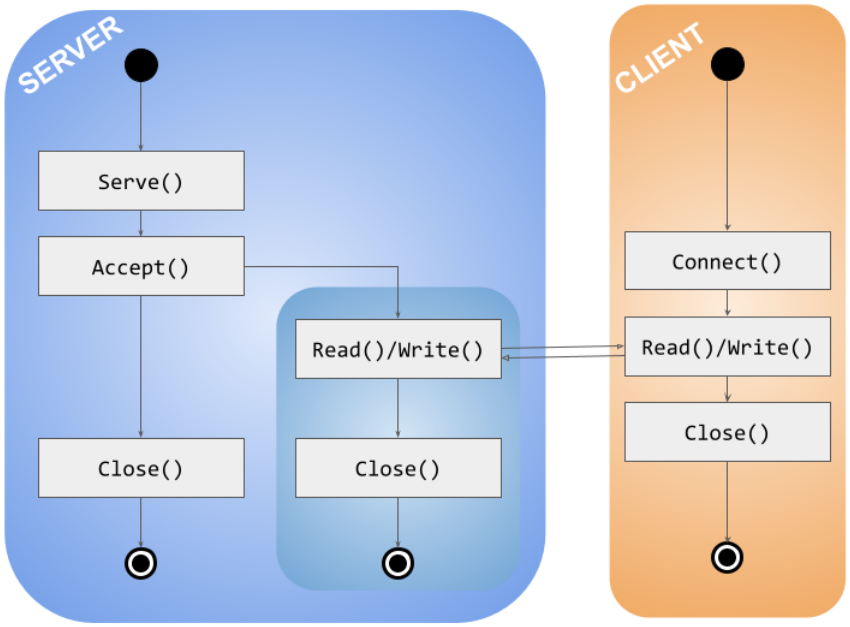


Figure 9: Communicator Interface flow

Now we will take a look to a basic implementation of an RDMA Client/Server application.

## 4.2 A basic RDMA client/server

A simple RDMA Server application programmed with the GVirtuS communicator interface should at least perform the following operations:

1. Instantiate a new communicator;
2. Put the newly instantiated communicator in server mode using Serve method in order to allow it to listen to incoming connection requests;
3. Accept an incoming connection request with the Accept method and save the communicator returned by this method. The new communicator will be used to handle the communication;
4. Perform data transmission using Read method to get data and Send method to send data;
5. Use the Close method to end the communication.

The following pseudocode describes a basic operation flow for a Server application that uses the GVirtuS communicator interface:

```
1    comm = new Communicator
2
3    comm.Serve()
4    client_comm = c.Accept()
5
6    buf = new Buffer of length 6
7    client_comm.Read(buf, 6)
8
9    client_comm.Close()
```

Listing 1: Simple Server implemented with the GVirtuS Communicator Interface that reads a message from the Client

The server application instantiates a new communicator, puts it in serving state using Serve, waits for a connection with Accept, the latter returns a new communicator for handling the connection. Communication can now happen. In

the example we simply read a message from the client. When the communication is done, we can now close it with Close. Likewise, a simple RDMA client programmed with the GVirtuS communicator interface should perform at least the following operations:

1. Instantiate a new communicator;
2. Initiate a new connection with the Connect method;
3. Perform data transmission Read method to get data and Send method to send data;
4. Use the Close method to end the communication.

The following pseudocode describes a basic operation flow for a Client application that uses the GVirtuS communicator interface:

```
1       comm = new Communicator
2
3       comm.Connect()
4
5       buf = "Hello!"
6       comm.Write(buf, 6)
7
8       c.Close()
```

Listing 2: Simple Client implemented with the GVirtuS Communicator Interface that sends a message to the Server

The client application instantiates a new communicator and connects it to an endpoint using Connect. Communication can now happen. In the example we simply sent a message to the server. When the communication is done, we can now close it with Close.

## 4.3 Designing an RDMA Communicator for GVirtuS

As seen in Section 4.1, the GVirtuS communicator interface is a simple yet effective way to abstract the communication needs of GVirtuS. Now we will leverage it to design an RDMA Communicator. Doing so is really easy: a new class is created, implementing the Communicator interface (Fig. 10). We will

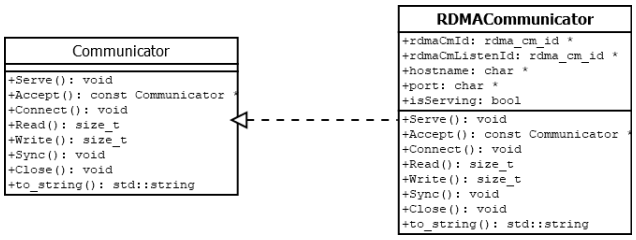


Figure 10: Implementation of GVirtuS communicator interface

discuss about the implementations of the methods in Section 5. This module will be loaded at runtime.

## 4.4 The properties.json file

The user can choose which communicator to use by editing the properties.json file. Such file looks like this:

```
1    {
2  "communicator": [
3    {
4      "endpoint": {
5        "suite": "tcp/ip",
6        "protocol": "tcp",
7        "server_address": "127.0.0.1",
8        "port": "9999"
9      },
10     "plugins": [
11       "cudart",
```

```
12            "cublas",
13            "curand",
14            "cudnn"
15          ]
16      }
17    ],
18    "secure_application": false
19 }
```

Listing 3: The properties.json file

The interesting section of the file is the endpoint one. By editing the suite and protocol values, one can choose which communicator will be loaded. By editing the server_address and port values, one can choose the IP and PORT of the remote machine where the backend is running. As for now, the two main communicator supported by GVirtuS are the TCP Communicator and the novel RDMA Communicator discussed in this work. To use the TCP Communicator the above json file should be edited like this:

```
1       "suite": "tcp/ip",
2       "protocol": "tcp",
```

Listing 4: TCP/IP suite and protocol configuration

Likewise, to use the RDMA Communicator the above json file should be edited like this instead:

```
1       "suite": "infiniband-rdma",
2       "protocol": "ib",
```

Listing 5: RDMA suite and protocol configuration

This file will be read and parsed by GVirtuS. Two Factory classes will then cooperate to load the actual communicator. It was needed to add our communicator into the factory classes, but nothing particularly relevant needed to be done. The actual operation flow for loading a communicator is the following:

1. GVirtuS reads the properties.json file;

2. The file is passed to an Endpoint Factory class. This class will instantiate and return the actual endpoint;

3. The endpoint is then passed to a Communicator Factory class. This class will dynamically load the correct communicator library based on the endpoint information provided.

## 4.5 The Endpoint RDMA class

As it can be seen, to load the communicator at runtime, another component is needed to be developed: the Endpoint RDMA Class. This class will be responsible of representing the endpoint property, carrying the informations of the properties.json file we just discussed. GVirtuS provides an Endpoint interface too, so the development of such class was not hard. Actually, we only needed to implement the interface. The source code is available in the Implementation Section.

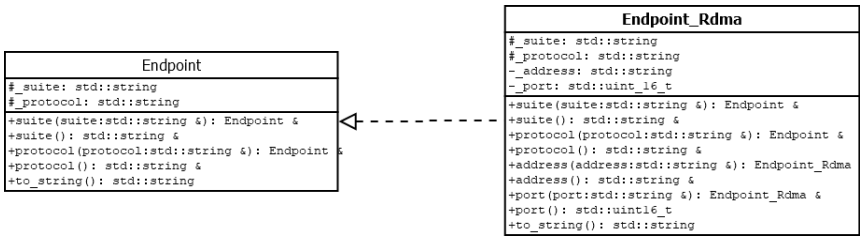


Figure 11: Implementation of the Endpoint_Rdma class

## 4.6 The getstring function

Last but not least, getstring function in the Process file needed a little tweak. This function is responsible to read a string from the communicator. The actual implementation of getstring relies on the stream nature of TCP communication, but our communicator does not rely on streams to perform data transmission. So we needed to check which communicator was loaded and perform a different way

of reading the string from it. Refactoring is needed since Liskov's Substitution and Dipendence Inversion principles are violated, but it's out of the scope of this work.

# 5 Implementation

In this Section we will dig deeper into the implementation details of our RDMA Communicator, and what had to be done to make it work.

## 5.1 Used Technologies

To develop our communicator, we had to rely on different technologies, libraries, SDKs and development environments. When it comes to languages, SDKs and libraries, the choice of C++17 [18] as our programming language was obliged, since GVirtuS is coded in C++. In addition, we had to naturally use CUDA 11.4.2 [3] to develop our test programs and evaluate the communicator's performances. Last but not least, we relied on libibverbs [19] and librdmacm [20] to set up our communicator for "RDMA over Infiniband". We compiled the source code using gcc-8.3.1 [21], cmake-3.25.2 [22] and nvcc [23]. We relied on CLion [24] as the IDE of choice because of its remote development feature. The latter was really useful, allowing us to immediately deploy our project in a remote HPC cluster (view Section 6.1). We also developed a really simple wrapper library for rdma-cm. The only purpose of the library is to check for errors and throw an exception if one occurs.

## 5.2 Communicator Abstract Class

As said before, GVirtuS offers a Communicator Interface to ease the designing and development of new communicators. The interface is implemented as a C++ abstract class, and all the methods are marked as virtual and will be implemented by the programmer in child classes. The source code of the communicator interface is available in Appendix 8.1.

## 5.3 A first approach to an RDMA Communicator

The RDMA over Infiniband communicator discussed in this work was implemented as a C++ class. In particular, as mentioned in Section 4.3, the

RDMACommunicator class is an implementation of the Communicator interface, therefore we had to implement the Serve, Accept, Connect, Read, Write and Sync methods. We also had to add other attributes to hold some necessary and useful data. These attributes are two rdma_cm_id pointers to hold the RDMA Communication Managers used representing the connection, two C-style strings holding the hostname and port, an ibv_wc field to hold the Work Completion used for checking if a send or recv operation is completed and an ibv_mr pointer to register the input/output buffers for messaging. The source code of the RdmaCommunicator class is available in the appendix. Now we will look at how we did a first implementation of the Read and Write methods. We disclose that these implementations have horrible performances. RDMA technology can allow really high throughput and low latency, but it has some hidden costs that could (and did) have a significant negative impact on performance.

### 5.3.1 Unoptimized RdmaCommunicator::Read()

The Read method is one of the two methods that will actually perform data transmission. Its purpose is to receive data from the peer, and write it into a buffer. As already said before countless times in this work, RDMA does not rely on streams like TCP does, so we had to develop this method accordingly. The Verbs API provides two types of primitives: one-sided primitives (RDMA Read/Write) and two-sided primitives (RDMA Send/Recv). Since GVirtuS is a two-sided kind of communication, we decided to rely on RDMA Send and Recv primitives, because they allow a straightforward synchronous communication. The Recv primitive is the RDMA counterpart of the read system call, but it does not rely on any kind of system call and/or context switch to read data from the sender. The Recv verb postes a Recv Work Request to the Receive Queue of a Queue Pair.

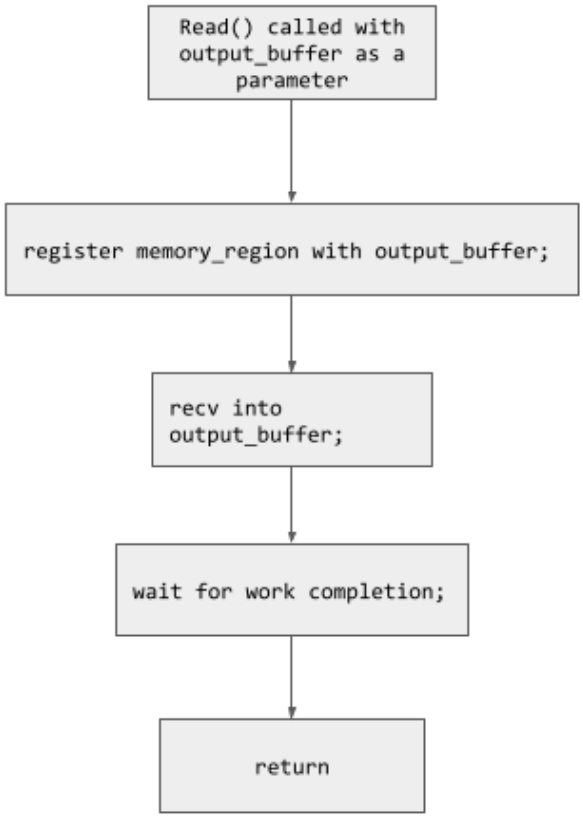


Figure 12: Unoptimized RDMA Communicator Read flow

To post a Recv Work Request through rdma-cm verbs, one should provide an rdma_cm_id, a buffer where the data will be written, the size of the buffer and a memory region associated with the buffer. The memory region associated with the buffer must be registered before any data transmission is performed, but this registration involves the kernel, hence it causes a context switch. It is actually the main reason why this first implementation is not efficient. To sync and wait for the Recv Request to be completed, the ibv_poll_cq function is used. This function polls Work Completions from a Completion Queue. Polling for a Work Completion basically means waiting for a Work Request to be completed. This function is called inside a while cycle until a Work Completion is read.

Summing up, the Read method does these operations: The actual source code of the unoptimized Read method is available in Appendix 8.3.

### 5.3.2 Unoptimized RdmaCommunicator::Write()

The Write method is the other method that actually performs data transmission. It's purpose is to send data to the receiver side of the connection. It leverages the Send verb primitive. The Write method works in the exact same way of the Read one:

1. Registers a Memory Region associated with the input buffer passed as parameter;
2. Posts a Send Work Request;
3. Waits for the Work Completion;

The unoptimized Write source code is available in Appendix 8.4. There is one little caveat in the code: The communicator interface needs the Write method to take a const char * as an input parameter, but the rdma_post_recv function does take a char * (not const)! Therefore we had to memcpy the const char * buffer into a newly malloc-ed buffer.

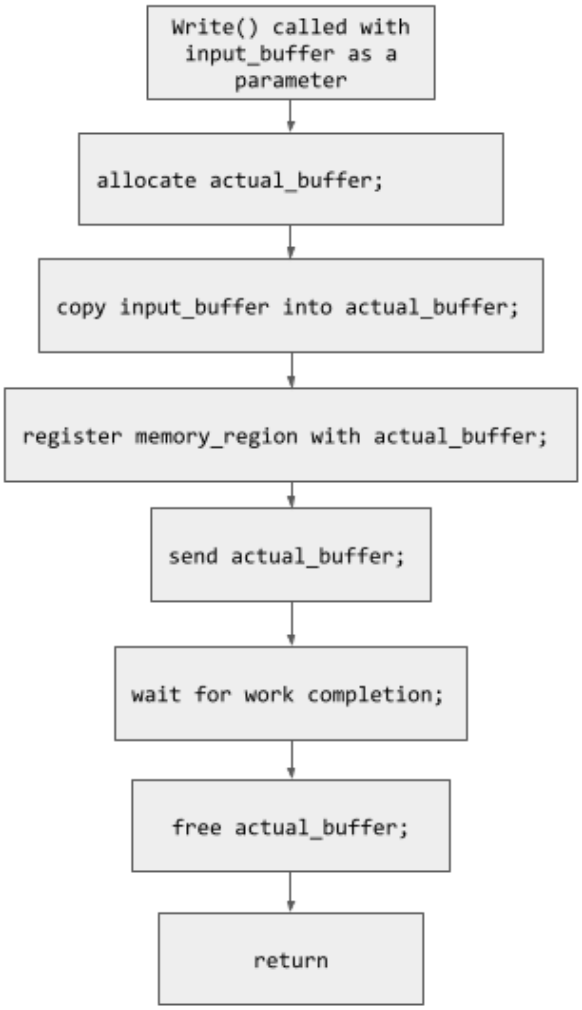


Figure 13: Unoptimized RDMA Communicator Write flow

## 5.4 Optimizing the RDMA code

The RDMA technology is able to provide great performances for data transmission because of its design based on bypassing the operating system while moving data from two peers. These performances boost is possible because the data operation of RDMA do not rely on system calls, and it does not copy the same buffer over and over (like TCP/IP does), consequently avoiding the overhead introduced by the context switches performed by these operations. In addition, Infiniband hardware is really fast and efficient in transmitting big data. However, RDMA has some hidden costs that could make the performances even worse than using a regular TCP/IP-based communicator.

### 5.4.1 Data operations and Control operations

The main hidden cost of RDMA is the *"use of control operation in the data path"*. The verbs API provides two kind of operation: data operations and control operations. We consider as data operation those that have the role of sending data back and forth between the communication peers. Therefore, verbs to send and receive data and verbs to confirm the completion of the send/recv operations are considered data operations. The data operations are really optimized, and they work without performing any context switch. On the other hand, control operations (such as the different create, destroy, modify, query) are really expensive because they perform context switches and work with dynamic memory. Consequently, one should avoid calling control operations in the data path. In the next Sections we will see why this insight is critical.

### 5.4.2 Memory Region Preregistration

To perform send and recv data operations, the RDMA device needs a memory region associated with the buffer that one wants to use during the transmission. This memory has to be registered before the data operations are performed, this memory registration is really expensive because of the reason we just discussed. Initially we registered the memory region right before performing

a send or recv call, but we experienced performances so bad that using TCP/IP provided less overhead and system calls count. We then decided to preregister the memory region and reuse the same buffer over and over again, as discussed in [25] and [26]. We're now using a buffer of a certain fixed size. When performing a transmission with data that fits in this buffer, no memory registration occurs. In contrast, when data is bigger than the buffer can accomodate, we then register a new memory region just for this transmission. The following pseudocode shows how this mechanism works:

```
1     let buf be the buffer used into the transmission
2     let size be the size of the transmission
3     let thresh be the size of the preregistered buffer
4     let prereg_buf be the preregistered buffer
5     let prereg_mr be the preregistered memory region
6
7     IF size < thresh
8         IF send
9             copy buf into prereg_buf
10        send/recv(prereg_buf, prereg_mr, size)
11        wait for completion
12        IF recv
13            copy prereg_buf into buf
14    ELSE
15        let mr be a newly allocated memory region
      associated with buf
16        send/recv(buf, mr, size)
17        wait for completion
18
19    IF size > thresh
```

```
20          deregister mr
```

Listing 6: Pseudocode about the usage of Preregistered Buffers in RDMA Communication

According to [26], copying a buffer into another preregistered buffer properly associated to a preregistered memory region is less expensive than registering a new memory region for buffers smaller than a certain size. We achieved a significant performance improvement with this optimization, since we were able to avoid numerous context switches to register the memory region.

### 5.4.3 Polling-based work completion

Another optimization we decided to leverage was the use of polling-based work completion instead of events-based work completion. While this approach increases the CPU usage, it significantly lowers the latency. Since the real-time-like nature of the virtualization, we decided that the drawback of an higher CPU usage was worth it if that means a lower latency.

### 5.4.4 Low retransmission delay

Last but not least, we modified the value of the min_rnr_timer field of the queue pair. This value represents the waited time before a retransmission when something goes wrong during the data transmission and a retry is required. This timer is set to 0.01 milliseconds, which is the lower value allowed. Higher values could lead to high latency because it's not unusual that one of the peers tries to post a send request before the other peer posts a recv request.

## 5.5 The optimized RdmaCommunicator Class

As will be discussed in Section 6, these optimizations are the key for a fast RDMA communication. Consequently we had to tune and tweak some things in the RDMA Communicator class and in the Read and Write methods to obtain good performances. We added a Preregistered Buffer of 5 Kilobytes and

a Preregistered Memory Region attributes, that will be registered for messaging before any Read or Write method is called. The modified source code of the class is available in the appendix.

## 5.6 RdmaCommunicator Constructors

To instantiate a communicator in GVirtuS there's the need of two Constructors. The first one will take the hostname and port as parameters, which will be used by the client to discover the server and by the server to identify on which port it will work. The second one will take an rdma_cm_id as parameter, and will be used by the server to return a Communicator that will be used to handle the connection with the client. In the second communicator, the preregistration of the buffer and memory region are also performed. The code of the two constructors is available in Appendix 8.5.

## 5.7 Communicator Factory and Endpoint Factory classes

Talking about Communicators instantiation, two classes have a key role in this process: the Communicator Factory and the Endpoint Factory classes. GVirtuS exploits the shared libraries to achieve a modular and plug-in style architecture, and it uses a configuration file called properties.json to choose which communicator will be used. We already discussed that we had to add our communicator and Endpoint Rdma class to the factories. To achieve that, we just coded an additional if-else into the get_communicator and get_endpoint methods. The get_communicator function instantiates the communicator by iterating through a vector in order to find out if the desired communicator is supported. The source code of this operation is available in Appendix 8.7. Even if iterating through a vector isn't the pretties solution, we thought that using patterns like the Chain Of Responsibility would be overkill for just instantiating a class. On the other hand, the Endpoint Factory instantiates an endpoint based on the suite and protocol fields of the properties.json file by checking if the suite is supported. The code is available in Appendix 8.8.

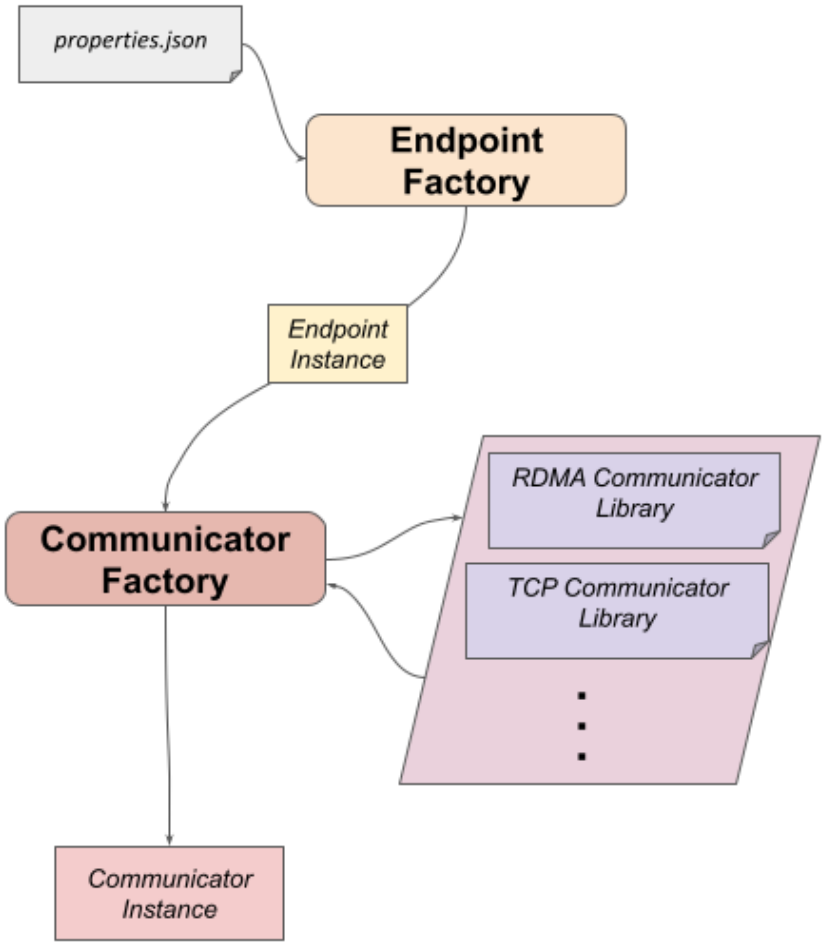


Figure 14: Instantiation of a Communicator

## 5.8 Endpoint Interface

As we have seen, GVirtuS relies on two factory classes to properly instantiate a communicator. The Endpoint Factory has the purpose of instantiating the correct endpoint, in order to pass all the informations a Communicator needs to establish a connection. For each kind of protocol, GVirtuS needs an endpoint that is compatible with the Communicator that will be created. Fortunately,

an Endpoint Interface is provided within the GVirtuS project. It provides the basic methods an Endpoint class should have. The source code of the Endpoint Interface is available in Appendix 8.9. The interface is really simple: it just provides some setters and getters for the suite and protocol attributes, and an equality operator overload.

## 5.9 Endpoint_Rdma Class implementation

To properly set up our RDMA Communicator we had to develop the Endpoint_Rdma Class. The only addition consists in two attributes and their respective setters and getters. These are the address and port attributes, that will hold the IP address and the service of the endpoint. The newly created Endpoint_Rdma class source code is available in Appendix 8.10.

## 5.10 RdmaCommunicator::Serve

Once an instance of the communicator is created, it could either be in client mode or serving mode. The Serve method is used by the RDMA Communicator class to set the communicator as a Server, and allow it to listen for incoming connections. The Serve function performs the following operations:

1. Allocates an rdma_addrinfo struct to hold the hints of services the caller supports. We initialize the ai_port_space field to RDMA_PS_IB and ai_flags to RAI_PASSIVE to indicate that it supports Infiniband port space and that it's the passive side of the connection;

2. Uses the rdma_addrinfo function to resolve the destination host and port and returns information needed to establish communication;

3. Allocates an ibv_qp_init_attr struct to hold the initial attributes of the Queue Pair that is about to be created. The QP type will be set to Reliable Connection (RC).

4. It initializes the Queue Pair and calls the rdma_listen function to wait for incoming connections.

After all these operations, the communicator is now set as a Server and it's listening for incoming connections. This method must be called before the Accept method. The source code of the Serve method is available in Appendix 8.11.

## 5.11 RdmaCommunicator::Accept

Once the communicator is set as a server and is actively listening for connections, it should be able to accept any connection request. To make this possible, we implement the Accept method. This method is really simple: it just calls the rdma_get_request function to retrieve a connection request event, and then it calls the rdma_accept function to finally establish the connection. Before returning a new RDMA Communicator instance to handle the connection, it sets the min_rnr_timer Queue Pair Attribute to the lower value possible, as already discussed in 5.4. The Accept code is available in Appendix 8.12.

## 5.12 RdmaCommunicator::Connect

The client counterpart of the Serve and Accept methods is the Connect method. This method will be used on the client side to initiate a connection request to a remote destination. The Connect method basically perform the same operations of the Serve and Accept methods, with just some small tweaks and additions:

1. Allocates an rdma_addrinfo struct to hold the hints of services the caller supports. We initialize the ai_port_space field to RDMA_PS_IB, but this time we do not set the ai_flags to RAI_PASSIVE. With these tunings, we're indicating that the communicator supports Infiniband port space and that it's the active side of the connection;

2. Uses the rdma_addrinfo function to resolve the destination host and port and returns information needed to establish communication;

3. Allocates an ibv_qp_init_attr struct to hold the initial attributes of the

Queue Pair that is about to be created. The values are the same of those used for the Serve method;

4. It initializes the Queue Pair and calls the rdma_connect function to initiate a connection request to a remote destination.

5. Once the connection is established, it sets the min_rnr_timer Queue Pair Attribute to the lower value possible;

6. Before finally returning, it performs the buffer and memory region preregistration.

The source code of the Connect method is available in Appendix 8.13.

## 5.13 Optimized RdmaCommunicator::Read

We already had a look at an unoptimized Read implementation. Now let's talk about how we optimized the code. As already discussed, control operation should be avoided in the data path because they cause context switches. To achieve this, we preregistered the memory region before the Read (or Write) calls. If the output buffer passed as parameter is not bigger than the preregistered buffer, we just read the data into it and then we call a memcpy to copy the data back into the output buffer. If the output buffer is bigger than the preregistered buffer, we then register it in a new Memory Region and then read the data into it. No copy is needed in this case. The RDMACommunicator Read implementation is available in Appendix 8.14.

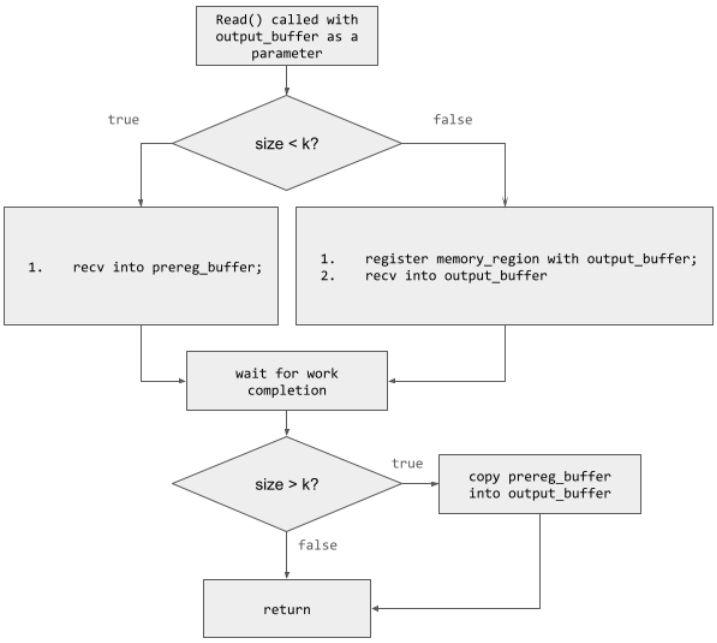


Figure 15: Optimized RDMA Communicator Read flow

## 5.14 Modified getstring() function

The getstring function isn't a method of RDMACommunicator, but it is strictly correlated to the Read method. It's used by the Process class and its purpose is to read a string sent from the peer. This string is usually a CUDA function symbol. As previously mentioned in Section 4.3, we had to tweak it a little bit to make it compatible with our communicator. The original implementation relied on the stream nature of TCP communication, and it's implementation is based on a while cycle that reads one char from the stream using the Read implementation of TCPCommunicator class, and appends that char to a string. When an EOF is encountered, the cycle ends and the string is returned. This implementation was incompatible with our Read implementation, since RDMA does not exploit streams to receive data, but it directly writes it into a buffer. We therefore used the to_string method to identify the type of the communicator that is being used, and implemented another logic for our RDMACommunicator. While this approach does violate some SOLID principles, refactoring was out of the scope of this work. The logic we implemented is the following: if the communicator is an RDMACommunicator, it tries to read 30 characters using the Read method of RDMACommunicator. We've chosen 30 as our size because it was enough for the function called during our testings. With the implementation of new plugins it could be not enough, but it's an easy fix anyway. After the read, the message is appended to the output string and the getstring function returns. The real code of the getstring function is available in Appendix 8.15.

## 5.15 Optimized RdmaCommunicator::Write

As for the Read method, also the Write method was modified to leverage the optimizations discussed in 5.4. Now the method performs the dynamic allocation of the actual buffer only if the input buffer is bigger than the preregistered buffer. This is possible because the preregistered buffer was already allocated in advance, therefore we can avoid an additional system call. The optimized

Write method code is available in Appendix 8.16.

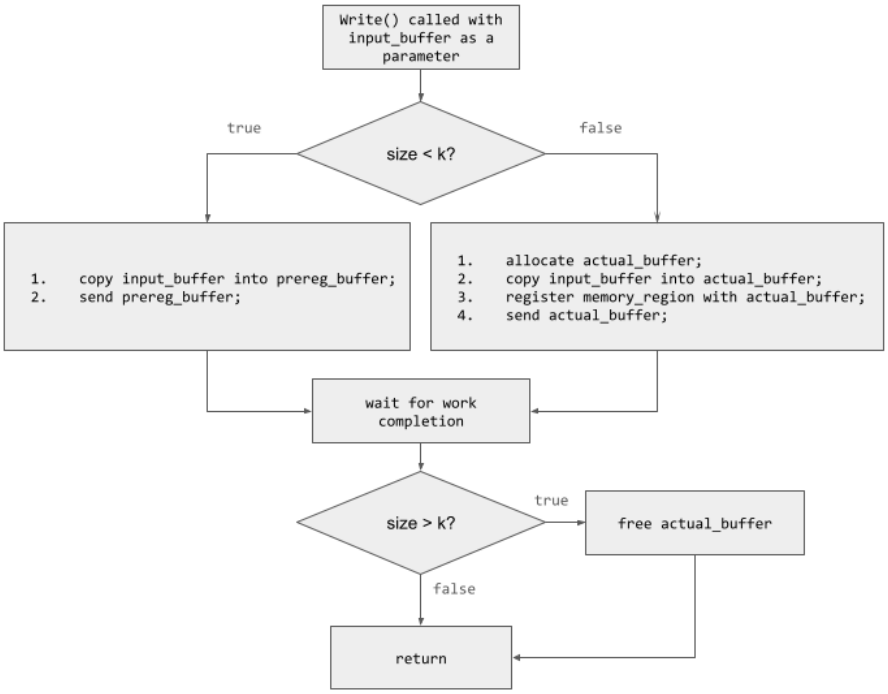


Figure 16: Optimized RdmaCommunicator Write flow

## 5.16 RdmaCommunicator::Sync

This method does actually nothing. In the TCPCommunicator class its purpose is to flush the output stream used for communication. Since RDMA-Communicator class does not rely on streams, the method is actually useless and therefore it's empty.

# 6 Evaluation

In this Section we will discuss about the performance evaluations of the RDMA Communicator that was developed for this work.

## 6.1 The HPC Cluster "Purple Jeans"

Since GVirtuS main purpose is to allow the leverage of a remote GPU, we decided to deploy the backend on the HPC Cluster "Purple Jeans" of University of Naples Parthenope [27]. Purple Jeans is a research HPC Cluster that students, PhD students, professors and researchers of Parthenope University can use. It can boast 4 GPU-accelerated nodes. Each node rocks four nVidia Tesla

| Resource | Partitions | Nodes | CPU/Node | Core/ Node | Memory/ Node | Notes |
|---|---|---|---|---|---|---|
| purpleJeans (2019) | xhicpu | 4 | 2 CPU Intel(R) Xeon(R) Xeon 16-Core 5218 2,3Ghz 22MB | 32 | 192 GB | Cluster dedicated to ML and Big Data researches. Mellanox CX4 VPI SinglePort FDR IB 56Gb/s x16. |
| | xgpu | 4 | 2 CPU Intel(R) Xeon(R) Xeon 16-Core 5218 2,3Ghz 22MB | 32 | 192 GB | 4 GPU (NVLINK) NVIDIA Tesla V100 32GB SXM2. Infiniband Mellanox CX4 VPI SinglePort FDR IB 56Gb/s x16. |

Figure 17: Purple Jeans Nodes

V100, for a total of 16 Tesla V100 GPUs. It certainly is a great platform to perform GPGPUs Virtualization tests. In addition, each node is equipped with Mellanox Infiniband hardware, which is crucial for the kind of tests we're about to perform.

## 6.2 Algorithms

We needed to perform our tests with some CUDA compatible algorithms. The candidates were a Saxpy algorithm and a Matrix Multiplication algorithm. These two were perfect candidates because they play a crucial role in many numerical computations that underpin Artificial Intelligence computations, and AI does a massive use of GPGPUs to accelerate the inference.

### 6.2.1 Saxpy Algorithm

The first algorithm we choose is a Saxpy algorithm. Saxpy stands for "Single-Precision A by X plus Y", and it consists in multiplying each element of a vector X by a scalar A and add the result to the corresponding element in vector Y, according to the following formula:

$$z = ax + y$$

We implemented it as a CUDA Kernel as described in the nVidia blog [28]. The next source code shows the SAXPY implementation as a CUDA Kernel:

```
1  __global__
2  void saxpy(int n, float a, float *x, float *y) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i < n)
5          y[i] = a*x[i] + y[i];
6  }
```

Listing 7: Saxpy CUDA Kernel

$$\begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \begin{bmatrix} y_1 \\ . \\ . \\ . \\ y_n \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix}$$

Figure 18: Saxpy visualization

### 6.2.2 Matrix Multiplication Algorithm

The second algorithm we choose is a Matrix Multiplication Algorithm. It consists in doing the dot product of the rows of a matrix A and the columns of

a matrix B, and put the resulting scalar in the corresponding entry in the result matrix C, according to the following formula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Since the source code is particularly long, it will be available on the GitHub page of GVirtuS [29].

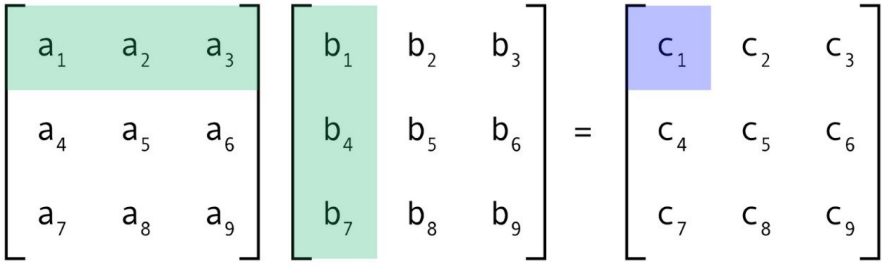


Figure 19: Matrix Multiplication visualization

## 6.3 CUDA Performance Metrics

To evaluate the performances of the communicators, we decided to use the CUDA events API [30] described in the CUDA Runtime API. The CUDA events API fits perfectly our scope because it is able to provide a light-weight performance metric alternative to CPU timers. As discussed in nVidia blog, the latter aren't a good fit for CUDA performance metrics because they require host-device syncronization points that will stall the GPU pipeline, and therefore they will invalidate the results. The example in the nVidia blog shows the performance metrics performed on the Saxpy algorithm we will use. However, the source code provided by nVidia only records the Saxpy kernel execution time, leaving out the host-to-device memory copy and the device-to-host memory copy. Our goal is to evaluate the performances of the whole execution including memory copies, since the virtualization of a GPGPU involves the memory operations

too. We had to tweak their code a little bit accordingly. We moved the cudaEventRecord(start) call before the host-to-device cudaMemcpy calls, and we moved the cudaEventRecord(stop) call after the device-to-host cudaMemcpy call. In this way, we can measure the whole execution including memory copies. The source code of both implementations is available in Appendix 8.17 and Appendix 8.18.

## 6.4 /usr/bin/time -v

Since RDMA should perform less context switches when sending data back and forth, we thought it would be interesting to evaluate the number of voluntary context switches GVirtuS performs while using both the TCP-based Communicator and the RDMA-based communicator. To accomplish this goal, we exploited the /usr/bin/time bash built in command, with the verbosity option.

## 6.5 Setup

To perform our tests we used the following setup: The Backend was deployed on one of the GPU nodes of Purple Jeans. We will exploit one of the four GPUs installed in the node. The frontend was deployed on Purple Jeans Frontend node. This node is equipped with an nVidia RTX Quadro 6000, but we won't use it since the CUDA function calls will be intercepted and offloaded. We evaluated the performance of our test programs in five different configurations:

- **Bare Metal**: the test programs are executed directly on the hardware, without any virtualization;
- **TCP/IP communication**: the test programs are executed through GVirtuS using the TCP communicator over a standard network adapter;
- **IP over Infiniband communication (IPoIB)**: the test programs are executed through GVirtuS using the TCP communicator and Infiniband hardware adapters;

- **RDMA over Infiniband communication (unoptimized)**: the test programs are executed through GVirtuS using the novel RDMA Communicator without the optimizations and with Infiniband hardware;
- **RDMA over Infiniband communication (optimized)**: the test programs are executed through GVirtuS using the novel RDMA Communicator with all the optimizations discussed in Section 5.4 and Infiniband hardware.

We decided to perform the tests with these five setups for different reasons: A Bare Metal evaluation is needed to show how the test programs run without any kind of virtualization. It will be useful to compare the non-virtualized performances with the virtualized performances, and understand how much overhead is introduced by the virtualization through GVirtuS. As already said, we evaluated the TCP/IP Communicator both on an Ethernet network and Infiniband network. This is done to show the impact of the Infiniband hardware on the transmission speed and set the bar for the RDMA Communicator. Finally, we evaluated the RDMA Communicator, with both optimized and unoptimized implementations. This is done to show the impact of the optimizations discussed in Section 5.4 and compare the optimized RDMA Communicator to the TCP/IP communicator in both Ethernet and IPoIB configurations.

## 6.6 Results

We performed multiple executions for each algorithm and setup combination. The data we will show is an average of the execution times. We reported the averages because in a virtualization environment the applications will be usually run for an extended period of time, consequently reporting the average execution time is more important than showing the fluctuations. We will report both the execution times and the voluntary context switches count. We thought reporting the number of voluntary context switches would be interesting because the optimizations of the RDMA Communicator are aimed to improve

performances through a reduction of the overhead introduced by the execution of numerous context switches.

### 6.6.1 Matrix Multiplication Performance Evaluation Results

The first test was performed on the Matrix Multiplication Algorithm. We evaluated the performances with the four virtualization setups and the bare metal configuration, and the results are the followings:

- **MatrixMul Bare Metal**: 52 milliseconds on average;
- **MatrixMul TCP/IP communication**: 647 milliseconds on average;
- **MatrixMul IPoIB communication**: 439 milliseconds on average;
- **MatrixMul RDMA communication (unoptimized)**: 2450 milliseconds on average;
- **MatrixMul RDMA communication (optimized)**: 287 milliseconds on average;

Our evaluations shows that the Optimized RDMA Communicator has better performances than all the other communicators, but the Unoptimized RDMA Communicator has severely worse performances than all the other communicators. In particular, The Optimized RDMA Communicator shows a 55% performance boost over the regular TCP/IP communication, and a 34% performance boost over the TCP/IP over Infiniband communication. On the other hand, the Unoptimized RDMA Communicator run almost four times slower than the regular TCP/IP communication, and almost nine times slower than the Optimized RDMA Communicator! These results show how important is the minimization of the hidden costs of RDMA discussed in Section 5.4. This enormous difference between the optimized and unoptimized RDMA Communicators is due to the overhead introduced by context switching, as we will see in the next section.

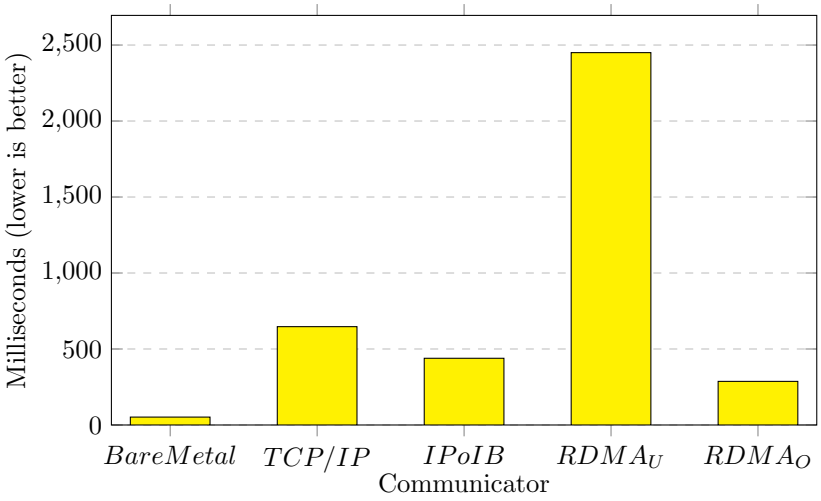


Figure 20: Matrix Multiplication Algorithm Performances Comparison: best result is achieved with our contribution

### 6.6.2 Matrix Multiplication Context Switches Count Results

We already discussed the performance boost that the RDMA Communicator delivers. Now we demonstrate that the reduction of the execution times was possible not only because Infiniband hardware is exceptionally fast, but also because the RDMA Communicator minimizes the number of the performed context switches. We counted the number of voluntary context switches through the /usr/bin/time -v command, and these are the results:

- **MatrixMul TCP/IP communication**: 1016 context switches on average;
- **MatrixMul IPoIB communication**: 956 context switches on average;
- **MatrixMul RDMA communication (unoptimized)**: 11788 context switches on average;
- **MatrixMul RDMA communication (optimized)**: 105 context switches on average;

This test confirms our hypotesis. The Optimized RDMA Communicator

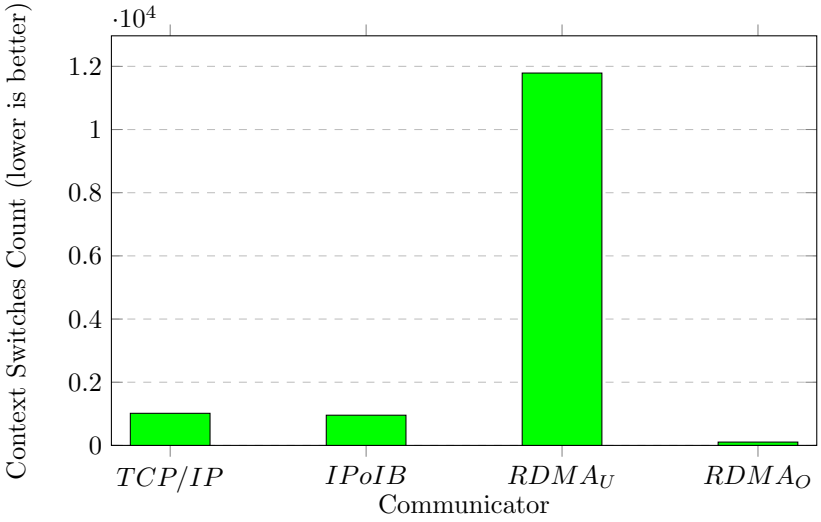


Figure 21: Matrix Multiplication Context Switches: best result is achieved with our contribution

shows a significant smaller context switches count, consequently reducing the over head caused by involving the kernel. On the other hand, the Unoptimized RDMA Communicator shows a really high context switches count, since it has to register the memory region used for the transmission every time a Write or Read call is performed. We will see in the next Sections that the performance boost our RDMA Communicator is capable of delivering is strictly related to the number of context switches performed.

### 6.6.3 SAXPY Performance Evaluation Results

The SAXPY Algorithm used in this test is simpler than the Matrix Multiplication Algorithm we used for the previous tests. The latter performs more cudaMemcpy calls, and therefore it transfers more data between the frontend and the backend. We will now see that the smaller the program, the smaller the performance boost. The execution times of the SAXPY Algorithm are the followings:

- **SAXPY Bare Metal**: 355 milliseconds on average;

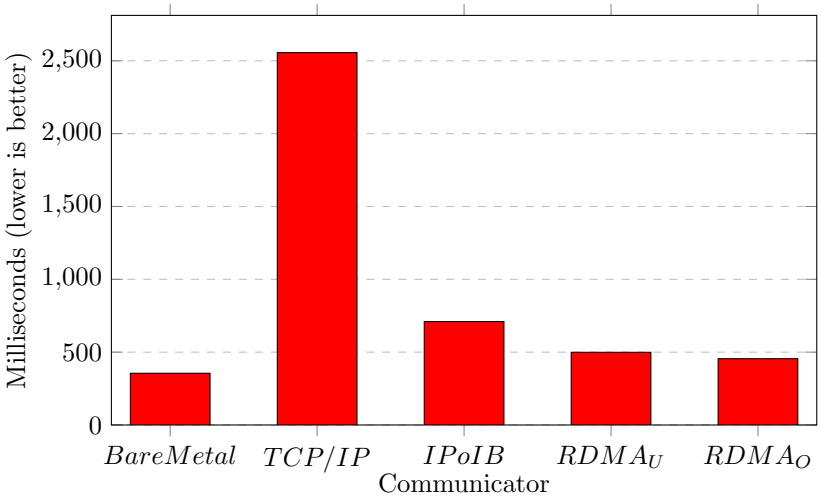


Figure 22: SAXPY Algorithm Performances Comparison: best result is achieved with our contribution

- **SAXPY TCP/IP communication**: 2556 milliseconds on average;
- **SAXPY IPoIB communication**: 710 milliseconds on average;
- **SAXPY RDMA communication (unoptimized)**: 499 milliseconds on average;
- **SAXPY RDMA communication (optimized)**: 455 milliseconds on average;

We were able to appreciate the same boost we observed for the Matrix Multiplication Algorithm, however this time the Unoptimized RDMA Communicator performed significantly better. The reason we were able to see good performances by the unoptimized implementation of the RDMA Communicator lies in the smaller CUDA calls count performed by the SAXPY Algorithm. Even if the Unoptimized RDMA Communicator still has to perform many system calls to register the Memory Regions for data transmission, the number of Read and Write calls is low, hence the system calls count is low and the performances aren't too affected by the context switches overhead. We appreciated an 82% boost if using the Optimized RDMA Communicator over the TCP/IP based

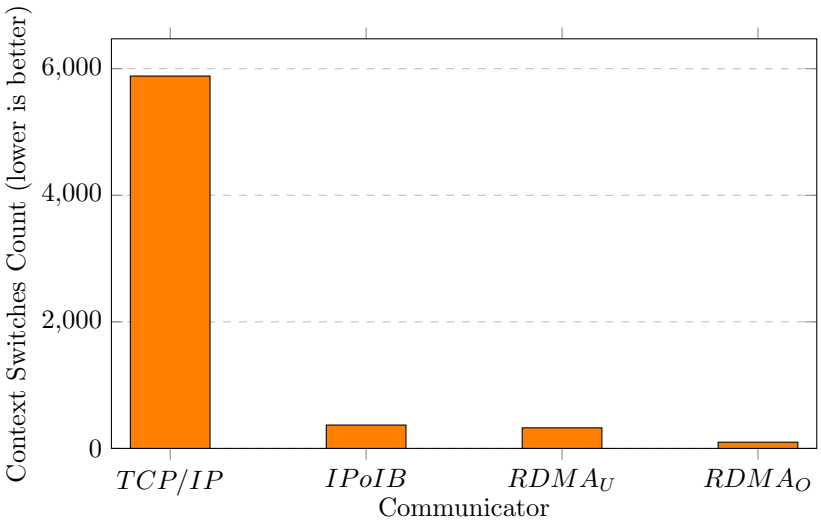


Figure 23: SAXPY Context Switches: best result is achieved with our contribution

Communicator, and a 36% boost over the IPoIB communication. However, this time the boost over the Unoptimized RDMA Communicator is smaller, settling at an 8% boost gained by the Optimized implementation for the reason we just discussed.

### 6.6.4 SAXPY Context Switches Count Results

Finally, let's take a look at the context switches happened during the virtualization of the SAXPY Algorithm. These are the results:

- **SAXPY TCP/IP communication**: 5884 context switches on average;
- **SAXPY IPoIB communication**: 369 context switches on average;
- **SAXPY RDMA communication (unoptimized)**: 325 context switches on average;
- **SAXPY RDMA communication (optimized)**: 98 context switches on average;

Again, we can appreciate the decrease in the context switches count. We have already seen the terrible performances delivered by the TCP/IP communication, and we can confirm again that the overhead introduced by numerous context switches was critical in our evaluations. The TCP/IP communication did more than sixty times the amount of context switches performed by the optimized implementation of the RDMA Communicator. It is easy to grasp how much overhead is introduced this way, and the execution times perfectly reflect this.

# 7 Conclusions and Future Developments

While the novel RDMA Communicator discussed in this work achieved the goal of improving the performances of GVirtuS, it is far from perfect. We optimized the communicator by preregistering the buffer and the associated Memory Region before the communication happens and by using a polling-based work completion. Other approaches could further improve the performances of our communicator, but they didn't fit in the scope of a Bachelor's thesis. Some of the future developments are:

- Finding the best preregistered buffer size with ad-hoc testing, in order to always guarantee the fastest send/recv politics;
- Designing and implementing an RDMA Communicator based on single-sided primitives instead of double-sided primitives, or an hybrid approach;
- Implementing a pipelined solution as the one discussed in [13];
- Implementing cache-alignement for the buffer used in send and recv operations, as discussed in [25];
- Testing our novel RDMA Communicator with more algorithms and applications, such as Artificial Intellicence inference or parallel compression and decompression, and implementing the stub-libraries functions needed to run these applications through GVirtuS;

In any case, our communicator was able to significantly improve the GPGPU remoting performances of GVirtuS. We were able to demonstrate that a performance improvement is possible reducing the overhead introduced by the TCP/IP suite through the numerous memory copies and context switches. Furthermore, we demonstrated that RDMA is a powerful technology, but its development needs a particular attention to details and wise optimization, since our evaluations showed that the Unoptimized RDMA Communicator could be even worse than a traditional TCP Communicator. For the nerds, the curious and

the enthusiasts, the source code of the whole project is available on my personal GVirtuS fork on GitHub [29].

# 8 Source Code Snippets

## 8.1 Communicator Abstract Class

```
1   class Communicator {
2    public:
3
4     virtual ~Communicator() = default;
5
6     virtual void Serve() = 0;
7     virtual const Communicator *const Accept() const =
    0;
8
9     virtual void Connect() = 0;
10
11    virtual size_t Read(char *buffer, size_t size) = 0;
12    virtual size_t Write(const char *buffer, size_t
    size) = 0;
13
14    virtual void Sync() = 0;
15
16    virtual void Close() = 0;
17
18    virtual std::string to_string() { return "
    communicator"; }
19   };
```

Listing 8: Communicator Abstract Class

## 8.2 Unoptimized RdmaCommunicator Class

```
1   class RdmaCommunicator : public Communicator {
```

```
2   private:
3       rdma_cm_id * rdmaCmId;
4       rdma_cm_id * rdmaCmListenId;
5
6       char * hostname;
7       char * port;
8
9       ibv_wc workCompletion;
10      ibv_mr * memoryRegion;
11
12  public:
13      RdmaCommunicator() = default;
14      RdmaCommunicator(char * hostname, char * port);
15      RdmaCommunicator(rdma_cm_id * rdmaCmId);
16
17      ~RdmaCommunicator();
18
19      void Serve();
20      const Communicator *const Accept() const;
21
22      void Connect();
23
24      size_t Read(char * buffer, size_t size);
25      size_t Write(const char * buffer, size_t size);
26
27      void Sync();
28
29      void Close();
30
31      std::string to_string();
```

```
32 };
```

Listing 9: Unoptimized RdmaCommunicator Class

## 8.3 Unoptimized RDMA Communicator Read

```
1    size_t RdmaCommunicator::Read(char *buffer, size_t
     size) {
2        memoryRegion = ktm_rdma_reg_msgs(rdmaCmId, buffer
     , size);
3        ktm_rdma_post_recv(rdmaCmId, nullptr, buffer,
     size, memoryRegion);
4
5        int num_comp;
6        do num_comp = ibv_poll_cq(rdmaCmId->recv_cq, 1, &
     workCompletion); while (num_comp == 0);
7        if (num_comp < 0) throw "ibv_poll_cq() failed";
8        if (workCompletion.status != IBV_WC_SUCCESS)
     throw 'Failed status " + std::string(ibv_wc_status_str
     (workCompletion.status));
9
10       return size;
11   }
```

Listing 10: Unoptimized RdmaCommunicator::Read()

## 8.4 Unoptimized RDMA Communicator Write

```
1    size_t RdmaCommunicator::Write(const char *buffer,
     size_t size) {
2        char * actualBuffer = nullptr;
3
```

```
4           actualBuffer = (char *) malloc(size);
5           memcpy(actualBuffer, buffer, size);
6           memoryRegion = ktm_rdma_reg_msgs(rdmaCmId,
    actualBuffer, size);
7           ktm_rdma_post_recv(rdmaCmId, nullptr,
    actualBuffer, size, memoryRegion);
8
9           int num_comp;
10          do num_comp = ibv_poll_cq(rdmaCmId->send_cq, 1, &
    workCompletion); while (num_comp == 0);
11          if (num_comp < 0) throw "ibv_poll_cq() failed";
12          if (workCompletion.status != IBV_WC_SUCCESS)
    throw 'Failed status " + std::string(ibv_wc_status_str
    (workCompletion.status));
13
14          free(actualBuffer);
15
16          return size;
17      }
```

Listing 11: Unoptimized RdmaCommunicator::Write()

## 8.5 Optimized RdmaCommunicator Class

```
1     class RdmaCommunicator : public Communicator {
2   private:
3       rdma_cm_id * rdmaCmId;
4       rdma_cm_id * rdmaCmListenId;
5
6       char * hostname;
7       char * port;
```

```
 8
 9       ibv_wc workCompletion;
10
11       ibv_mr * memoryRegion;
12
13       char preregisteredBuffer[1024 * 5];
14       ibv_mr * preregisteredMr;
15
16   public:
17       RdmaCommunicator() = default;
18       RdmaCommunicator(char * hostname, char * port);
19       RdmaCommunicator(rdma_cm_id * rdmaCmId);
20
21       ~RdmaCommunicator();
22
23       void Serve();
24       const Communicator *const Accept() const;
25
26       void Connect();
27
28       size_t Read(char * buffer, size_t size);
29       size_t Write(const char * buffer, size_t size);
30
31       void Sync();
32
33       void Close();
34
35       std::string to_string();
36 };
```

Listing 12: Optimized RdmaCommunicator class

### 8.6 RDMA Communicator Constructors

```
1   RdmaCommunicator::RdmaCommunicator(char * hostname,
    char * port) {
2       if (port == nullptr or std::string(port).empty())
     {
3           throw "RdmaCommunicator: Port not specified
    ...";
4       }
5
6       hostent *ent = gethostbyname(hostname);
7       if (ent == NULL) {
8           throw "RdmaCommunicator: Can't resolve
    hostname \"" + std::string(hostname) + "\"...";
9       }
10
11      auto addrLen = ent->h_length;
12      this->hostname = new char[addrLen];
13      memcpy(this->hostname, *ent->h_addr_list, addrLen
    );
14      this->port = port;
15
16      memset(&rdmaCmId, 0, sizeof(rdmaCmId));
17      memset(&rdmaCmListenId, 0, sizeof(rdmaCmListenId)
    );
18   }
19
20   RdmaCommunicator::RdmaCommunicator(rdma_cm_id *
    rdmaCmId) {
21      this->rdmaCmId = rdmaCmId;
22      preregisteredMr = ktm_rdma_reg_msgs(rdmaCmId,
```

```
    preregisteredBuffer, 1024 * 5);
23  }
```

Listing 13: RdmaCommunicator Constructors

## 8.7 Communicator Factory instantiation of a Communicator

```
1   // Supported unsecure communicators
2   std::vector<std::string> unsecureMatches = {"tcp", "
    http", "oldtcp", "ws", "ib"};
3
4   // Supported secure communicators
5   std::vector<std::string> secureMatches = {"https", "
    wss"};
6
7   // Is the desired communicator a secure communicator?
8   if (not secure) {
9       // No, then search it into the supported unsecure
     communicators vector
10      auto foundIt = std::find(unsecureMatches.begin(),
     unsecureMatches.end(), end->protocol());
11      // Is the desired communicator supported?
12      if (foundIt == unsecureMatches.end()) {
13          // No, then throw an exception.
14          throw std::runtime_error("Unsecure
    communicator not supported");
15      }
16  }
17  else {
18      // Yes, then search it into the supported secure
```

```
    communicators vector
19          auto foundIt = std::find(secureMatches.begin(),
    secureMatches.end(), end->protocol());
20          // Is the desired communicator supported?
21          if (foundIt == secureMatches.end()) {
22              // No, then throw an exception.
23              throw std::runtime_error("Secure communicator
     not supported");
24          }
25      }
26
27      std::string dl_string = gvirtus_home + "/lib/
    libgvirtus-communicators-" + end->protocol() + ".so";
28
29      // dl is a ptr to an LD_Lib<Communicator, *Endpoint>
30      dl = std::make_shared<common::LD_Lib<Communicator,
    std::shared_ptr<Endpoint>>>(dl_string , "
    create_communicator");
31      dl->build_obj(end);
32      return dl;
```

Listing 14: Communicator Factory instantiation of a Communicator

### 8.8 Endpoint Factory instantiation of a Communicator

```
1       // tcp/ip
2       if ("tcp/ip" == j["communicator"][ind_endpoint]["
    endpoint"].at("suite")) {
3           auto end = common::JSON<Endpoint_Tcp>(json_path).
    parser();
4           ptr = std::make_shared<Endpoint_Tcp>(end);
```

```
5       }
6       // infiniband
7       else if ("infiniband-rdma" == j["communicator"][
        ind_endpoint]["endpoint"].at("suite")) {
8           auto end = common::JSON<Endpoint_Rdma>(json_path)
        .parser();
9           ptr = std::make_shared<Endpoint_Rdma>(end);
10      }
11      else {
12          throw "EndpointFactory::get_endpoint(): Your
        suite is not compatible!";
13      }
14
15      ind_endpoint++;
16
17      j.clear();
18      ifs.close();
19
20      return ptr;
```

Listing 15: Endpoint Factory instantiation of an Endpoint

## 8.9 Endpoint Interface

```
1       class Endpoint {
2       public:
3           Endpoint() = default;
4
5           virtual Endpoint &suite(const std::string &suite)
        = 0;
6           virtual inline const std::string &suite() const {
```

```
    return _suite; }
7
8       virtual Endpoint &protocol(const std::string &
    protocol) = 0;
9       virtual inline const std::string &protocol()
    const { return _protocol; }
10
11      virtual inline const std::string to_string()
    const {
12          return _suite + _protocol;
13      };
14
15      inline bool operator==(const Endpoint &endpoint)
    const {
16          return this->to_string() == endpoint.
    to_string();
17      }
18
19    protected:
20      std::string _suite;
21      std::string _protocol;
22 };
```

Listing 16: Endpoint Abstract Class

### 8.10 Endpoint_Rdma Class

```
1   class Endpoint_Rdma : public Endpoint {
2   private:
3       std::string _address;
4       std::uint16_t _port;
```

```
5
6     public:
7         Endpoint_Rdma() = default;
8
9         explicit Endpoint_Rdma(const std::string &
      endp_suite, const std::string &endp_protocol, const
      std::string &endp_address, const std::string &
      endp_port);
10
11        explicit Endpoint_Rdma(const std::string &
      endp_suite): Endpoint_Rdma(endp_suite, "ib", "
      127.0.0.1", "9999") {}
12
13        Endpoint &suite(const std::string &suite)
      override;
14        Endpoint &protocol(const std::string &protocol)
      override;
15
16        Endpoint_Rdma &address(const std::string &address
      );
17        inline const std::string &address() const {
      return _address; }
18
19        Endpoint_Rdma &port(const std::string &port);
20        inline const std::uint16_t &port() const { return
       _port; }
21
22        virtual inline const std::string to_string()
      const {
23            return _suite + _protocol + _address + std::
```

```
    to_string(_port);
24          }
25
26          std::string to_string() {
27              return "EndpointRdma";
28          }
29
30      };
```

Listing 17: Endpoint_Rdma Class

### 8.11 RdmaCommunicator::Serve()

```
1       void RdmaCommunicator::Serve() {
2           // Setup address info
3           rdma_addrinfo hints;
4           memset(&hints, 0, sizeof(hints));
5           hints.ai_port_space = rdma_port_space::RDMA_PS_IB
        ;
6           hints.ai_flags = RAI_PASSIVE;
7
8           rdma_addrinfo * rdmaAddrinfo;
9
10          ktm_rdma_getaddrinfo(hostname, port, &hints, &
        rdmaAddrinfo);
11
12          // Create communication manager id with queue
        pair attributes
13          ibv_qp_init_attr qpInitAttr;
14          memset(&qpInitAttr, 0, sizeof(qpInitAttr));
15
```

```
16          qpInitAttr.cap.max_send_wr = 1;
17          qpInitAttr.cap.max_recv_wr = 1;
18          qpInitAttr.cap.max_send_sge = 1;
19          qpInitAttr.cap.max_recv_sge = 1;
20          qpInitAttr.sq_sig_all = 1;
21          qpInitAttr.qp_type = ibv_qp_type::IBV_QPT_RC;
22
23          ktm_rdma_create_ep(&rdmaCmListenId, rdmaAddrinfo,
        NULL, &qpInitAttr);
24          rdma_freeaddrinfo(rdmaAddrinfo);
25
26          // Listen for connections
27          ktm_rdma_listen(rdmaCmListenId, BACKLOG);
28      }
```

Listing 18: RdmaCommunicator::Serve()

### 8.12 RdmaCommunicator::Accept()

```
1       const gvirtus::communicators::Communicator *const
        RdmaCommunicator::Accept() const {
2       rdma_cm_id * clientRdmaCmId;
3
4       ktm_rdma_get_request(rdmaCmListenId, &clientRdmaCmId)
        ;
5       ktm_rdma_accept(clientRdmaCmId, nullptr);
6
7       auto *ibvQpAttr = static_cast<ibv_qp_attr *>(malloc(
        sizeof(ibv_qp_attr)));
8       ibvQpAttr->min_rnr_timer = 1;
9       if (ibv_modify_qp(clientRdmaCmId->qp, ibvQpAttr,
```

```
    IBV_QP_MIN_RNR_TIMER)) {
10          fprintf(stderr, "ibv_modify_attr() failed: %s\n",
     strerror(errno));
11      }
12
13      return new RdmaCommunicator(clientRdmaCmId);
14  }
```

Listing 19: RdmaCommunicator::Accept()

### 8.13 RdmaCommunicator::Connect()

```
1       void RdmaCommunicator::Connect() {
2       // Setup address info
3       rdma_addrinfo hints;
4       memset(&hints, 0, sizeof(hints));
5       hints.ai_family = AF_INET;
6       hints.ai_port_space = rdma_port_space::RDMA_PS_IB;
7
8       rdma_addrinfo * rdmaAddrinfo;
9
10      char testhost[50] = "192.168.4.102";
11      char testport[50] = "9999";
12      ktm_rdma_getaddrinfo(testhost, testport, &hints, &
    rdmaAddrinfo);
13
14      // Create communication manager id with queue pair
    attributes
15      ibv_qp_init_attr qpInitAttr;
16      memset(&qpInitAttr, 0, sizeof(qpInitAttr));
17
```

```
18      qpInitAttr.cap.max_send_wr = 1;
19      qpInitAttr.cap.max_recv_wr = 1;
20      qpInitAttr.cap.max_send_sge = 1;
21      qpInitAttr.cap.max_recv_sge = 1;
22      qpInitAttr.sq_sig_all = 1;
23      qpInitAttr.qp_type = ibv_qp_type::IBV_QPT_RC;
24
25      ktm_rdma_create_ep(&rdmaCmId, rdmaAddrinfo, nullptr,
        &qpInitAttr);
26      rdma_freeaddrinfo(rdmaAddrinfo);
27
28      ktm_rdma_connect(rdmaCmId, nullptr);
29
30      auto *ibvQpAttr = static_cast<ibv_qp_attr *>(malloc(
        sizeof(ibv_qp_attr)));
31      ibvQpAttr->min_rnr_timer = 1;
32      if (ibv_modify_qp(rdmaCmId->qp, ibvQpAttr,
        IBV_QP_MIN_RNR_TIMER)) {
33          fprintf(stderr, "ibv_modify_attr() failed: %s\n",
         strerror(errno));
34      }
35      preregisteredMr = ktm_rdma_reg_msgs(rdmaCmId,
        preregisteredBuffer, 1024 * 5);
36  }
```

Listing 20: RdmaCommunicator::Connect()

### 8.14 Optimized RdmaCommunicator::Read()

```
1       size_t RdmaCommunicator::Read(char *buffer, size_t
        size) {
```

```
2         if (size < 1024 * 5) {
3             ktm_rdma_post_recv(rdmaCmId, nullptr,
    preregisteredBuffer, size, preregisteredMr);
4         }
5         else {
6             memoryRegion = ktm_rdma_reg_msgs(rdmaCmId,
    buffer, size);
7             ktm_rdma_post_recv(rdmaCmId, nullptr, buffer,
     size, memoryRegion);
8         }
9
10        int num_comp;
11        do num_comp = ibv_poll_cq(rdmaCmId->recv_cq, 1, &
    workCompletion); while (num_comp == 0);
12        if (num_comp < 0) throw "ibv_poll_cq() failed";
13        if (workCompletion.status != IBV_WC_SUCCESS)
    throw 'Failed status " + std::string(ibv_wc_status_str
    (workCompletion.status));
14
15        if (size < 1024 * 5) {
16            memcpy(buffer, preregisteredBuffer, size);
17        }
18
19        return size;
20    }
```

Listing 21: Optimized RdmaCommunicator::Read()

### 8.15 getstring() function

```
1     bool getstring(Communicator *c, string &s) {
```

```
2         if (c->to_string() == "tcpcommunicator") {
3             s = "";
4             char ch = 0;
5             while (c->Read(&ch, 1) == 1) {
6                 if (ch == 0) {
7                     return true;
8                 }
9                 s += ch;
10            }
11            return false;
12        }
13        // newly added rdma communicator string read
      mechanism
14        else if (c->to_string() == "rdmacommunicator") {
15            char * msg = static_cast<char *>(malloc(30));
16            c->Read(msg, 30);
17            s.append(msg);
18            return true;
19        }
20
21        throw "getstring error: unknown communicator...";
22    }
```

Listing 22: Modified getstring() function

### 8.16 Optimized RdmaCommunicator::Write()

```
1     size_t RdmaCommunicator::Write(const char *buffer,
      size_t size) {
2         char * actualBuffer = nullptr;
3
```

```
4         if (size < 1024 * 5) {
5             memcpy(preregisteredBuffer, buffer, size);
6             ktm_rdma_post_recv(rdmaCmId, nullptr,
    preregisteredBuffer, size, preregisteredMr, 0);
7         }
8         else {
9             actualBuffer = (char *) malloc(size);
10            memcpy(actualBuffer, buffer, size);
11            memoryRegion = ktm_rdma_reg_msgs(rdmaCmId,
    actualBuffer, size);
12            ktm_rdma_post_recv(rdmaCmId, nullptr,
    actualBuffer, size, memoryRegion);
13        }
14
15        int num_comp;
16        do num_comp = ibv_poll_cq(rdmaCmId->send_cq, 1, &
    workCompletion); while (num_comp == 0);
17        if (num_comp < 0) throw "ibv_poll_cq() failed";
18        if (workCompletion.status != IBV_WC_SUCCESS)
    throw 'Failed status " + std::string(ibv_wc_status_str
    (workCompletion.status));
19
20        if (size > 1024 * 5) {
21            free(actualBuffer);
22        }
23
24        return size;
25    }
```

Listing 23: Optimized RdmaCommunicator::Write()

## 8.17 nVidia implementation of performance evaluation on Saxpy

```
1   cudaEvent_t start, stop;
2   cudaEventCreate(&start);
3   cudaEventCreate(&stop);
4
5   cudaMemcpy(d_x, x, N*sizeof(float),
    cudaMemcpyHostToDevice);
6   cudaMemcpy(d_y, y, N*sizeof(float),
    cudaMemcpyHostToDevice);
7
8   cudaEventRecord(start);
9   saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
10  cudaEventRecord(stop);
11
12  cudaMemcpy(y, d_y, N*sizeof(float),
    cudaMemcpyDeviceToHost);
13
14  cudaEventSynchronize(stop);
15  float milliseconds = 0;
16  cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 24: nVidia performance evaluation on Saxpy kernel

## 8.18 Updated implementation of performance evaluation on Saxpy

```
1   cudaEvent_t start, stop;
2   cudaEventCreate(&start);
3   cudaEventCreate(&stop);
4
```

```
5       cudaEventRecord(start);
6
7       cudaMemcpy(d_x, x, N*sizeof(float),
        cudaMemcpyHostToDevice);
8       cudaMemcpy(d_y, y, N*sizeof(float),
        cudaMemcpyHostToDevice);
9
10      saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
11
12      cudaMemcpy(y, d_y, N*sizeof(float),
        cudaMemcpyDeviceToHost);
13
14      cudaEventRecord(stop);
15
16      cudaEventSynchronize(stop);
17      float milliseconds = 0;
18      cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 25: Our performance evaluation on Saxpy CUDA Implementation

# Acknowledgements

Every beginning has an end, and my university career is not different. In this work I only spoke in first person plural; It was not an accident, it was a choice. I'm genuinely sure I wouldn't have gotten so far if I was alone, but luckily that's not the case. I'm not really good with words, but I have to say an enormous "thank you" to all the people that believed in me, that supported me in this journey and that pushed me to always be the best version of myself. I know most of you are used to tear-jerking words in thesis aknowledgements, but I'm not the kind to do so. Thanks to my mom, dad and brother: Enza, Lello and Nunzio, that always believed in me, and provided both financial and emotional support, even if they didn't understand a single thing I said when explaining my studies. Thanks to Nonno Nunzio, Nonna Maria, Zia Rossa and Nonno Catello. I hope you're cheering with me from up there. Thanks to all the friends and colleagues I made during my career, in random order (and in hopes of not forgetting anyone...): Aniello, Santo, Denny, Alfredo, Max, Dr.Manu, Nicola, Mario, Simone, Gianfranco, Ilaria, Gigi, Chiara, Maurizio, Alfio, Andrea, Zeno, Francesco, Eugenio, Vincenzo S., Riccardo, Vincenzo D.G. and Alessio. You all were great for one reason or another. Thanks to Prof. Raffaele Montella, Diana, Ciro, and all the people in the High Performance Scientific Computing Smart Lab. If the lab felt like a second home that's because to you. Thanks to Prof. Sokol Kosta and to Peppe Coviello: your technical support was priceless. Thanks to all the professors of the Computer Science course. I learnt a lot in these years because of you. Thanks to Giustino, Susi, Nonna Rosa and Nonno Lello, that believed in me too. Thanks to Salvatore Vozza, for always thinking highly of me, to the point of giving me the nickname "o' scienziat". But now I have to give a special thanks to two special persons. Thank you, Rosa. You're the definition of serendipity. Thanks for always believing in me even when I can't do it myself, thanks for all your teachings, thanks for always being the first person to cheer at my accomplishments, thanks for being the shoulder to cry on when everything feels too heavy, and thanks for loving me like nobody

ever did. But the biggest "thanks" goes to Nonna Angela. If I am the person I am today, is only because of you.

# Ringraziamenti

Ad ogni inizio corrisponde una fine, e la mia carriera universitaria non è da meno. In questa tesi ho parlato solo in prima persona plurale; Non è stato un errore, è stata una scelta. Credo fermamente che non sarei arrivato così lontano se fossi stato da solo, ma fortunatamente non ne è stato il caso. Non sono molto bravo con le parole, però mi sento in dovere di ringraziare enormemente tutte le persone che hanno creduto in me, che mi hanno supportato in questo percorso e che mi hanno spinto ad essere sempre la versione migliore di me stesso. So che molti di voi sono abituati a ringraziamenti strappalacrime, ma non ne sono il tipo. Grazie a mia madre, mio padre e mio fratello: Enza, Lello e Nunzio, che hanno sempre creduto in me, e che mi hanno supportato sia economicamente che emotivamente, anche se non capivano neanche una parola quando parlavo di ciò che studiavo. Grazie a Nonno Nunzio, Nonna Maria, Zia Rossa e Nonno Catello. Spero che stiate gioendo con me da lassù. Un grazie a tutti gli amici e colleghi che ho conosciuto durante la mia carriera, in ordine sparso (nella speranza di non dimenticare nessuno...): Aniello, Santo, Denny, Alfredo, Max, Dr.Manu, Nicola, Mario, Simone, Gianfranco, Ilaria, Gigi, Chiara, Maurizio, Alfio, Andrea, Zeno, Francesco, Eugenio, Vincenzo S., Riccardo, Vincenzo D.G. ed Alessio. Siete stati tutti meravigliosi, in un modo o nell'altro. Grazie al Prof. Raffaele Montella, a Diana, a Ciro ed a tutte le persone dello High Performance Scientific Computing Smart Lab. Se il laboratorio è stato una seconda casa è grazie a voi. Grazie al Prof. Sokol Kosta ed a Peppe Coviello: il vostro supporto tecnico è stato inestimabile. Grazie a tutto il corpo docenti del corso di Informatica. Se ho imparato tanto in questi anni è grazie a voi. Grazie a Giustino, Susi, Nonna Rosa e Nonno Lello, che hanno sempre creduto in me. Grazie a Salvatore Vozza, che ha sempre avuto molta stima di me, al punto da soprannominarmi "o' scienziat". Ma adesso ho due ringraziamenti speciali da fare a due persone altrettanto speciali. Grazie, Rosa. Sei la definizione di serendipità. Grazie per aver sempre creduto in me anche quando non riuscivo a farlo da solo, grazie per tutti i tuoi insegnamenti, grazie per essere sempre la

prima ad esultare per i miei successi, grazie per essere la spalla su cui piangere quando tutto sembra troppo difficile, e grazie per amarmi come nessuno ha mai saputo fare. Ma il grazie più grande va a Nonna Angela. Se sono ciò che sono è grazie a te.

# References


[1] Shuai Che et al. "Accelerating Compute-Intensive Applications with GPUs and FPGAs". In: *2008 Symposium on Application Specific Processors*. 2008, pp. 101–107. DOI: 10.1109/SASP.2008.4570793.

[2] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. en. 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145.

[3] *CUDA Toolkit 11.4 Update 2 Downloads — developer.nvidia.com*. https://developer.nvidia.com/cuda-11-4-2-download-archive.

[4] Giulio Giunta et al. "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds". In: *Euro-Par 2010 - Parallel Processing*. Ed. by Pasqua D'Ambra, Mario Guarracino, and Domenico Talia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 379–391. ISBN: 978-3-642-15277-1.

[5] Raffaele Montella et al. "A General-Purpose Virtualization Service for HPC on Cloud Computing: An Application to GPUs". In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 740–749. ISBN: 978-3-642-31464-3.

[6] *About InfiniBand — infinibandta.org*. https://www.infinibandta.org/about-infiniband/.

[7] *GVirtuS/src/communicators/rdma at master · marianoktm/GVirtuS — github.com*. https://github.com/marianoktm/GVirtuS/tree/master/src/communicators/rdma.

[8] C. Reaño et al. "POSTER: Boosting the performance of remote GPU virtualization using InfiniBand connect-IB and PCIe 3.0". In: *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 2014, pp. 266–267. DOI: 10.1109/CLUSTER.2014.6968737.

[9] Federico Silla et al. “On the benefits of the remote GPU virtualization mechanism: The rCUDA case”. In: *Concurrency and Computation: Practice and Experience* 29.13 (2017). e4072 cpe.4072, e4072. DOI: https://doi.org/10.1002/cpe.4072. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4072. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4072.

[10] A. Kawai et al. “Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability”. In: *FUTURE COMPUTING 2012, The Fourth International Conference on Future Computational Technologies and Applications*. 2012, pp. 7–12.

[11] Lin Shi et al. “vCUDA: GPU accelerated high performance computing in virtual machines”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), pp. 1–11. URL: https://api.semanticscholar.org/CorpusID:1326309.

[12] Vishakha Gupta et al. “GViM: GPU-accelerated virtual machines”. In: *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*. HPCVirt ’09. Nuremburg, Germany: Association for Computing Machinery, 2009, pp. 17–24. ISBN: 9781605584652. DOI: 10.1145/1519138.1519141. URL: https://doi.org/10.1145/1519138.1519141.

[13] Shreya Amit Bhandare. “Designing RDMA-based efficient Communication for GPU Remoting”. Virginia Polytechnic Institute and State University, 2023. URL: https://vtechworks.lib.vt.edu/items/447ddaef-8852-481f-92c3-905943f3f0eb.

[14] *GitHub - pfent/libibverbscpp: C++ bindings for libibverbs — github.com*. https://github.com/pfent/libibverbscpp. [Accessed 21-03-2024].

[15] Antonio Mentone et al. “CUDA Virtualization and Remoting for GPGPU Based Acceleration Offloading at the Edge”. In: *Internet and Distributed Computing Systems*. Ed. by Raffaele Montella et al. Cham: Springer International Publishing, 2019, pp. 414–423. ISBN: 978-3-030-34914-1.

[16] *Unified Memory in CUDA 6 — NVIDIA Technical Blog — developer.nvidia.com.* https://developer.nvidia.com/blog/unified-memory-in-cuda-6/.

[17] Raffaele Montella et al. "On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework". In: *International Journal of Parallel Programming* 45.5 (Oct. 2017), pp. 1142–1163. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0462-1. URL: https://doi.org/10.1007/s10766-016-0462-1.

[18] *C++17 - cppreference.com — en.cppreference.com.* https://en.cppreference.com/w/cpp/17. [Accessed 20-03-2024].

[19] *infiniband/libibverbs.git - A library for direct userspace access to InfiniBand hardware. — git.kernel.org.* https://git.kernel.org/pub/scm/libs/infiniband/libibverbs.git.

[20] *GitHub - ofiwg/librdmacm — github.com.* https://github.com/ofiwg/librdmacm/tree/master.

[21] *GCC, the GNU Compiler Collection - GNU Project — gcc.gnu.org.* https://gcc.gnu.org/. [Accessed 20-03-2024].

[22] *CMake Documentation and Community — cmake.org.* https://cmake.org/documentation/. [Accessed 20-03-2024].

[23] *NVIDIA CUDA Compiler Driver — docs.nvidia.com.* https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[24] *CLion: A Cross-Platform IDE for C and C++ by JetBrains — jetbrains.com.* https://www.jetbrains.com/clion/.

[25] Dotan Barak. *Tips and tricks to optimize your RDMA code - RDMAmojo — rdmamojo.com.* https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/.

[26] Philip Werner Frey and Gustavo Alonso. "Minimizing the Hidden Cost of RDMA". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. 2009, pp. 553–560. DOI: 10.1109/ICDCS.2009.32.

[27] *Guida all'uso — rcf.uniparthenope.it.* https://rcf.uniparthenope.it/it/userguide/.

[28] *Six Ways to SAXPY — NVIDIA Technical Blog — developer.nvidia.com.* https://developer.nvidia.com/blog/six-ways-saxpy/. [Accessed 24-03-2024].

[29] *GitHub - marianoktm/GVirtuS: A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. — github.com.* https://github.com/marianoktm/GVirtuS/.

[30] *How to Implement Performance Metrics in CUDA C/C++ — NVIDIA Technical Blog — developer.nvidia.com.* https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/.