Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Performance analysis of GPU virtualization in low-power systems

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Alejandro Rodriguez Segrelles

Tutor: Federico Silla
Carlos Reaño González

Course 2016-2017

Resum

El consum d'energia és una gran preocupació en els centres de dades. Els sistemes moderns intenten aconseguir el millor rendiment possible al preu d'un alt consum d'energia, elevats costos d'adquisició i manteniment, i amb una gran quantitat d'espai per a emmagatzemar-ho. En sistemes basats en GPUs, la GPU està a càrrec de la càrrega de treball de càlcul intens, mentres que la CPU estaria fent la resta de les tasques. Açò podria proporcionar la possibilitat d'usar una CPU de baix consum sense afectar massa el rendiment general. Si s'utilitzara la virtualització de GPU en un sistema, les GPUs es col·locarien en només uns pocs nodes, creant una part localitzada amb un diferent nivell de consum d'energia i calfament. Açò simplificaria les tasques de manteniment, centralitzant la complexitat en només alguns nodes en compte de distribuir-los a nivell global en el centre de dades.

En aquest treball provem les capacitats de les CPUs de baix consum a l'hora d'executar aplicacions CUDA a través de la virtualització de GPU. Els resultats demostren que els processadors de baix consum no són necessàriament menys potents en termes de rendiment brut, però altres factors importants en la seua configuració poden influir en gran manera en el resultat, com la velocitat de la memòria o els carrils PCIe de la targeta de xarxa.

Paraules clau: GPU, rCUDA, virtualizació, CUDA, baix consum, CPU

Resumen

El consumo de energía es una gran preocupación en los centros de datos. Los sistemas modernos intentan lograr el mejor rendimientoposible al precio de un alto consumo de energía, elevados costes de adquisición y de mantenimiento, y necesitando una gran cantidad de espacio para ser almacenados. En sistemas basados en GPUs, la GPU está a cargo de la carga de trabajo de cálculo intenso, mientras que la CPU estaría haciendo el resto de las tareas. Esto podría proporcionar la posibilidad de usar una CPU de bajo consumo sin afectar demasiado el rendimiento general. Si se utilizara la virtualización de GPU en un sistema, las GPUs se colocarían en solo unos pocos nodos, creando una parte localizada con un diferente nivel de consumo de energía y temperatura. Esto simplificaría las tareas de mantenimiento, centralizando la complejidad en solo algunos nodos en lugar de distribuirlos a nivel global en el centro de datos.

En este trabajo probamos las capacidades de las CPUs de bajo consumo al ejecutar aplicaciones CUDA a través de la virtualización de GPU. Los resultados demuestran que los procesadores de bajo consumo no son necesariamente menos potentes en términos de rendimiento bruto, pero otros factores importantes en su configuración pueden influir en gran medida en el resultado, como la velocidad de la memoria o los carriles PCIe de la tarjeta de red.

Palabras clave: GPU, rCUDA, virtualización, CUDA, bajo consumo, CPU

Abstract

Energy consumption is a big concern in data centres. Modern systems try to achieve the best throughput at the price of high energy consumption, high acquisition and maintenance costs and taking a large amount of space to be stored. In GPU-based systems, the GPU is in charge of the compute-intense workload, while the CPU would be doing the rest of the tasks. This might provide the possibility to use low-power CPUs without

affecting too much the overall performance. If GPU virtualization was to be used in a system, the GPUs would all be placed in just a few nodes, creating a localized part with a different level of energy consumption and heating. This would simplify the maintenance tasks by centralizing the complexity in only some nodes instead of distributing it globally in the data centre.

In this work, we tested the capabilities of low-power CPUs when executing CUDA applications through GPU virtualization. The results demonstrate that low-power processors are not necessarily less powerful in terms of raw performance, but other important factors in their configuration might greatly influence the outcome, like memory speed or the PCIe lanes from the network card.

Key words: GPU, rCUDA, virtualization, CUDA, low-power, CPU

Contents

Contents v
List of Figures vii
List of Tables vii

1 Introduction 1

2 Ecosystem description 9
2.1 Central processing unit . . . 9
2.1.1 Intel Xeon CPU E5-2620 v2 . . . 10
2.1.2 Intel Atom CPU C2750 . . . 10
2.1.3 Intel Xeon CPU D-1541 . . . 11
2.1.4 CAVIUM ThunderX . . . 11
2.2 Graphics Processing Unit . . . 11
2.2.1 NVIDIA Tesla K20m . . . 12
2.3 CUDA . . . 14
2.4 Remote CUDA . . . 16
2.5 High-performance networks . . . 17
2.5.1 InfiniBand . . . 17
2.5.2 RDMA over Converged Ethernet . . . 18

3 Applications 19
3.1 CUDA samples: Bandwidth test . . . 19
3.2 LAMMPS . . . 21
3.3 CUDA-MEME . . . 22
3.4 NAMD . . . 23

4 Performance analysis 25
4.1 System configuration . . . 25
4.1.1 Setting up rCUDA . . . 28
4.2 InfiniBand bandwidth test . . . 30
4.2.1 Description . . . 30
4.2.2 Results . . . 31
4.3 Testing bandwidth of CUDA and rCUDA . . . 34
4.3.1 Description . . . 34
4.3.2 Results . . . 35
4.4 LAMMPS . . . 41
4.4.1 Installation . . . 41
4.4.2 Execution . . . 42
4.4.3 Performance results . . . 43
4.5 CUDA-MEME . . . 45
4.5.1 Installation . . . 45
4.5.2 Execution . . . 47
4.5.3 Performance results . . . 48
4.6 NAMD . . . 49
4.6.1 Installation . . . 49

4.6.2 Execution 51
4.6.3 Performance results 52
5 Conclusions **55**
Bibliography **57**

List of Figures

1.1 Performance comparison between CPU and GPU 1
1.2 GPU utilization during a LAMMPS execution 4
1.3 GPU utilization during a GPU-BLAST execution 4
1.4 GPU utilization during a CUDASW execution 5
1.5 GPU utilization during a CUDA-MEME execution 5

2.1 Accelerator and co-processor system share from the TOP500 list 13
2.2 CUDA unified memory on Kepler and Pascal microarchitectures 15
2.3 Visual representation of rCUDA architecture 16

3.1 CUDA Toolkit sample N-Body render 19
3.2 LAMMPS logo animation as an example of molecular dynamics 21
3.3 DNA sequence motif 23
3.4 NAMD VMD render 24

4.1 InfiniBand RDMA *write* bandwidth test results 33
4.2 InfiniBand RDMA *read* bandwidth test results 33
4.3 InfiniBand RDMA *send* bandwidth test results 34
4.4 CUDA bandwidth test results from host to device with pinned memory . 39
4.5 CUDA bandwidth test results from device to host with pinned memory . 40
4.6 CUDA bandwidth test results from host to device with pageable memory 40
4.7 CUDA bandwidth test results from device to host with pageable memory 41
4.8 LAMMPS CHAIN benchmark results 43
4.9 LAMMPS LJ benchmark results 44
4.10 LAMMPS EAM benchmark results 44
4.11 CUDA-MEME NRSF-500 benchmark results 48
4.12 CUDA-MEME NRSF-1000 benchmark results 48
4.13 CUDA-MEME NRSF-2000 benchmark results 49
4.14 NAMD ApoA1 benchmark results 52
4.15 NAMD ATPase benchmark results 52
4.16 NAMD STMV benchmark results 53

List of Tables

1.1 Top ten supercomputers from TOP500 2
1.2 Top ten supercomputers from Green500 3

2.1 InfiniBand data rates 18

4.1 Specification for the nodes mlxm1 and mlxm2 26
4.2 Specification for the node mlxd1 26
4.3 Specification for the node mlxa1 26
4.4 Specification for the node mlxarm1 27

CHAPTER 1
Introduction

Traditionally, graphics processing units (GPUs) were used primarily for 3D game rendering. However, their capabilities are being harnessed to accelerate computational workloads in different fields such as data analysis (Big Data) [1], computational fluid dynamics [2], chemical physics [3], computational algebra [4], image analysis [5], finance [6], biology [7], and artificial intelligence [8], among others. The use of GPUs for general-purpose computing began over one decade ago. While in 2003 the arrival of floating-point colour buffers meant a more realistic gameplay experience in the game environment, for the scientific community it was the solution to the overflow associated with the fixed-point arithmetic. A bit later, the fixed graphics pipeline that was common in GPUs was improved to a freely programmable one. The release of NVIDIA's CUDA (Compute Unified Device Architecture) in 2007 established a turning point for GPU programming because the old graphics-specific terms became obsolete to be replaced by more programming-generic terms like threads, vector processing and so on. Those and further improvements, like supporting Error-Correction Codes (ECC) memory, make the GPU computing even more fit for scientific use [9].

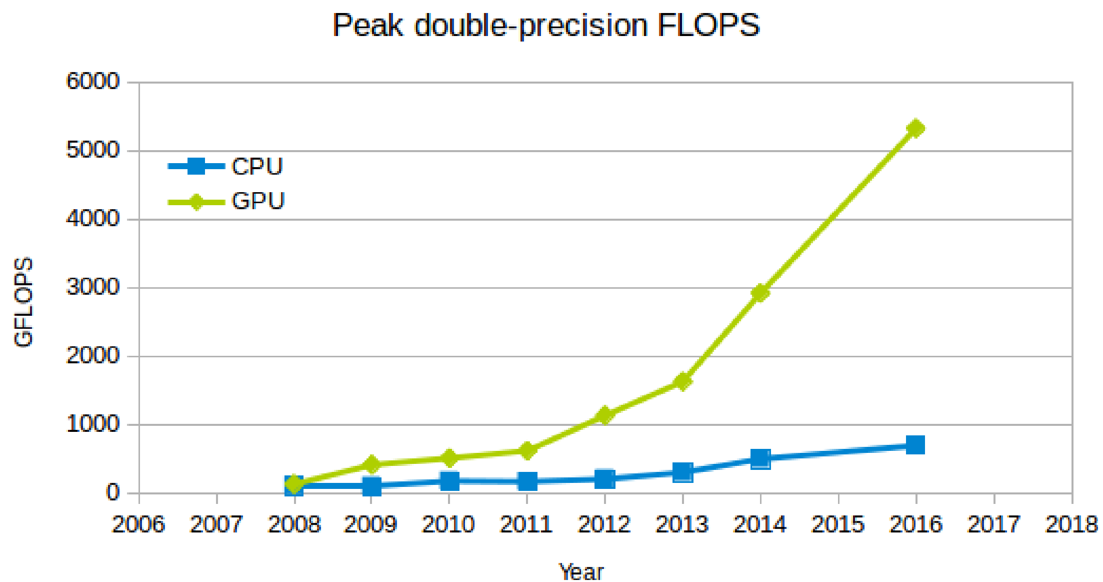

Figure 1.1: Performance comparison between CPU and GPU [10]. Higher is better.

The central processing unit (CPU) is usually composed of several cores with a large cache memory that can handle a few software threads at a time. On the contrary, the GPU

is composed of hundreds or thousands of cores operating at lower frequencies and is capable of handling thousands of threads simultaneously. This makes them well suited for acceleration of highly parallelizable applications. Therefore, a sole GPU has the ability to accelerate a program by 10x or even 100x over a single CPU. Hence, in the computing field, a GPU might easily be more power and cost efficient than a CPU.

Thanks to their architecture, GPUs excel at performing floating point operations when compared to a CPU. Integer operations can be solved easily with a simple hardware implementation. Therefore it is a trivial task and it is highly unlikely that it will be the cause of a bottleneck. On the contrary, operations with floating point numbers are very expensive given that they use the necessary decimal precision which cannot be ignored. In the majority of cases, the floating point numbers are implemented following the IEEE 754 standard, which stores the number as a sign, exponent and mantissa. This structure allows for the representation of very large and small numbers. This comes at the cost of performance. Even though they are so costly, GPUs have vast amounts of floating point units (FPU), allowing for high parallelization when operating with floating point numbers, while CPUs have just a few of them.

In Figure 1.1 it can be seen roughly the difference between CPUs and GPUs when operating with floating point numbers and how much the former improved lately. Because floating point numbers are necessary for many fields, it is important to have systems able to compute them fast and efficiently. Every system administrator has to find the appropriate configuration for the new system which will fulfil the prompted requirements of the users of the data centre. In this regards, some are more inclined to use many-core CPUs, some of them prefer GPUs, some administrators try to reach the highest possible performance by using not the best processors, some just order custom-built processors, etc. High-performance computing (HPC) has benefited many industries and the needs of these industries greatly influence the specific system requirements.

Rank	System		Power Efficiency (GFlops/Watt)
1	Sunway TaihuLight	**CPU** Sunway SW26010 260C 1.45GHz	6.051
2	Tianhe-2 (MilkyWay-2)	**CPU** Xeon E5-2692 12C 2.200GHz **Co-processor** Xeon Phi 31S1P	1.902
3	Piz Daint	**CPU** Xeon E5-2690v3 12C 2.6GHz **GPU** NVIDIA Tesla P100	10.398
4	Titan	**CPU** Opteron 6274 16C 2.200GHz **GPU** NVIDIA Tesla K20x	2.143
5	Sequoia	**CPU** Power BQC 16C 1.60 GHz	2.177
6	Cori	**CPU** Xeon Phi 7250 68C 1.4GHz	3.558
7	Oakforest-PACS	**CPU** Xeon Phi 7250 68C 1.4GHz	4.986
8	K computer	**CPU** SPARC64 VIIIfx 2.0GHz	0.830
9	Mira	**CPU** Power BQC 16C 1.60GHz	2.177
10	Trinity	**CPU** Xeon E5-2698v3 16C 2.3GHz	1.914

Table 1.1: Top ten supercomputers simplified list from TOP500 [11] sorted by maximum achieved performance (FLOPS). System information has been reduced to just processing units, other significant performance factors have been omitted for simplicity.

Rank	System		Power Efficiency (GFlops/Watt)
1	TSUBAME3.0	**CPU** Xeon E5-2680v4 14C 2.4GHz **GPU** NVIDIA Tesla P100	14.110
2	kukai	**CPU** Xeon E5-2650Lv4 14C 1.7GHz **GPU** NVIDIA Tesla P100	14.046
3	AIST AI Cloud	**CPU** Xeon E5-2630Lv4 10C 1.8GHz **GPU** NVIDIA Tesla P100	12.681
4	RAIDEN GPU subsystem	**CPU** Xeon E5-2698v4 20C 2.2GHz **GPU** NVIDIA Tesla P100	10.603
5	Wilkes-2	**CPU** Xeon E5-2650v4 12C 2.2GHz **GPU** NVIDIA Tesla P100	10.428
6	Piz Daint	**CPU** Xeon E5-2690v3 12C 2.6GHz **GPU** NVIDIA Tesla P100	10.398
7	Gyoukou	**CPU** Xeon D-1571 16C 1.3GHz PEZY-SC2	10.226
8	Research Computation Facility for GOSAT-2 (RCF2)	**CPU** Xeon E5-2650v4 12C 2.2GHz **GPU** NVIDIA Tesla P100	9.797
9	Unnamed (Facebook)	**CPU** Xeon E5-2698v4 20C 2.2GHz Xeon E5-2650v4 12C 2.2GHz **GPU** NVIDIA Tesla P100	9.462
10	DGX Saturn V	**CPU** Xeon E5-2698v4 20C 2.2GHz **GPU** NVIDIA Tesla P100	9.462

Table 1.2: Top ten supercomputers simplified list from Green500 [12] sorted by power efficiency (FLOPS/W). System information has been reduced to just processing units, other significant performance factors have been omitted for simplicity.

As seen in Table 1.2, GPU system configurations are almost mandatory to achieve excellent power efficiency. From the Green500 top ten, nine of the systems are accelerated by GPUs. In the TOP500 top ten list (see Table 1.1) there are only three systems that are not only CPU-based, and from those three, one uses co-processors and the other two GPUs.

In the TOP500 list, the system in the first place, Sunway TaihuLight, is undoubtedly powerful. It is able to outperform, in terms of throughput, by almost three times the Tianhe-2 (placed second) and by almost five times the Piz Daint (placed third). This might sound remarkable when out of context, but are there any drawbacks from having such a powerful system?

The Sunway TaihuLight is accelerated by 40,960 260-core Sunway SW26010 many-core processors. It is the third most power-consuming system on the list with 15 MW. If the price per kWh was 0,10€, just to put an example, the organisation maintaining it (National Supercomputing Center in Wuxi) would be spending over ten million euros a month just for having the system switched on. This bill would be far from the monthly cost,

as this is just the electricity expenses, not counting any maintenance, water or gas costs. However, the Sunway Taihulight is not recklessly spending so much energy to achieve its high performance. It is ranked as the seventeenth most energy-efficient system, meaning that regardless of the performance goals, energy optimization was a high priority on its design. Another interesting system is the Piz Daint. The Swiss supercomputer is ranked third most powerful and sixth most energy-efficient, almost doubling the energy efficiency of the Sunway TaihuLight supercomputer but at the cost of achieving only one fifth of its throughput. The Piz Daint is accelerated by the Tesla P100, the latest general-purpose GPU by NVIDIA, which provides energy efficiency of 18.8 GFLOPs/W in double precision [13]. In this configuration, the GPUs are in charge of all the parallelizable compute-intense workloads, while the CPU would be doing the rest.

The previous paragraphs show the theory of using GPUs, but in some cases, it might be far from reality. Notice the TOP500 and Green500 lists are created by leveraging the LINPACK benchmark [14], which keeps the CPU and GPU busy 100% of the time during its execution. GPU's utilization at 100% would be the perfect case scenario. However, most applications are not able to keep GPUs busy 100% of the time. In this regard, it is not so common to assign the GPU to an application, but the application is not able to keep the GPU busy 100% of its execution time, thus reducing, in practice, GPU utilization. This is what happens, for instance, with the LAMMPS [15], CUDASW [16] and GPU-BLAST [17] applications, as shown in Figures 1.2, 1.3 and 1.4, respectively. These figures depict the GPU utilization as well as the utilization of GPU memory along the execution of these applications. It can be seen in the figures that GPU is not busy during the entire execution of the applications but it is idle during a significant amount of time. In a similar way, Figure 1.5 shows the utilization of the GPU during the execution of the CUDA-MEME [18] application. In this case, the GPU is busy during almost the whole execution of the application. However, GPU utilization is never higher than 50%.

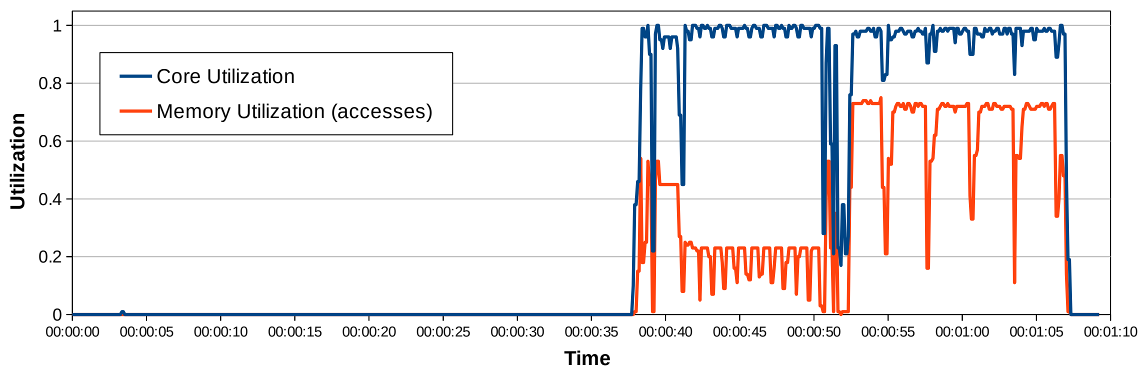

Figure 1.2: GPU utilization during a LAMMPS execution. Courtesy of Javier Prades.

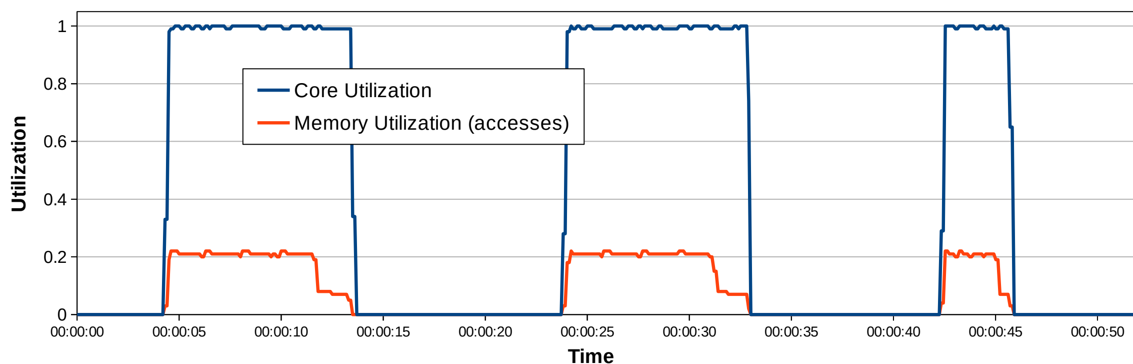

Figure 1.3: GPU utilization during a GPU-BLAST execution. Courtesy of Javier Prades.

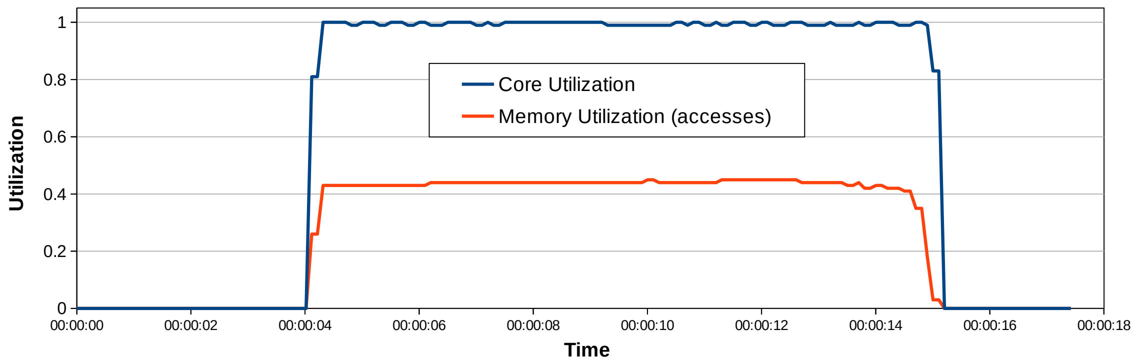

Figure 1.4: GPU utilization during a CUDASW execution. Courtesy of Javier Prades.

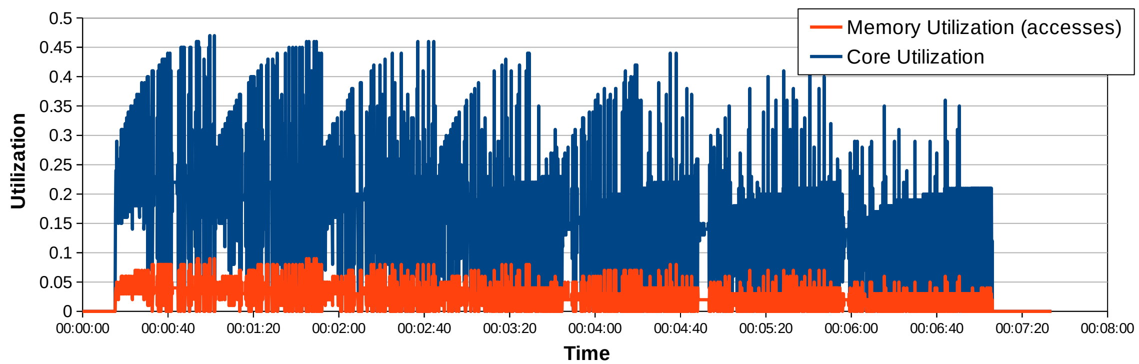

Figure 1.5: GPU utilization during a CUDA-MEME execution. Courtesy of Javier Prades.

Despite the relatively low GPU utilization for most real workloads, many data centres still include GPUs in their configurations. This is a common approach in this field when trying to reduce energy consumption. Nevertheless, given that on the several computing improvements that the GPUs have received over time, low GPU utilization is still a concern, GPU virtualization appears to be an appealing option when trying to give a twist to the average GPU-focused data centre [19]. GPU virtualization enables the access to GPUs that are physically located in a different cluster node. This creates a new layer of abstraction that can potentially increase the GPU utilization by letting the GPUs make computations requested by any node in the cluster. One request does not lock the GPU's resources, therefore it is able to serve several requests in different threads to use the most free resources, which can increase the GPU utilization even more.

The *Parallel Architectures Group* (GAP) from *Universitat Politècnica de Valencia* has developed rCUDA [20] [21], which is a middleware framework that removes a relevant restriction that any data centre might have: a cluster node can only use the GPUs directly connected to it. This is achieved by defining server nodes, the ones that must have GPUs and that will compute when requested, and client nodes, the ones that will emulate being in possession of GPUs and will execute the application and distribute the GPU tasks between the servers. The main purpose of this project is to abstract the GPU-computing task scope from the same node to the whole data centre. Additionally, the rCUDA middleware is the most powerful and modern GPU virtualization solution as explained in [22].

Once GPUs are decoupled from nodes, thus being used from any node in the cluster thanks to a remote GPU virtualization framework such as rCUDA, it is possible to think about further improving energy efficiency, which is a big concern nowadays given that data centres represent the 1.3% of worldwide energy consumption and is predicted to become an 8% in 2020 [23]. In this regard, an interesting system configuration in the

context of GPUs is using low-power processors. The main idea is the following one: if the CPU is used mainly for the serial tasks and the heavy workload is offloaded to the GPU, it might not be necessary to have the most powerful CPUs but low energy-consumption ones could be a better choice and afterwards invest in more or better GPUs. There are several product families whose main focus is low-power. A fair known example is the Intel Atom processor [24], which is Intel's ultra-low-voltage product family, that features low power and performance but is able to attend concurrent tasks competently. Another interesting low-power processor is the Intel Xeon D [25] processor, which provides higher performance than the Intel Atom processor at the cost of a slightly higher power budget. The ARM [26] processor is another example of low-power processors.

By combining the use of remove GPU utilization with that of low-power processors it is possible to devise a system composed of nodes populated with low-power processors that offload the compute-intense GPU parts of their code to remote GPU servers where powerful GPUs are installed. In this way, the utilization of GPUs would be noticeably increased because they would be concurrently shared among several applications and, on the CPU side, the amount of energy required by processors during the execution of the application would be expected to be reduced with respect to the use of traditional processors.

This is actually the leitmotiv of this work: analyzing the use of low-power processors along with remote GPU utilization in order to execute CUDA accelerated applications. Notice, however, that in this work we only focus on the performance side of this analysis. In order to complete the study, it would be required to analyze also power consumption. However, this latter analysis is out of the scope of this thesis due to the limitation in hours associated to these theses.

In this thesis, we have used the rCUDA framework along with different low-power processors in order to study the execution time of several scientific applications. More precisely, we have used rCUDA to execute the LAMMPS, NAMD and CUDA-MEME applications in nodes with Intel Xeon, Intel Xeon D, Intel Atom and 64-bit ARM processors in order to assess the feasibility of the idea described above.

Notice that although this work seems to be purely research work, it is very helpful for a student of the computer engineering degree given that the student will acquire many different skills during the completion of this work. Among the many benefits for the student we can enumerate the following ones:

1. The student is introduced to the most recent CPU and GPU technologies and he is provided with the opportunity to test them in a real cluster. This knowledge will for sure be useful in his future professional activities.

2. The student is introduced to the high-performance InfiniBand network fabric, thus improving his knowledge about current top-performance interconnects. This kind of high-performance networks, as well as other RDMA based ones, are used in many current data centres.

3. The student is introduced to several scientific applications commonly used in many data centres. In this regard, the student will have to install and execute them in several processor architectures. The experience acquired during this stage of the thesis will be useful for future professional activities of the student in the case he works in a job related to system administration.

4. The student is introduced to the tasks related to performance measurement of applications, which will be useful in many different aspects of his future professional

life. Furthermore, the task of analysing performance results will also be useful for the student.

The rest of this thesis is structured as follows: in Chapter 2, titled "Ecosystem description", we go into detail about all the hardware and software related to the environment we put to test; in Chapter 3, titled "Applications", we describe the applications that have been used for the testing process; in Chapter 4, titled "Performance analysis", we have explained how the tests work and in what they consist of before analyzing the results; finally, in Chapter 5, titled "Conclusions", we discuss about the big picture of the previously mentioned test results and the associated conclusions.

CHAPTER 2
Ecosystem description

In this chapter, we focus on describing the hardware and software that conform the different systems used in this work. The few elements described are just some essential parts of the ecosystem that we considered related to the subject scope.

2.1 Central processing unit

A *central processing unit* (CPU) is the electronic circuit within a computer capable of performing basic arithmetic, logical, control and input/output operations when specified by the instructions given [27]. The main components are:

- Arithmetic logic unit (ALU): it performs the arithmetic and logic operations.
- Processor registers: make possible for the ALU to retrieve stored operands and also to store the result of ALU operations.
- Control unit (CU): Directs the process of fetching the instructions from memory and directing their execution within the rest of components.

Nowadays, CPUs are microprocessors. Hence, the whole CPU is contained in a single integrated circuit (IC). In the context of ICs, this particular one containing the CPU is named *CPU socket*. Modern microprocessors often integrate two or more CPUs in the same IC.

Every CPU inside an IC is referred as a *core*, more specifically a *physical core*. Sometimes, a physical core is capable of emulating several cores to share the workload whenever possible. These are *virtual cores*, and even that they are not capable of performing real parallel computing, they might increase the overall performance by interweaving tasks within the physical core. The conjunction of both physical and virtual cores is known as *logical cores*.

A CPU that has more than one physical core is commonly referred to as a *multi-core processor*. The main benefit of multi-core processors compared to single-core processors is the ability to multitask, that is, performing parallel computing: multi-core processors can execute concurrently as many tasks as physical cores it has; whilst single-core processors, even having the computing power to switch swiftly between tasks, will execute them sequentially.

In the next subsections, we introduce the CPUs that have been used for this work.

2.1.1 Intel Xeon CPU E5-2620 v2

Intel Xeon is the CPU family from Intel Corporation that targets non-consumer workstations and servers, focusing on high-bandwidth, large-memory and highly concurrent workloads [28]. The first generation of Intel Xeon was released in 1998. The main features of Intel Xeon processors are:

- Great support for systems with more than one CPU socket installed (multi-socket).
- High amount of cores per socket. It is also common to have two threads per core.
- Large cache memory.
- Support for ECC memory, which helps significantly to prevent data corruption.

Since 2010, Intel Xeon CPUs are split into three categories: E3, E5 and E7, for the low, mid and high-end tiers respectively.

We have used in this study the Intel Xeon E5-2620 v2, a model from late 2013 [29]. It has six cores and twelve threads working at a frequency of 2.1 GHz and a maximum of 2.6 GHz when Intel Turbo is activated. It has 15 MB of cache and benefits from the SmartCache technology from Intel, which provides the ability to dynamically share access to the last level cache between cores. This CPU itself can hold up to 768 GB of RAM in four memory channels, which must be DDR3 and have a frequency between 800 and 1600 MHz. The maximum memory bandwidth it can sustain is 51.2 GB/s. It dissipates 80 W when operating at its base frequency. It is worth mentioning that it supports hardware virtualization (VT-x and VT-d) and Hyper-Threading technologies. This particular model works on a system configuration with up to two CPUs. The recommended price for this CPU is $408.00.

2.1.2 Intel Atom CPU C2750

Intel Atom is the ultra-low-voltage CPU family from Intel Corporation [24]. The first generation of Intel Atom CPUs was released in 2008. These CPUs target portable and embedded systems, where low consumption and long battery life are preferred. Intel Atom's CPU power usage can go as low as 0.65 W (Intel Atom Z500 [30]), but it is usually between 2 and 20 W.

The drawback from ultra-low-voltage CPUs is its low performance. They usually have a subpar amount of cores and their clock rate is reduced on purpose, translating into a lower voltage, power usage and processor frequency.

Some of the Intel Atom's processors are specifically designed for servers and storage. They can be identified by its series' name starting with an S or a C followed by four numbers. This subfamily is not planned to be sold separately as a normal CPU, but it will be part of a whole *system on chip* (SoC) that will integrate all the components required in a computer. Therefore, their availability is limited to manufacturers. It comes with on-die GPU, compatibility with the x86-64 Intel instruction set (Intel 64) and with Intel's technology for hardware virtualization (VT-x).

We used in this thesis the Intel Atom C2750, which was released in September 2013 [31]. This CPU is not compatible with the Hyper-threading technology from Intel. Therefore, the parallelization is not optimized by hardware. However, it is an octa-core processor that will operate at 2.40 GHz while only dissipating 20 W. Under heavy workloads, the Intel Turbo technology will boost frequency up to 2.60 GHz. The cache memory is only 4 MB. Supports up to 64 GB of RAM and only supports DDR3/DDR3L at 1600 MHz, with

a maximum memory bandwidth of 25.6 GB/s. The recommended price for this CPU is $171.00.

2.1.3 Intel Xeon CPU D-1541

The Intel Xeon D processor is part of the Xeon family that is in middle ground between the Intel Xeon E3 and Intel Atom families since the latter does not offer enough performance and could easily become a bottleneck. The Xeon D offers higher performance than the Intel Atom while using less power than the Intel Xeon E3. Hence, the Xeon D focuses on efficiency in dense, lower-power, lightweight hyperscale workloads. The first and only generation was released in March 2015. Its design is based on the Intel Atom, therefore it is another SoC.

In this study, we used the Intel Xeon D-1541 [32]. It is an octa-core processor with sixteen threads working at a frequency of 2.1 GHz and up to 2.7 GHz thanks to the Intel Turbo technology. It has 12 MB of cache memory and supports up to 128 GB of RAM in two memory channels, either DDR3L up to 1600 MHz or DDR4L up to 2133 MHz. It dissipates 45 W when operating at its base frequency. This CPU does not support multi-socket configurations but supports Hyper-Threading and hardware virtualization (VT-x and VT-d) technologies. The recommended price for this CPU is $581.00.

2.1.4 CAVIUM ThunderX

The CAVIUM ThunderX product family consists of different 64-bit ARMv8 processors that focus on computing, storage and servers [26]. It is also another SoC, but this time they are available in both single and dual socket configurations. The first generation of the CAVIUM ThunderX processor was released in 2015.

This processor consists of forty-eight ARMv8-A cores and ninety-six threads with up to 2.5 GHz frequency. Its cache is 16 MB and it is shared between the cores. This processor can hold up to 1 TB of memory in a dual socket configuration in four memory channels, either DDR3 or DDR4 up to 2400 MHz. It is estimated to dissipate around 80 W [33] when at maximum frequency, whilst just 50 W at 2.0 GHz. It is worth to remark that the instruction sets of Intel and ARM processors are not compatible and therefore the applications used in this study will be compiled for both processor architectures.

2.2 Graphics Processing Unit

A *graphics processing unit* (GPU) is an electronic circuit specialized in generating images in a frame buffer and sending them to an output display [34]. Usually, the GPU is either embedded on the motherboard, is present on the CPU die or is a graphics card that is connected to the motherboard.

Modern GPUs are efficient and far superior to general-purpose CPUs when operating large blocks of data in parallel. There exist different types of GPUs:

Dedicated graphics card The most powerful GPUs. They are usually connected to the motherboard by an expansion slot (currently PCI Express), although it is not necessarily removable or connected by this standard. They have their own dedicated memory named graphics double data rate synchronous dynamic random-access memory (GDDR SDRAM), independent from the system memory, which is specialized for graphic processing work-

loads. There is the possibility of using multiple dedicated GPUs simultaneously thanks to the SLI technology by NVIDIA and CrossFire by AMD.

Integrated graphics These GPUs have small to no dedicated memory, therefore they have to use a portion of the system's memory. Graphical tasks are often memory intensive, so sharing part of the system's memory bandwidth will have a negative impact on its overall performance. Integrated graphics processors can be integrated onto the motherboard, as part of the chipset, or onto the CPU die.

Hybrid graphics processing The newest type, competing with integrated graphics in the low-end computer market. They share the memory with the system and have a small dedicated memory cache, to compensate for the high latency of the system's RAM.

General-purpose GPU (GPGPU) These GPUs focus on performing computations usually handled by a CPU. It is common to have them running compute kernels, which transform the high computational power of a GPU shader pipeline into general-purpose computing power. There are specific API extensions for C programming like CUDA (only compatible with NVIDIA) or OpenCL (for any GPU, CPU and more devices) that make the programs capable of using the GPU to operate large buffers when necessary. Usually, they lack the ability to output to a display, despite support technologies like OpenGL or DirectX.

External GPU (eGPU) A GPU located outside of the computer box. As laptops usually have low graphics processing capabilities, connecting an eGPU might give them enough power to carry graphic intensive tasks. This is an unpopular type of GPU. It has little official support from GPU manufacturers and is only supported by certain computers since they need to have a specific port to connect the eGPU properly (ExpressCard, mPCIe or, preferably, Thunderbolt).

2.2.1 NVIDIA Tesla K20m

Tesla is the GPGPU product family from NVIDIA. They are mainly used for [35]:

- Simulations and large scale calculations (computational chemistry, molecular dynamics, computational fluid dynamics...).
- High-end image generation from scientific data (magnetic resonance imaging, computed tomography...).
- Analysis of extensive data (computational finance, numerical analytics, machine learning...).

We have used the NVIDIA Tesla K20m, a GPU manufactured in a Kepler microarchitecture, which is the semiconductor device fabrication process and is usually referred to as the size of one transistor from the integrated circuit with that manufacturing process. For Kepler, it means 28 nm. It has 2496 CUDA cores and a base clock of 706 MHz. It has 5 GB GDDR5 of memory and its bandwidth is 208 GB/s. It is able to process up to 3.5 TFLOPS when operating with single-precision floating-point numbers and 1.1 TFLOPS when operating with double-precision floating-point numbers. It dissipates 225 W [36].

Although we have used only this model in this work, other GPU models are also available for general-purpose GPU computing. For instance, NVIDIA Tesla is a wide

family of general-purpose GPUs. Before the Kepler microarchitecture, the Tesla GPGPUs were first manufactured in the Tesla microarchitecture (90 nm, 80 nm, 65 nm, 55 nm and 40 nm) from May 2007. The GPGPUs from this series were split into different categories depending on its designated purpose: computing server (models S870, S1070 and S1075), computing module (models C870 and C1060), deskside computer (model D870) and visual computing (Quadro Plex 2200 D2 and Quadro Plex 2200 S4).

Afterwards, from April 2010, they started to be manufactured in Fermi microarchitecture (40 nm and 28 nm). This series followed a similar naming system to the previous microarchitecture, splitting the GPGPUs into categories: computing server (models S2050 and S2070), computing module (models C2050, C2070, C2075, M2070 and M2090).

The Kepler microarchitecture (28 nm) was released in April 2012, more than five years from now, meaning that it could be outdated for today's problems and many new GPUs can easily outperform it. However, for this work, we do not need the very latest GPUs, and additionally, the resources in the university are not usually the latest models. Besides the NVIDIA Tesla K20m that we used for this work, there are more Kepler GPUs, this time referred to as *accelerators* and not split into categories. The NVIDIA GPGPUs with this manufacturing process are the Tesla K10, Tesla K20, Tesla K20x, Tesla K40 and Tesla K80. Notice that the models are preceded by the microarchitecture's initial. This naming rule is still active to this day. It is worth mentioning that Titan, the system number four in the TOP500 (shown in Figure 1.1), uses Kepler GPUs (NVIDIA Tesla K20x). Also, many other systems are still based on different Kepler GPUs, as seen in Figure 2.1.

The Maxwell microarchitecture (28 nm) is the Kepler successor, released in February 2014. Five different NVIDIA GPGPUs were made with this manufacturing process: Tesla M4, Tesla M6, Tesla M10, Tesla M40 and Tesla M60.

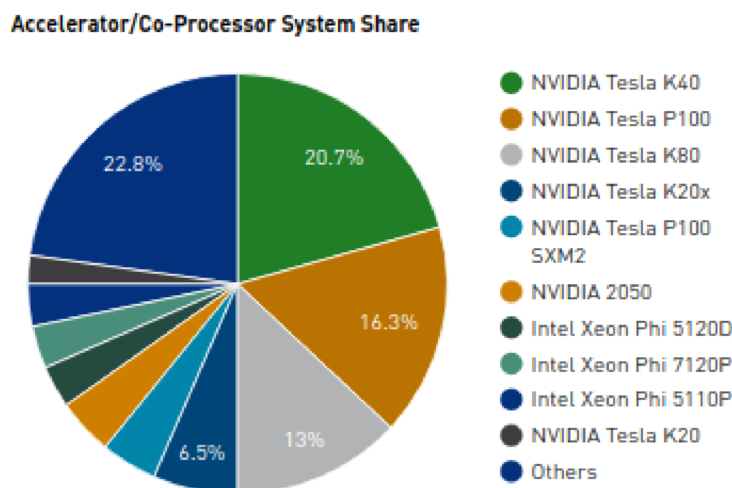

Figure 2.1: Accelerator and co-processor system share from the TOP500 list from June 2017. Pie chart retrieved from the TOP500 statistics generator.

The following microarchitecture is the latest: Pascal (14 nm and 16 nm), released in April 2016. There are three different NVIDIA GPGPUs with this microarchitecture: Tesla P4, Tesla P40 and Tesla P100. Although, the Tesla P100 has two different tiers and form factors. The most powerful tier is the one with 16 GB 4096-HBM2 memory and uses the SXM2 connector, which benefits from the NVLink technology that provides a much higher bi-directional interconnect bandwidth (300 GB/s NVLink versus 32 GB/s PCIe). The other two tiers are PCI Express, both have very similar specifications but always inferior to the

one with NVLink. These two PCI Express GPGPUs are differentiated by their memory capacity (16 GB 4096-HBM2 and 12 GB 4096-HBM2). Thus, resulting in different memory bandwidth between them (732 GB/s for the 16 GB model versus 549 GB/s for the 12 GB model).

The NVIDIA Tesla P100, being the latest GPGPU released, it has a strong presence in the HPC scene. Fifteen systems from the five hundred supercomputers that conform the TOP500 list are accelerated by NVIDIA Tesla P100, and nine out of those fifteen systems are in the Green500 top ten list (Figure 1.2) of most energy efficient systems.

Volta [37] is the new microarchitecture for the next generation of NVIDIA Tesla general-purpose GPUs. This new series will only manufacture one tier of V100, unlike the P100. Even though the GPU details have been completely revealed, no release date has been established.

The Tesla V100 is supposed to be a direct upgrade from the Tesla P100, as they both have 16 GB 4096-bit HBM2 (high-bandwidth memory) and the power consumption remains the same for both form factors: 250 W for PCIe and 300 W for NVLink.
In terms of computing capabilities, the Tesla V100 can reach peaks of 15 GFLOPS and 7.5 GFLOPS when operating with single-precision and double-precision floating point numbers, respectively. This score beats the Tesla P100, as it peaks 10.6 GFLOPS in single-precision and 5.3 GFLOPS in double-precision floating point numbers. This is due to the increase of the CUDA core count from 3584 (Tesla P100) to 5120 (Tesla V100) thanks to the new 12nm manufacturing process that allows the new Tesla V100 to have 21.1 billion ($21.1 \cdot 10^9$) transistors against the 15.3 billion ($15.3 \cdot 10^9$) from the Tesla P100. The performance lead is also because of the higher memory clock from the Tesla V100 over the P100 (1750 MHz vs 1406 MHz) translating in higher memory bandwidth (900 Gb/s vs 720 Gb/s).

2.3 CUDA

CUDA [38] stands for Compute Unified Device Architecture. At some point, they dropped the use of the full name and started officially referring to it as just CUDA. It is a parallel computing platform and programming model created by NVIDIA. Effectively, it is a software layer that eases the direct access to the GPU's virtual instruction set and parallel computational elements.

The first CUDA software development kit (SDK) was released in 2007 for Microsoft Windows and Linux, whilst macOS support had to wait until the next year. Since then, CUDA is compatible with every graphics card released by NVIDIA, both workstation-oriented (Tesla, Quadro) and consumer-oriented (GeForce).

It provides low and high-level API, meaning that gives full control to the developer over the data transfers, instruction paths and order of operations. CUDA is designed to work natively with C, C++ and FORTRAN and supports OpenACC directives for accelerators to automatically parallelize loops, but thanks to third party wrappers it is also compatible with Python, F#, .NET, Java, Perl, Ruby, Lua, Haskell, R, Matlab, IDL and Mathematica [39].

CUDA is used to accelerate non-graphical applications in different fields, like computational biology, chemistry or cryptography. However, it is used sometimes to accelerate graphical applications such as graphics rendering or physics calculations in video games.

CUDA, in the version 6.0, introduced a feature called *unified memory*. This feature consists of creating a pool of managed memory shared between the GPU and CPU, as shown in the left part of Figure 2.2. This simplifies the programming process as there

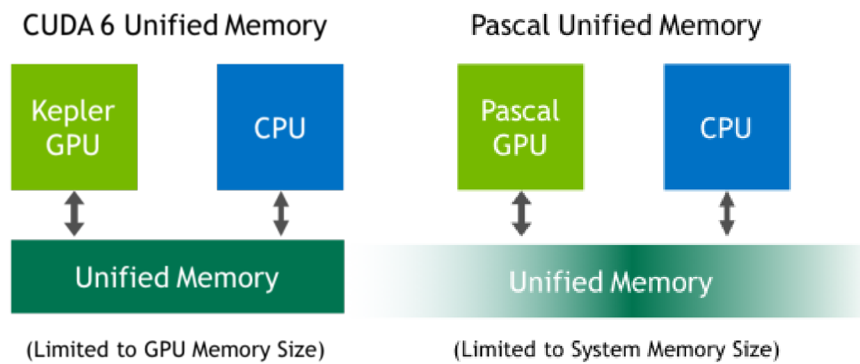

Figure 2.2: CUDA unified memory on Kepler and Pascal microarchitectures [40].

would be only one virtual address space for accessing all memory from the system. This feature has been further improved, as the total unified memory was limited to the GPU memory size until CUDA 8.0 was released along with the new Pascal GPUs. The new CUDA version with the new hardware is capable of having larger unified memory, limited to system memory size, as seen in the right part of Figure 2.2. The new Pascal devices are the first GPUs to include the *Page Migration Engine*, which is hardware support for unified memory page faulting and migration.

CUDA is just one of the options for parallel computing. We have listed in the following paragraphs its strengths and weaknesses compared to OpenCL [41] (its biggest competitor):

Strengths

+ CUDA has a great deep learning library named *deep neural network* (cuDNN) with immense support from NVIDIA, while OpenCL's focus is not oriented towards this field.

+ CUDA's focus is to get the highest computing performance possible, while OpenCL's goal is to make a heterogeneous computing API. We are counting this as an advantage in this thesis since our goal was is test general performance and not compatibility or system heterogeneity.

+ The exact same CUDA code will run without problems on every device that supports the necessary CUDA version, while with OpenGL you might need vendor-specific extensions to squeeze the maximum performance or to use the different device-specific features, meaning that, on very heterogeneous configurations, performance might be affected negatively, code complexity might see an increase or the code might not be fully compatible with some device.

Weaknesses

– In licensing terms, CUDA is the proprietary solution, whereas OpenCL is the open standard.

– CUDA is only supported by NVIDIA cards, while OpenCL is compatible with virtually any GPU, CPU, *digital signal processor* (DSP), *field-programmable gate array* (FPGA) or any other processor type.

– Windows, Linux and macOS are the only operating systems that support CUDA, while also FreeBSD and even some Android devices [42] support OpenCL.

2.4 Remote CUDA

Remote CUDA (rCUDA) is a middleware software framework for remote GPU virtualization. It allows the allocation of one or more CUDA-enabled GPUs to a single application. Each of the GPUs can be part of the cluster or run inside a virtual machine. rCUDA aims to increase the GPU utilization in systems that are lacking it, improving performance, and to lower the GPU unit count necessary per cluster, which lowers energy, acquisition and maintenance costs.

rCUDA follows a client-server architecture as shown in Figure 2.3:

Client is the cluster node demanding GPU acceleration services. The client runs the desired CUDA application using a library of wrappers to the high-level CUDA Runtime API as well as to the low-level CUDA Driver API. It emulates local GPUs when it has none attached. The client handles the communication with remote GPUs located in the server nodes.

Server is the cluster node offering GPU acceleration services. It is constantly listening for requests on a TCP port. The servers execute the request's code (CUDA kernels) in the GPU. The server is able to time-multiplex the GPU, for a higher utilization, by spawning different server processes for each request.

A possible hardware distribution in an HPC cluster is to have all the available GPUs attached to just a few selected nodes.

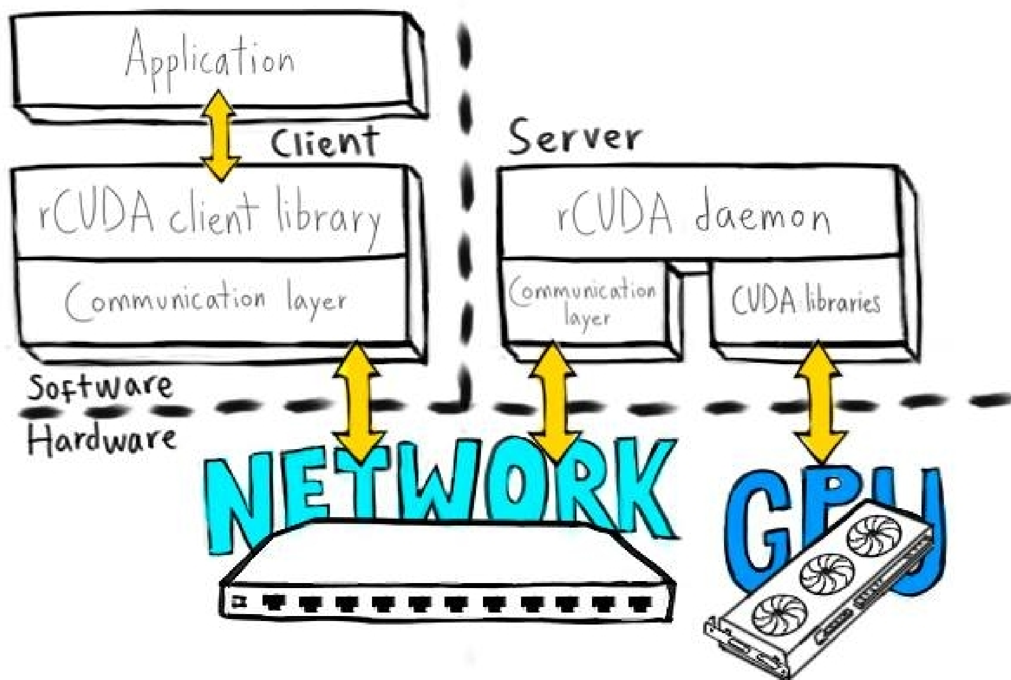

Figure 2.3: Visual representation of the rCUDA client-server communication and the hardware implication.

rCUDA is fully compatible with the CUDA API, therefore the use of this middleware is completely transparent to the user, as she/he just needs to execute a CUDA application without any special modification. After setting up rCUDA, it will take care of everything. The only requirement is that the application must be compiled with the *nvcc* compiler flag `-cudart=shared`, as rCUDA works with dynamic libraries.

rCUDA targets 64-bit Linux operating systems. The latest rCUDA version is 16.11 and comes with CUDA 8.0 support (except for graphics interoperability).

2.5 High-performance networks

In the HPC context, systems are not only able to handle large amounts of information in the cluster nodes, but also the communication between them. It allows the nodes to send data to each other as a part of parallel applications. As HPC focuses on performance, low latency networking and high bandwidth are mandatory to avoid bottlenecks.

In the next subsections, a couple of high-performance networks (HPN) will be introduced.

2.5.1 InfiniBand

InfiniBand (IB) is a computer networking standard used in high-performance computing. It is the best alternative to Ethernet and Fibre Channel because of its high bandwidth and low latency [43]. IB provides remote direct memory access capabilities (RDMA), which allows the communication from a computer's memory to another computer's memory without involving any operating system [44]. RDMA is intended for low CPU overhead. IB adapters can even handle networking protocols, unlike Ethernet networking protocols which are run on the CPU. The low CPU overhead and networking protocol isolation are optimal for HPC since high bandwidth transfers are highly likely to happen and more computing resources available could translate into higher throughput.

Throughout the years, improved data rates have been released to use with IB. At the very start, the IB transfer rates corresponded to the maximum ones supported by PCI Extended (PCI-X), but it severely affected performance. Nowadays, the IB transfer rates are set by the PCI Express (PCIe) revisions. IB is trying to achieve maximum throughput until it is limited by hardware, then it waits for a new PCIe revision and continues developing itself further.

The existing IB rates are Single Data Rate (SDR), Double Data Rate (DDR), Quad Data Rate (QDR), Fourteen Data Rate 10Gb/s per lane (FDR10), Fourteen Data Rate (FDR14 or simply FDR), Enhanced Data Rate (EDR) and High Data Rate (HDR).

In table 2.1 we can observe that there are two variants of FDR: FDR10 and FDR, and they offer different throughput. This is due to the improvement from 8/10 to 64/66 encoding. With the former, from every ten bits, eight bits carry data and two bits are dedicated to error correction. Therefore, effective throughput is lowered to 80%. With the latter, from every sixty-six bits, sixty-four carry data and two are dedicated to error correction. This allows an effective throughput of 96.97% [45].

As of today, the *Next Data Rate* (NDR) and *Extended Data Rate* (XDR) are still under development but their signaling rate has been predicted to be 100 Gb/s and 250 Gb/s respectively. NDR is expected to be released after 2020 and XDR release date has yet to be determined.

	SDR	DDR	QDR	FDR10	FDR	EDR	HDR [46]
Signaling rate (Gb/s)	2.5	5	10	13.125	14.1	25.8	51.6
Effective throughput (Gb/s per PCIe lane)	2	4	8	10	13.64	25	50
Common 4x link bandwidth (Gb/s)	10	20	40	52	56	100	200
Encoding (bits)	8/10	8/10	8/10	8/10	64/66	64/66	64/66
Adapter latency(μs)	5	2.5	1.3	0.7	0.7	0.5	< 0.5
PCIe revision	1.0	2.0	2.0	3.0	3.0	3.0	4.0
Year	2001	2005	2007	2011	2011	2014	2017

Table 2.1: InfiniBand data rates

2.5.2 RDMA over Converged Ethernet

RDMA over Converged Ethernet (RoCE) [47] is a network protocol that allows RDMA over an Ethernet network. There are two RoCE versions: RoCE v1 and RoCE v2.

RoCE v1 It is an Ethernet link layer protocol, allowing communication between any two hosts in the same Ethernet broadcast domain. Frame length is limited by the Ethernet's protocol limit: 1500 bytes for a regular frame and 9000 bytes for a jumbo frame.

RoCE v2 It is an Ethernet internet layer protocol, which enables packet routing. It exists on top of either UDP/IPv4 or UDP/IPv6. Even though UDP protocol does not guarantee the packet delivery order, it is required by RoCE v2 specification to not reorder them. This RoCE version includes a congestion control mechanism.

CHAPTER 3

Applications

In this chapter, we describe the software applications that we have used to measure the performance of the different systems and technologies.

3.1 CUDA samples: Bandwidth test

The CUDA toolkit comes with a wide set of code samples to test the capabilities of CUDA-compatible GPUs or just to see how some specific algorithms are implemented to be CUDA friendly.

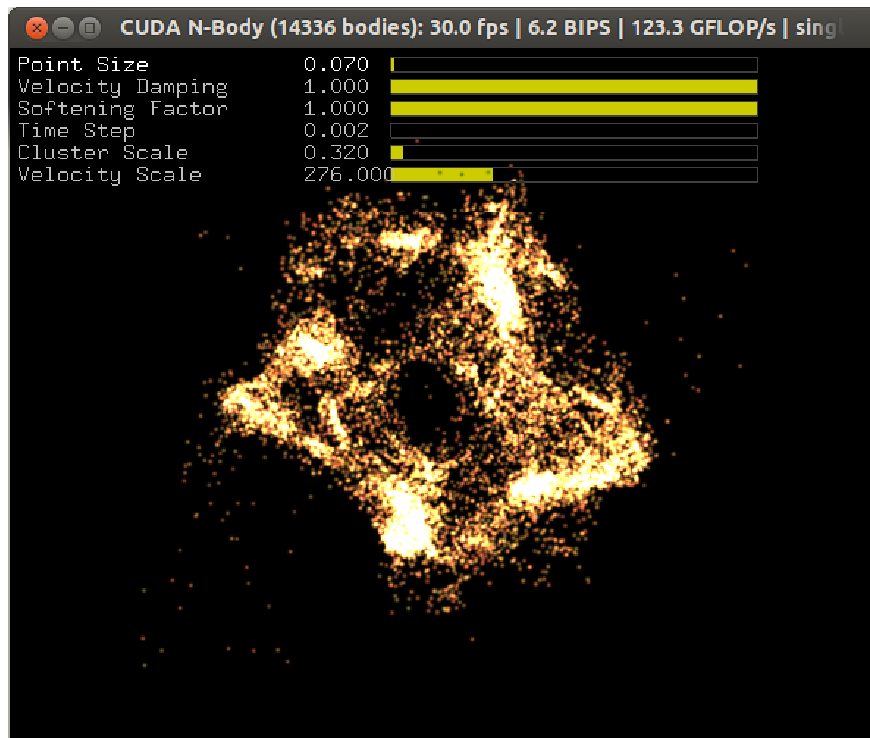

Figure 3.1: CUDA Toolkit sample N-Body from the simulations category.

They have been categorized by NVIDIA in 8 different types, starting from number zero.

0. **Simple:** basic samples to show how CUDA and CUDA runtime APIs work.

1. **Utilities:** samples that demonstrate how to query device capabilities and measure CPU/GPU bandwidth.

2. **Graphics:** samples that demonstrate interoperability between CUDA and DirectX or OpenGL.

3. **Imaging:** samples that demonstrate image processing, compression, and data analysis.

4. **Finance:** samples that demonstrate parallel algorithms for financial computing.

5. **Simulations:** samples that show some simulation algorithms implemented with CUDA.

6. **Advanced:** samples that show some advanced algorithms implemented with CUDA.

7. **CUDA Libraries:** samples that show how to use CUDA platform libraries (NPP, cuBLAS, cuFTT, cuSPARSE and cuRAND).

We have chosen as an introductory test the *bandwidth test*, from the *utilities* sample category. The choice for this test is motivated by its use in our work to characterize the maximum bandwidth achieved by each of the systems when accessing the local GPUs with CUDA and the remote ones with rCUDA.

Bandwidth test

This test is used to measure the bandwidth by executing *memcpy* instructions that copy a certain amount of data from CPU memory to GPU memory, and vice versa, or to copy the data from and to the GPU memory.

It has the possibility to use different parameters to adjust the test to our needs. As for this test, we are going to measure the transfers from CPU memory (referred to in the application as *host*) to GPU memory (referred to in the application as *device*) and the other way round. The device-to-device test is unnecessary for this work, as it would try out the GPU capabilities, which is not in our objectives. Therefore, we separated it into four different benchmarks:

Bandwidth test number	Memory type	Host to Device	Device to Host	Device to Device
1	Pinned	×		
2	Pinned		×	
3	Pageable	×		
4	Pageable		×	

Simplifying the terms: pageable memory is the one that can be swapped to disk and loaded back in memory when required. Pinned memory is the one that cannot be swapped out but skips the allocation of the memory block, translating in lower latencies. If the host does not have enough amount of memory, using pinned memory has a higher risk of leaving the system resourceless. We considered both memory types important to carry an exhaustive bandwidth test.

3.2 LAMMPS

LAMMPS [48] is the acronym of *Large-scale Atomic/Molecular Massively Parallel Simulator*. It is a molecular dynamics free and open-source software. It was originally developed under a Cooperative Research and Development Agreement (CRADA) between two public laboratories (Sandia and LLNL) and three companies (Cray, Bristol Myers Squibb, and Dupont) in the United States of America. Since 2006, the Sandia National Laboratories and Temple University are in charge of its maintenance and distribution.

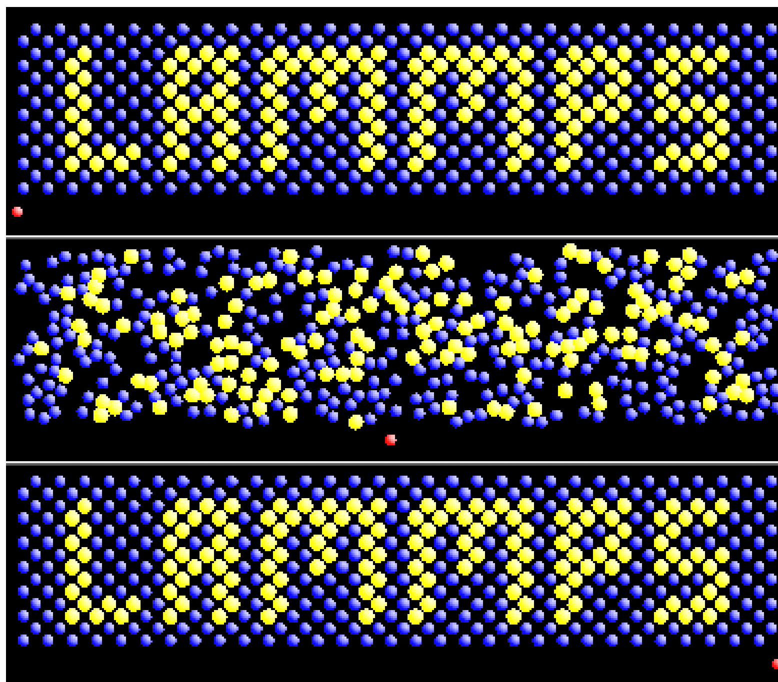

Figure 3.2: LAMMPS logo animation as an example of molecular dynamics. The top frame represents the lattice's initial state. The middle frame represents an intermediate step where the lattice has been dissolved. The bottom frame represents the final state with the reassembled lattice.

LAMMPS integrates Newton's equations of motion for collections of atoms, molecules, or macroscopic particles of which forces interact with a range of initial and boundary conditions.

To improve efficiency, LAMMPS uses *Verlet lists*: a data structure used in molecular dynamics simulations to efficiently keep track of neighbour particles within a set distance from each other. On parallel machines, LAMMPS uses spatial-decomposition techniques to partition the simulation domain into small 3D subdomains, one of which is assigned to each processor. Processors communicate and store *ghost* atom information for atoms that border their subdomain. LAMMPS is most efficient (in a parallel sense) for systems whose particles fill a 3D rectangular box with roughly uniform density.

It is highly scalable, as it has the ability to be run from a single processor machine to highly parallel or distributed machines thanks to the Message Passing Interface (MPI) [49] library. LAMMPS has the ability to model from just a few atoms up to billions of them. It supports Intel Xeon Phi co-processors[50], GPU computing with CUDA and OpenGL and even some OpenMP code features to squeeze the maximum performance.

In the Sandia National Laboratory they acknowledged the potential of the GPUs for computing, so they wanted to give LAMMPS the possibility to use them. Mike Brown, a researcher in Sandia, developed the first two features, that previously were to be computed

completely in the CPU, with GPU support: the Lennard-Jones potential and the Gay-Berne potential [51]. These features were introduced on January 2010 and over the years the number of features with a GPU-supported version increased significantly.

The data sets included in the package distribution consist of 32,000 atoms and run for 100 time-steps. It is possible to replicate the atoms via parameters to extend the problem dimensions like a 3D grid. As the problems with 32,000 are small sized, we have tuned the replication values to test our systems with different problem sizes. The data sets included in the package distribution are the following ones:

CHAIN benchmark

This benchmark is a simulation of a bead-spring polymer melt of 100-mer chains. It calculates FENE bonds and LJ pairwise interactions with a $2^{1/6}$ σ cutoff (5 neighbours per atom).

LJ benchmark

This benchmark is a simulation of an atomic fluid. It calculates the Lennard-Jones potential with 2.5 σ cutoff (55 neighbours per atom).

EAM benchmark

This benchmark is a simulation of a metallic solid. It calculates the Cu (copper) EAM potential with 4.95 Å cutoff (45 neighbours per atom).

3.3 CUDA-MEME

CUDA-MEME [18] is an ultrafast scalable motif discovery algorithm based on *multiple em for motif elicitation* (MEME) algorithm for multiple GPUs using a combination of CUDA, MPI and OpenMP.

CUDA-MEME and, afterwards, mCUDA-MEME are software tools that Dr. Yongchao Liu (Nanyang Technological University) developed associated to his paper publications. The motivation to improve the MEME method was that its complexity was too high to be feasible: $O(N^2 * L^2)$ where N is the number of input sequences and L is the length of each sequence. There were different applications also attempting to optimize the MEME method, such as ParaMEME (parallel MEME implementation using MPI) or GPU-MEME (MEME parallelized using OpenGL to run in GPUs), but CUDA-MEME was the only one at the moment that was using CUDA.

The first paper from CUDA-MEME was published online on October 2009, and the respective CUDA-MEME application was released shortly after. The first version of follow-up tool, mCUDA-MEME, was released in July 2011.

A sequence motif is a nucleotide or amino-acid sequence pattern that is widespread and has a biological significance. The sequence alphabets can be from DNA, RNA or protein, but CUDA-MEME is compatible with all of them and even custom alphabets.

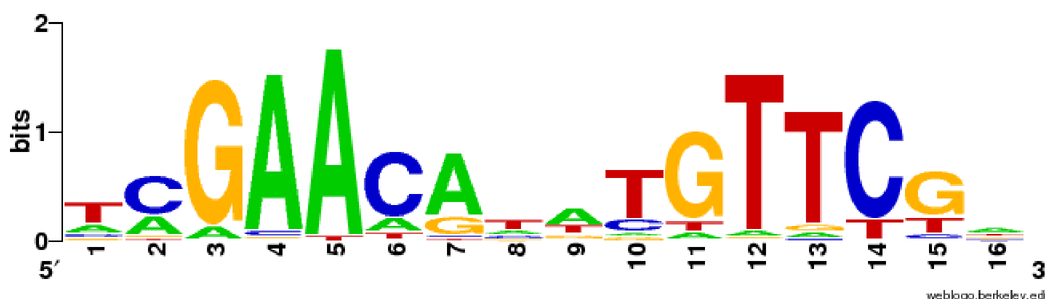

Figure 3.3: DNA sequence motif in a sequence logo representation

CUDA-MEME supports single GPU only and it is based on MEME version 3.5.4, while mCUDA-MEME is a further extension with higher accuracy, speed and support for GPU clusters that is based on the use of the MPI library.

NRSF benchmark

For our work we used the benchmarking data sets provided by the CUDA-MEME team as test cases. They are from neuron-restrictive silencer factor (NRSF), a protein present in humans.

The files containing the data set are in the FASTA format. This format is text-based and has become a standard in bioinformatics, as it is really simple and easy to manipulate programmatically. It consists of just a description line starting with the character > (greater-than) followed by the sequence, in which the nucleotides or amino acids are represented using single-letter codes.

The benchmarking suite contains three FASTA files. The first file consists of five hundred NRSF sequences, the second file contains one thousand NRSF sequences and the last file comprises two thousand NRSF sequences.

3.4 NAMD

NAMD [52], short for *Nanoscale Molecular Dynamics*, is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. Based on Charm++ parallel objects, NAMD scales to hundreds of cores for typical simulations and beyond 500,000 cores for the largest simulations. To that end, NAMD uses the MPI library, which allows distributing NAMD processes across a cluster. NAMD is distributed free of charge with source code.

NAMD uses the molecular graphics program VMD (Visual Molecular Dynamics) for simulation setup and trajectory analysis.

The release of the first version of NAMD was in 1995. It was developed by only four people, of which three were computer science graduate students and one professor with experience in theoretical chemistry and biology. Since then, it has been developed and maintained by the Theoretical and Computational Biophysics Group (TCB) and Parallel Programming Laboratory (PPL) from the University of Illinois at Urbana–Champaign.

At first, NAMD would make all the calculations in the CPU. The CUDA support was introduced in the version 2.7b2, for testers, and in the version 2.7, for the general public. This version was released in late 2010. It supported CUDA versions 2.3, 3.0 and 3.1, the latest ones at that time [53].

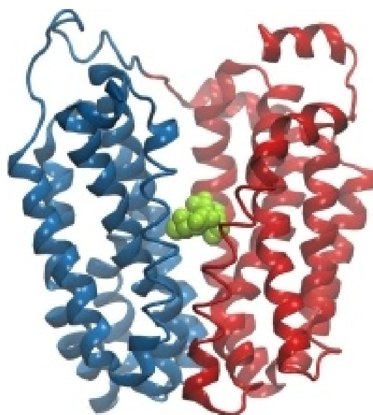

Figure 3.4: *NewRibbons* representation rendered with VMD.

We chose to use the following tests, which are included in the NAMD software package:

ApoA1 benchmark

This test makes a simulation of the *apolipoprotein A1*. It is periodic, meaning that the unit cells conform tiles and they are constantly repeated. This test consists of 92,224 atoms. This has been the standard NAMD cross-platform benchmark for years.

ATPase benchmark

This test makes a simulation with *ATPase* enzymes. It is periodic as well and consists of 327,506 atoms.

STMV benchmark

This test makes a simulation of the *satellite tobacco mosaic virus*. This one is periodic too and consists of 1,066,628 atoms. The STMV benchmark is useful for demonstrating scaling to thousands of processors.

CHAPTER 4
Performance analysis

In this chapter, we present the actual systems that were tested, how they were tested and which were the results. The idea of the tests carried out is to analyze the feasibility of using the remote GPU virtualization mechanism in order to provide GPU access to low-power processors based systems, which usually do not include the required PCIe connector and power supply to own a GPU. In this manner, the applications described in the previous chapter will be executed in the nodes with low-power processors while using the GPU installed in another node of the cluster. In order to reduce the overhead of using a remote GPU, the low-power processor systems are equipped with InfiniBand network cards. On the other hand, in order to have the proper performance reference numbers, we have additionally executed the test applications in a regular non-low-power system. In order to make this analysis as complete as possible, we have carried out the tests with the regular non-low-power system in two different scenarios. In the first scenario, the GPU is installed inside the regular non-low-power system and CUDA was used in the traditional way. In the second scenario, the GPU was installed in another node of the cluster and rCUDA was used to access that GPU.

On the other hand, given that rCUDA provides a different (and longer) path to the remote GPU before studying the performance of the three applications described in Chapter 3, we characterize the bandwidth attained by CUDA and rCUDA when moving data to/from the GPU. This characterization of the achieved bandwidth is carried out for all the systems considered in this work. Furthermore, given that the performance of rCUDA depends on the performance provided by the underlying network fabric, we also characterize the bandwidth attained by InfiniBand for each of the systems used in this study.

This chapter is organized in the following way: Section 4.1 provides a thorough description of the different nodes used in our study along with the installation procedure of rCUDA. Next, in Section 4.2, a performance study of InfiniBoard is provided. This study is carried out in the systems described in Section 4.1. Later, Section 4.3 provides the bandwidth numbers for CUDA and rCUDA. Finally, Sections 4.4, 4.5 and 4.6 provide the performance results of the LAMMPS, CUDA-MEME and NAMD applications, respectively, when executed with CUDA and rCUDA in the systems described in 4.1. The discussion from this chapter is presented in Chapter 5 titled "Conclusions"

4.1 System configuration

The *Parallel Architectures Group* (GAP) from *Universitat Politècnica de Valencia* owns a cluster, named *virtualgap*, which includes nodes comprising several processor architectures. GPUs are included in some of the nodes. We are going to use five nodes from the cluster. We are

going to use their usual identifiers to refer to them in the analysis. The characteristics of the nodes are depicted in Tables 4.1, 4.2, 4.3 and 4.4.

Barebone	Super Micro SuperServer 1027GR-TSF
CPU(s)	Intel Xeon E5-2620 V2
GPU(s)	1x NVIDIA Tesla K20m
RAM	8x 4 GB 1600 MHz DDR3 ECC LOW-VOLTAGE
PCIe	v3.0 x16 lanes
Network card	InfiniBand card MCX353A-FCBT
Data rate	FDR (56 Gb/s)
OS	CentOS 7.3

Table 4.1: Specification for the nodes mlxm1 and mlxm2

Barebone	Super Micro SuperServer 5018D-FN4T
CPU(s)	Intel Xeon D-1541
GPU(s)	-
RAM	4x 16 GB 2133 MHz DDR4 ECC 1.2V
PCIe	v3.0 x16 lanes
Network card	InfiniBand card MCX353A-FCBT
Data rate	FDR (56 Gb/s)
OS	CentOS 7.3

Table 4.2: Specification for the node mlxd1

Barebone	Super Micro SuperServer 5018A-TN4
CPU(s)	Intel Atom C2750
GPU(s)	-
RAM	4x 8 GB 1600 MHz DDR3 ECC 1.2V
PCIe	v2.0 x8 lanes
Network card	InfiniBand card MCX353A-FCBT
Data rate	QDR (40 Gb/s)
OS	CentOS 6.9

Table 4.3: Specification for the node mlxa1

Barebone	Avantek R120-T30
CPU(s)	CAVIUM ThunderX
GPU(s)	-
RAM	8x 16 GB 2133 MHz DDR4 ECC
PCIe	v3.0 x8 lanes
Network card	1x QSFP+ socket / 4x SFP+ sockets
Data rate	FDR (56 Gb/s)
OS	CentOS 7.3

Table 4.4: Specification for the node mlxarm1

Table 4.1 describes the configuration of nodes mlxm1 and mlxm2. It can be seen in this table that these nodes include an Intel Xeon CPU and one NVIDIA Tesla K20m as processing units, eight RAM sticks of 4 GB working at a frequency of 1600 MHz for a total of 32 GB as system memory and an InfiniBand card in a PCIe v3.0 x16 slot for networking. The InfiniBand card is connected to an FDR switch, thus, allowing a bandwidth of up to 56 Gb/s. These parts are assembled inside a barebone, which is a chassis with a motherboard. The barebones are commonly used in data centres thanks to its form factor being suitable for distributed systems and its high-end hardware support. The mlxm1 and mlxm2 nodes are assembled in Super Micro SuperServer 1027GR-TSF [54] barebones.

Furthermore, the configuration of the node mlxd1 is provided in Table 4.2. It can be seen in this table that mlxd1 comprises an Intel Xeon D CPU, four RAM sticks of 16 GB working at a frequency of 2133 MHz for a total of 64 GB system memory and an InfiniBand card in a PCIe v3.0 x16 slot for networking. The InfiniBand card is also connected to an FDR switch for up to 56 Gb/s maximum bandwidth. The node mlxd1 is assembled in a Super Micro SuperServer 5018D-FN4T [55] barebone.

Moreover, Table 4.3 describes the Atom-based system. This node contains an Intel Atom CPU, four RAM sticks of 8 GB working at a frequency of 1600 MHz for a total of 32 GB system memory and an InfiniBand card in a PCIe v2.0 x8 slot for networking. The InfiniBand card is connected to a QDR switch this time, allowing a maximum bandwidth of 40 Gb/s. The node mlxa1 is assembled in a Super Micro SuperServer 5018A-TN4 [56] barebone.

Finally, Table 4.4 depicts the configuration for the ARM-based system. The table shows that this system has a CAVIUM ThunderX ARMv8 CPU and eight RAM sticks of 16 GB working at 2133 MHz for a total of 128 GB system memory. As the motherboard includes its own networking card, it was unnecessary to add an InfiniBand card like in the other systems. The QSFP+ socket is directly compatible with InfiniBand, therefore is connected to an FDR switch allowing up to 56 Gbps of bandwidth for this node. The node mlxarm1 is assembled in an Avantek R120-T30 [57] barebone.

Notice that every node but the ones with Intel Xeon (mlxm1 and mlxm2) are lacking GPUs. Hence, they completely rely on the shared GPUs provided by rCUDA for the tests as they are unable to make CUDA computations by themselves.

For the test in this work, we have defined five different configurations according to the available devices and technologies previously described. These are the five configurations studied:

- **CUDA-Xeon** - mlxm1 running the application using CUDA locally. rCUDA is not used. This configuration will be the baseline for the performance comparison, as it

is the default system configuration nowadays (although the exact characteristics of the node will probably change, it is the default characteristic because it comprises a GPU which is used with CUDA in the traditional way, that it, the application and the GPU are on the same node)

- **rCUDA-Xeon** - mlxm2 running the applications as an rCUDA client, mlxm1 as a rCUDA server. This configuration will be useful to assess the overhead of CUDA, thus providing a second reference in our comparison.
- **rCUDA-XeonD** - mlxd1 running the applications as an rCUDA client, mlxm1 used as an rCUDA server.
- **rCUDA-Atom** - mlxa1 running the applications as an rCUDA client, mlxm1 used as an rCUDA server.
- **rCUDA-ARM** - mlxarm1 running the applications as an rCUDA client, mlxm1 used as an rCUDA server.

To distinguish easily the configurations in the rest of the thesis, we have used the bold names from above as identifiers in the rest of this chapter.

As the CAVIUM ThunderX has the only ARM64 processor in our configuration, it was necessary to have the applications built again specifically for its architecture. As there is no ARM64 version for CUDA 8.0, *cross compiling* was mandatory, which means creating executable code for a platform other than the one on which the compiler is running. NVIDIA released different packages to support cross compilation for ARMv7, ARMv8 and POWER8. In our case, we installed the ARMv8 package, named `cuda-cross-aarch64`. We encountered the problem that not every library is supported by cross compilation. Thus, it does not include the display driver. Therefore, not all applications could be compiled nor tested in this investigation.

4.1.1 Setting up rCUDA

Installing CUDA should be something trivial and NVIDIA has made available extensive documentation about the installation process. Therefore, in this thesis we are assuming, for the sake of simplicity, that latest CUDA version supported by rCUDA has already been installed on every cluster node involved.

Regarding the installation of rCUDA, he tarball provided by the rCUDA team has to be decompressed first. The resulting contents are described in the file *contents.txt*, included in the tarball, as follows:

```
- bin: rCUDA server daemon
- doc: rCUDA user guide
- lib: rCUDA libraries
- contents.txt: this file
- LICENSE.txt: rCUDA terms of use
```

Setting up the rCUDA server

To start up a rCUDA server we need to set the following environment variables first:

```
# Replace <CUDA> with the actual CUDA folder
# IF Deep Neural Networks are going to be used, replace <CUDA_cuDNN> with the
```

```
#    CUDA Deep Neural Network library folder
# OTHERWISE replace <CUDA_cuDNN> with the dummy cuDNN library folder provided
#    in the rCUDA tarball -> rCUDA/lib/cudnn/
export LD_LIBRARY_PATH=<CUDA>/lib64/:<CUDA_cuDNN>:$LD_LIBRARY_PATH
export PATH=<CUDA>/bin/:$PATH
export RCUDAPROTO=IB  # Only necessary when using InfiniBand
```

We need to execute the server daemon to start listening for requests. The daemon executable file, named *rCUDAd*, is located in the *bin* folder.

```
cd /usr/local/rCUDA/bin/
./rCUDA
```

For debugging purposes, it could be interesting for the user to execute the daemon as interactive and/or verbose:

```
./rCUDA -iv
```

This could be a working example:

```
export
↪  LD_LIBRARY_PATH=/usr/local/cuda/lib64/:/usr/local/rCUDA/lib/cudnn/:$LD_LIBRARY_PATH
export PATH=/usr/local/cuda/bin/:$PATH
cd /usr/local/rCUDA/bin/
./rCUDA
```

Because the server is a daemon, there is no need to keep the session open. For as long as the cluster node is powered on, the node will continue working as a GPU server, although it needs to be set up again on node restart.

Setting up the client

To set up the client we need to set the following environment variables:

```
# Replace <rCUDA> with the actual rCUDA folder
export LD_LIBRARY_PATH=<rCUDA>/lib/:$LD_LIBRARY_PATH

# Replace <CUDA> with the actual CUDA folder
export PATH=<CUDA>/bin/:$PATH

export RCUDAPROTO=IB        # Only necessary when using InfiniBand

# Replace <total_devices> with the amount of devices that this client will
#   send requests
export RCUDA_DEVICE_COUNT=<total_devices>

# Replace:
#  -  <n> with a number starting from 0, increasing by 1 for every rCUDA device
#  -  <server_ip_address> with the IP address from the cluster node that will
#         work as a server
#  -  <device_number> with the server device number that will do the CUDA
#         computations. For single GPU systems, just write a 0 (zero number)
export RCUDA_DEVICE_<n>=<server_ip_address>:<device_number>
```

This could be a working example:

```
export LD_LIBRARY_PATH=/usr/local/rCUDA/lib/:$LD_LIBRARY_PATH
export PATH=/usr/local/cuda/bin/:$PATH
export RCUDA_DEVICE_COUNT=4
export RCUDA_DEVICE_0=192.168.0.10:0
export RCUDA_DEVICE_1=192.168.0.10:1
export RCUDA_DEVICE_2=192.168.0.11:0
export RCUDA_DEVICE_3=192.168.0.12:0
```

In this example, the client has been set up to send the requests to four different rCUDA devices, each of them being one GPU from a remote node. Notice that there is only three different IP addresses but four rCUDA devices, meaning that the server `192.168.0.10` has, at least, two GPUs connected. The other two servers have, at least, one GPU connected. A client does not necessarily need to use every GPU connected to a server. If a remote GPU is not defined as an rCUDA device, the client will not send it any requests.

Once the environment variables are set, the client should already be able to execute any CUDA application and send requests to the assigned servers.

4.2 InfiniBand bandwidth test

Given that in our work we are going to make use of the InfiniBand network in order to reduce the overhead of using a remote GPU, in this section we characterize the achieved bandwidth by InfiniBand adapters installed in each of the systems.

4.2.1 Description

Before testing and comparing results from the applications with GPU, we benchmarked the network bandwidth. The systems use different InfiniBand data rates, therefore we would reach an inaccurate conclusion just by checking the GPU results.

Notice that IB uses RDMA, therefore the operating system does not interfere in the data transfers. Thus, what it is measured is the data transfers done directly from one system's memory to the other system's memory.

We separated the network bandwidth benchmark into three tests, according to the available tests provided by the IB software: *write*, *read* and *send*.

In order to understand the difference between the write, read, and send benchmarks provided in the InfiniBand software package, it is important to remark that InfiniBand adapters provide the InfiniBand Verbs API. This API offers two communication mechanisms: the channel semantics and the memory semantics. The former refers to the standard send/receive operations typically available in any networking library, while the latter offers RDMA operations where the initiator of the operation specifies both the source and destination of a data transfer, resulting in zero-copy data transfers with minimum involvement of the CPUs. rCUDA employs both semantics, selecting one or the other depending on the exact communication to be carried out. Thus, the send benchmark follows the channel semantics whereas the write and read benchmarks obey to the memory semantics. Furthermore, when the benchmarks are executed (typically between two nodes which can be referred to as node A and node B), the send benchmark sends data from node A to node B. On the other hand, the write benchmark executed in node A writes data into the memory of node B whereas the read benchmark executed in node A reads memory from node B and brings those memory contents to node A.

We added the options `-aF` for convenience, so we get a deeper test and ignore some debugging messages. A precise description is given in its *man* file:

```
Usage:
ib_write_bw           start a server and wait for connection
ib_write_bw <host>    connect to server at <host>

Options:
-a, --all  Run sizes from 2 till 2^23
...
-F, --CPU-freq  Do not show a warning even if cpufreq_ondemand module is
   loaded, and cpu-freq is not on max.
```

It can be seen in the *man* file that there are two different usage choices to execute the test. This is because the InfiniBand bandwidth benchmarks follow a client-server architecture. Therefore, the server has to be created first, so it starts listening for connections. To start a server, the IP address field must be left empty when executing the command to initialize it. Afterwards, the client can be set up to connect to an existing server. To connect to a server with the client, the IP address of the server must be written when executing the command to initialize the client.

The command `ib_write_bw` performs RDMA write transactions. It does five thousand iterations from each different transaction size.

The syntax would be:

```
ib_write_bw [ options ] [ server IP address ]
```

Therefore, we have to leave it blank after the options when executing it on the server and write the server IP address when executing it on the client. It looks as follows:

mlxm1:	`ib_write_bw -aF`
Other systems:	`ib_write_bw -aF mlxm1`

The command `ib_read_bw` performs RDMA read transactions. It does one thousand iterations from each different transaction size.

The command `ib_send_bw` performs RDMA send transactions. It does one thousand iterations from each different transaction size.

The syntax for the options is exactly the same for all three, so the commands used are:

mlxm1:	`ib_read_bw -aF`	`ib_send_bw -aF`
Other systems:	`ib_read_bw -aF mlxm1`	`ib_send_bw -aF mlxm1`

In summary, the InfiniBand tests carried out in this section will use the mlxm1 node as server for all the tests whereas the other systems (mlxm2, mlxd1, mlxa1, mlxarm1) will be used as clients for their respective tests.

4.2.2 Results

Once the client successfully connects to the server, the test would start. As we chose to use the options `-aF`, the tests will be repeated with increasing transaction sizes from 2 B until 8 MB. Per example, the output from the write test between mlxm1 (server) and mlxm2 (client) would be:

```
---------------------------------------------------------------------------
RDMA_Write BW Test
Dual-port       : OFF          Device         : mlx4_0
Number of qps   : 1            Transport type : IB
Connection type : RC           Using SRQ      : OFF
TX depth        : 128
CQ Moderation   : 100
Mtu             : 2048[B]
Link type       : IB
Max inline data : 0[B]
rdma_cm QPs     : OFF
Data ex. method : Ethernet
---------------------------------------------------------------------------
local address: LID 0x08 QPN 0x021b PSN 0x73dc1f RKey 0x18010900 VAddr
↪   0x007fabefc00000
remote address: LID 0x16 QPN 0x027c PSN 0x8dc71 RKey 0xa801093a VAddr
↪   0x007f466f000000
---------------------------------------------------------------------------
#bytes   #iterations   BW peak[MB/sec]   BW average[MB/sec]   MsgRate[Mpps]
2        5000          6.28              6.23                   3.263732
4        5000          17.68             17.59                  4.610201
8        5000          39.56             38.22                  5.010221
16       5000          79.32             76.45                  5.010288
32       5000          158.25            155.37                 5.091120
64       5000          315.71            305.83                 5.010644
128      5000          631.43            610.81                 5.003752
256      5000          1265.81           1220.17                4.997808
512      5000          2507.19           2371.44                4.856700
1024     5000          5026.63           4845.25                4.961535
2048     5000          5712.71           5656.47                2.896115
4096     5000          5789.37           5785.95                1.481204
8192     5000          5899.68           5894.84                0.754540
16384    5000          5943.47           5939.46                0.380126
32768    5000          5967.84           5966.42                0.190925
65536    5000          5978.92           5978.37                0.095654
131072   5000          5984.50           5983.75                0.047870
262144   5000          5986.02           5985.87                0.023943
524288   5000          5986.57           5984.98                0.011970
1048576  5000          5987.59           5986.22                0.005986
2097152  5000          5986.24           5986.15                0.002993
4194304  5000          5989.25           5988.87                0.001497
8388608  5000          6013.91           6011.74                0.000751
---------------------------------------------------------------------------
```

The returned data is enough to be able to plot it on a graph and have a clear chart to compare the different resulting bandwidths from the systems. Therefore, we carried out each test (write, read and send) once in every system designated as a client (mlxm2, mlxd1, mlxa1, mlxarm1) while the server (mlxm1) was listening for connections, collecting the output data. Then we gathered it on a spreadsheet and created the charts. The test results were as follows:

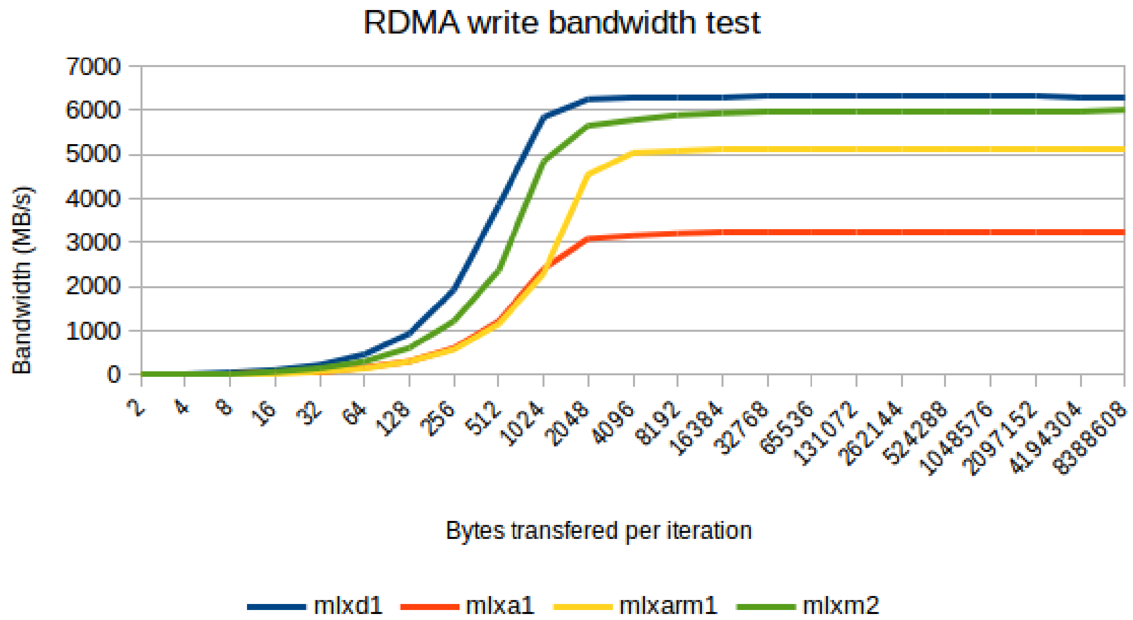

Figure 4.1: InfiniBand RDMA *write* bandwidth test results. Higher is better.

The results from the write test shown in Figure 4.1 are as expected. The XeonD is leading with a slight advantage over the Xeon around the 6000 MB/s mark, followed by the ARM just above the 5000 MB/s mark and leaving behind the Atom as low as almost 3000 MB/s. It clearly reflects the difference between QDR (mlxa1) and FDR (mlxd1, mlxarm1 and mlxm2) InfiniBand data rates. The XeonD is the node with the highest bandwidth because has the fastest memory (2133 MHz). On the other hand, ARM's memory is as fast as the one from XeonD, although it appears that provides lower performance.

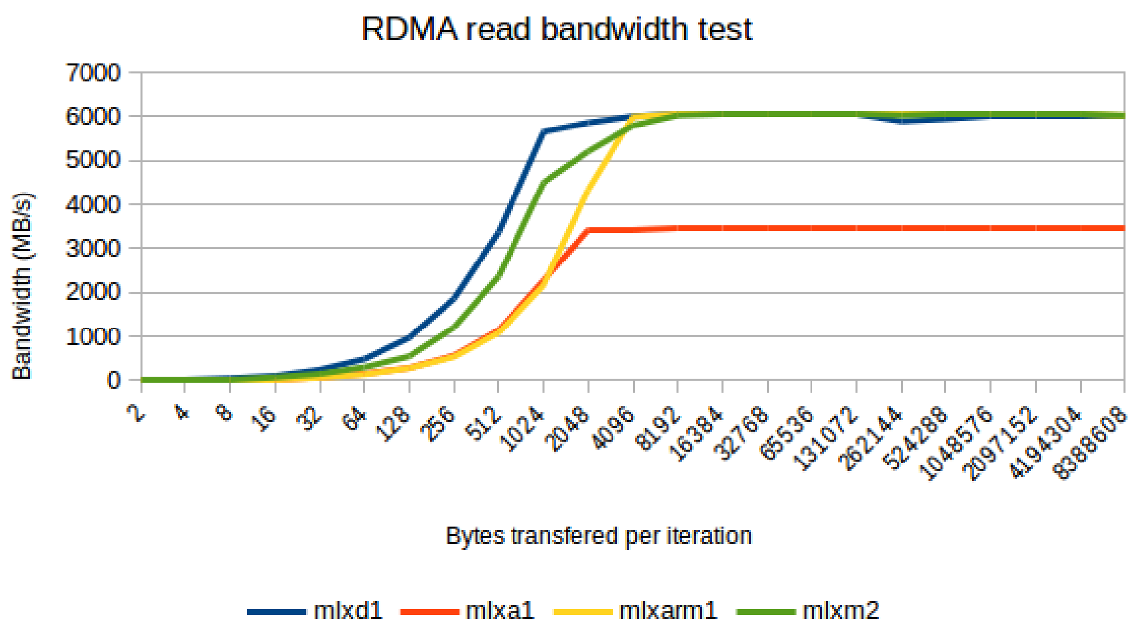

Figure 4.2: InfiniBand RDMA *read* bandwidth test results. Higher is better.

Figure 4.3 depicts the chart derived from the send test. The XeonD is leading until at 4096 bytes per iteration all but the Atom converge to 6000 MB/s. The Atom and the ARM have similar values from the start, but the ARM eventually raises its bandwidth with the Xeons while the Atom stays at 3500 MB/s.

This test is bottlenecked by the network speed, as even having different memory speeds, the nodes with FDR data rate coincide in peak memory bandwidth. The Atom node, as the only node with QDR data rate, is also bottlenecked by the network but the QDR bandwidth peak is significantly smaller than the FDR one.

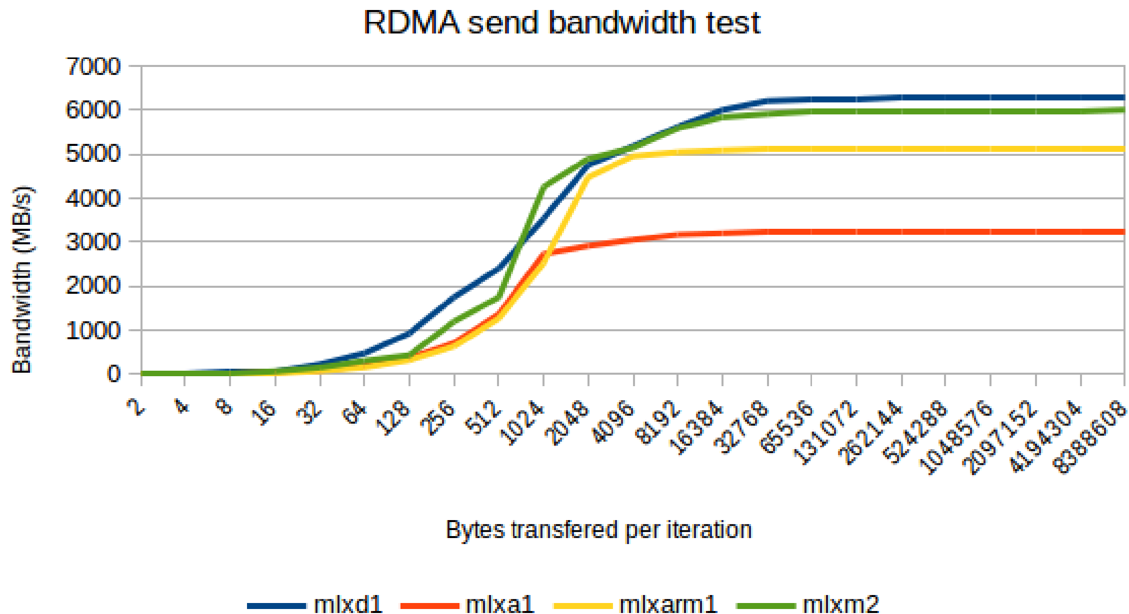

Figure 4.3: InfiniBand RDMA *send* bandwidth test results. Higher is better.

The last chart, Figure 4.3, shows that the results are very similar to the *write* bandwidth test: the Xeons at the top of the list, the ARM following them from not very far away and the Atom got stuck at half the others' bandwidth. The reasons for this scores are also the same ones: XeonD is the first, having the fastest performing memory; Xeon is in second place; ARM is the third system, having fast memory but without optimal performance; and the last one is the Atom, with the slowest InfiniBand data rate (QDR).

4.3 Testing bandwidth of CUDA and rCUDA

Once the bandwidth of the underlying network fabrics has been characterized, our next step is to study the bandwidth achieved when moving data to/from the GPU with CUDA and rCUDA.

4.3.1 Description

The *BandwidthTest* application is a good benchmark for measuring GPU bandwidth, as we need to measure first just the potential bandwidth from every system before jumping to conclusions about the performance of the applications.

A basic bandwidth test can be done just executing this application without any parameters:

```
$ ./bandwidthTest
```

It performs just a *Quick Mode* run, which means it uses pinned memory, has a transfer size of 33554432 bytes and tests all three directions: host to device, device to host and device to device.

This is not very thorough, therefore we studied its launch options to adapt the benchmark to our needs. A simple description of the supported parameters is given by executing the command `$ ./bandwidthTest --help`:

```
Usage:  bandwidthTest [OPTION]...

[...]

Options:
   --help  Display this help menu
   [...]
   --memory=[MEMMODE]      Specify which memory mode to use
      pageable - pageable memory
      pinned   - non-pageable system memory
   --mode=[MODE]   Specify the mode to use
       quick - performs a quick measurement
       range - measures a user-specified range of values
       shmoo - performs an intense shmoo of a large range of values
   --htod  Measure host to device transfers
   --dtoh  Measure device to host transfers
   --dtod  Measure device to device transfers
   [...]
```

From these options, we chose to use the memory type parameter for our tests, as we need to benchmark both pageable and pinned. We also use the mode parameter, because selecting the *shmoo* mode automatically sets up a wide test transfer size range. As we did not need the *device to device* measure, we selected the other two.

The final instructions to be executed in each node were:

```
$ ./bandwidthTest --memory=pageable --mode=shmoo --htod --dtoh
$ ./bandwidthTest --memory=pinned   --mode=shmoo --htod --dtoh
```

So for the four different test we are doing we only need two instructions, as each bandwidthTest measured both *device to memory* and *memory to device* at once.
These instructions will be executed once each in every system configuration we are studying. After running the benchmarks, we have to gather the results into a spreadsheet and create the appropriate plots.

4.3.2 Results

The output from the bandwidthTest execution has a simple structure but long and not very pleasant to the human eye, as it is just a lengthy list of numbers. For instance, this was the result from the execution of the test with pageable memory in the mlxm1 node using CUDA:

```
[CUDA Bandwidth Test] - Starting...
Running on...

Device 0: Tesla K20m
Shmoo Mode

............................................................................
Host to Device Bandwidth, 1 Device(s)
PAGEABLE Memory Transfers
Transfer Size (Bytes)        Bandwidth(MB/s)
1024                         76.7
2048                         149.3
3072                         207.5
4096                         267.9
5120                         318.5
6144                         363.2
7168                         411.8
8192                         463.8
9216                         475.9
10240                        566.9
11264                        605.4
12288                        646.2
13312                        675.7
14336                        709.9
15360                        742.9
16384                        807.7
17408                        861.2
18432                        863.7
19456                        919.7
20480                        935.9
22528                        981.7
24576                        1036.7
26624                        1083.8
28672                        1088.6
30720                        1130.4
32768                        1147.0
34816                        1192.6
36864                        1221.6
38912                        1259.1
40960                        1301.2
43008                        1302.7
45056                        1333.7
47104                        1240.6
49152                        1358.6
51200                        1363.5
61440                        1450.0
71680                        1497.0
81920                        1568.0
92160                        1639.9
102400                       1709.1
204800                       1931.8
307200                       2013.8
```

```
409600                      1425.2
512000                      1368.7
614400                      1216.5
716800                      1201.6
819200                      1258.1
921600                      1354.5
1024000                     1848.2
1126400                     2270.4
2174976                     2706.8
3223552                     2908.8
4272128                     3021.0
5320704                     3094.8
6369280                     3136.8
7417856                     3180.7
8466432                     3210.8
9515008                     3233.0
10563584                    3252.3
11612160                    3267.2
12660736                    3277.7
13709312                    3260.1
14757888                    3273.7
15806464                    3256.4
16855040                    3277.8
18952192                    3215.1
21049344                    3219.9
23146496                    3171.7
25243648                    3189.9
27340800                    3154.7
29437952                    3182.8
31535104                    3158.1
33632256                    3187.3
37826560                    3193.3
42020864                    3196.9
46215168                    3200.9
50409472                    3205.1
54603776                    3205.0
58798080                    3209.8
62992384                    3210.0
67186688                    3213.2

.........................................................................
Device to Host Bandwidth, 1 Device(s)
PAGEABLE Memory Transfers
Transfer Size (Bytes)       Bandwidth(MB/s)
1024                        73.0
2048                        143.2
3072                        206.3
4096                        268.1
5120                        338.8
6144                        389.6
7168                        433.7
8192                        460.6
```

```
9216                        520.5
10240                       566.3
11264                       600.4
12288                       645.9
13312                       710.0
14336                       750.7
15360                       745.2
16384                       779.3
17408                       804.5
18432                       834.5
19456                       879.2
20480                       916.2
22528                       945.8
24576                       982.4
26624                       1063.0
28672                       1094.2
30720                       1131.2
32768                       1153.6
34816                       1153.7
36864                       1203.6
38912                       1270.1
40960                       1331.6
43008                       1335.9
45056                       1348.5
47104                       1379.7
49152                       1408.0
51200                       1428.2
61440                       1505.3
71680                       1584.1
81920                       1670.5
92160                       1704.2
102400                      1745.5
204800                      1936.9
307200                      2058.6
409600                      2112.0
512000                      2147.4
614400                      2165.0
716800                      2190.6
819200                      2200.1
921600                      2208.4
1024000                     2219.3
1126400                     2268.5
2174976                     2724.1
3223552                     2938.3
4272128                     3056.6
5320704                     3131.9
6369280                     3185.0
7417856                     3228.5
8466432                     3249.9
9515008                     3279.8
10563584                    3302.5
11612160                    3315.5
```

```
12660736                3325.3
13709312                3344.7
14757888                3354.7
15806464                3364.8
16855040                3368.8
18952192                3368.4
21049344                3379.0
23146496                3389.1
25243648                3397.8
27340800                3403.7
29437952                3405.7
31535104                3142.1
33632256                3139.3
37826560                3146.9
42020864                3150.6
46215168                3155.8
50409472                3158.7
54603776                3163.6
58798080                3165.0
62992384                3167.9
67186688                3169.1

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results
↪   may vary when GPU Boost is enabled.
```

This output would be stored in a spreadsheet. After repeating the process with all the nodes, we were able to create different charts to represent the results.

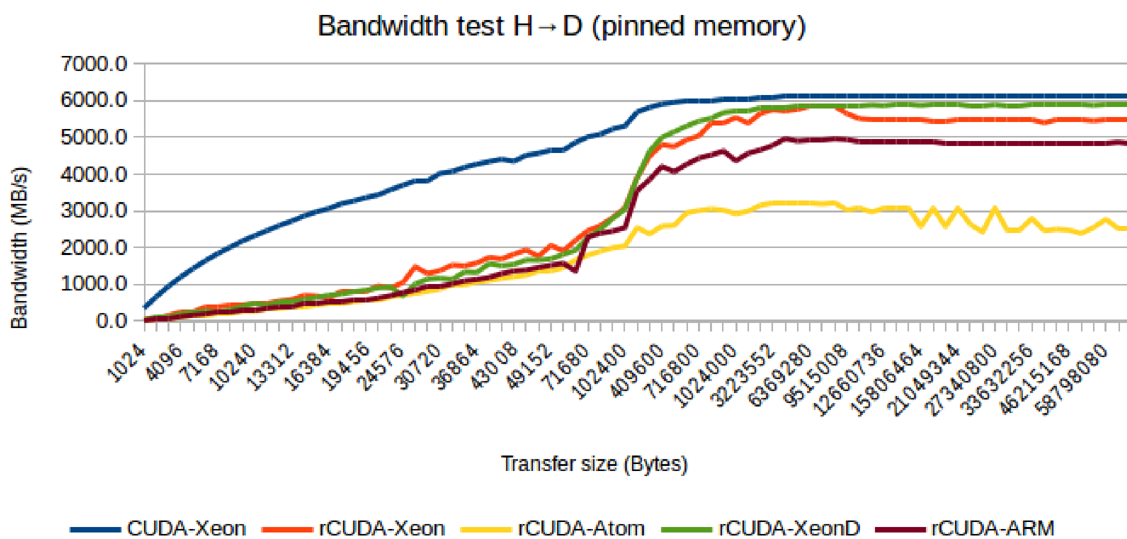

Figure 4.4: CUDA bandwidth test results from host to device with pinned memory. Higher is better.

In Figure 4.4, all the systems with rCUDA show a slow starting growth until 102400 bytes transfer size, then they rise up to their peak bandwidth and stay stable near those values until the end. The CUDA-Xeon (using CUDA instead of rCUDA), on the contrary,

has a much greater growth from the very start and there is barely any growth step at 102400 bytes transfer size.

The CUDA-Xeon leads the chart with a peak bandwidth of 6100 MB/s, followed by the XeonD with 5900 MB/s which has scored higher bandwidth than the 5500 MB/s from rCUDA-Xeon. The next spot would be for the ARM with a peak bandwidth of 4800 MB/s. The Atom stays at the lowest position, scoring 3200 MB/s bandwidth peak as it is the only system using PCIe 2.0.

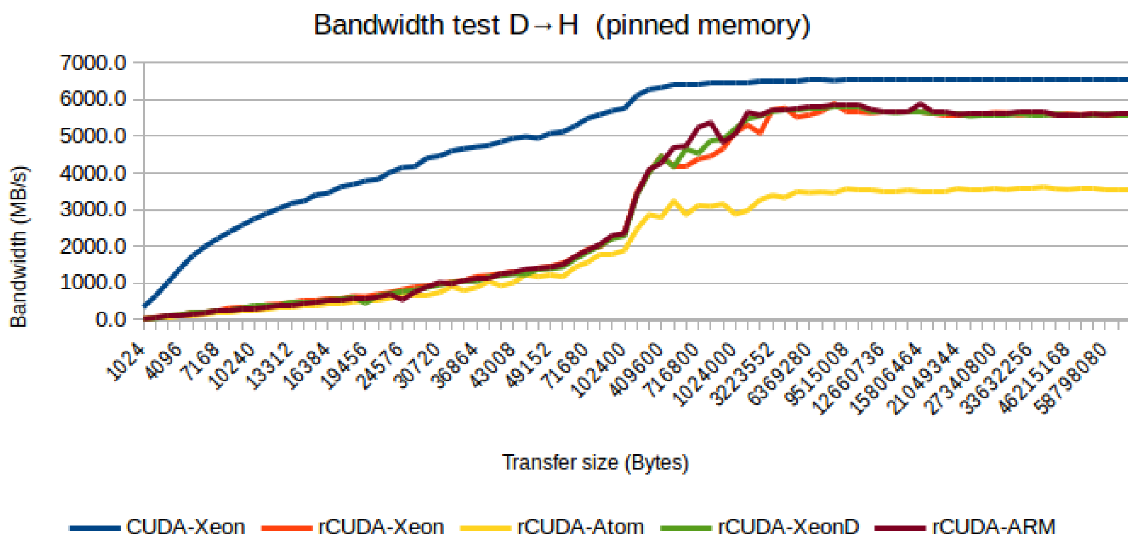

Figure 4.5: CUDA bandwidth test results from device to host with pinned memory. Higher is better.

Figure 4.5 resembles Figure 4.4: slow start for rCUDA systems while CUDA-Xeon grows swiftly to 6500 MB/s. The rCUDA systems also speed up their growth rate from 102400 transfer size, but this time all of them stay at 5600 MB/s. All but the Atom, which stays at 3500 MB/s.

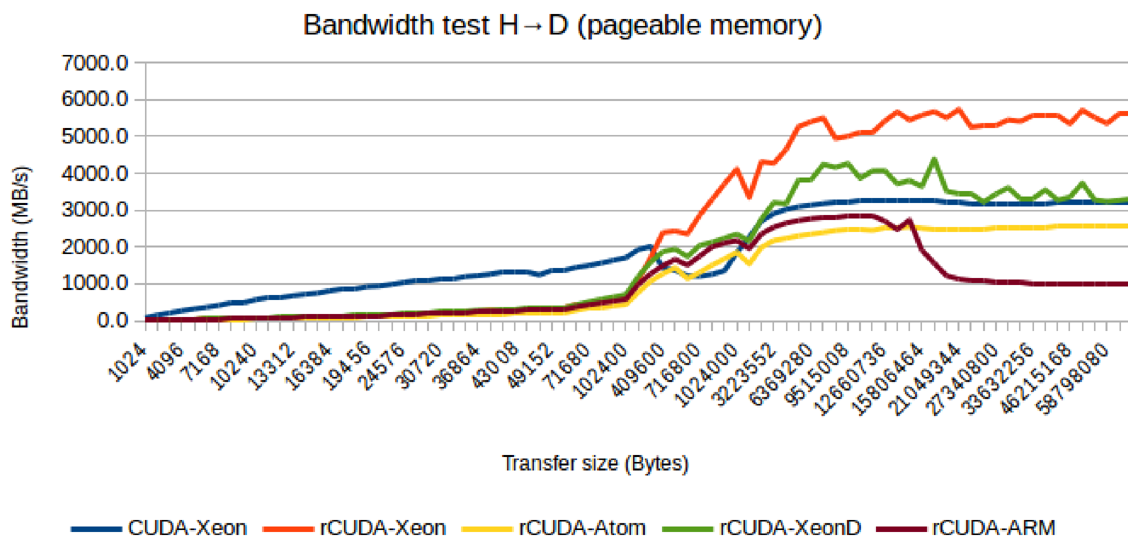

Figure 4.6: CUDA bandwidth test results from host to device with pageable memory. Higher is better.

With pageable memory, shown in Figure 4.6 CUDA-Xeon starts faster too, but the gap with rCUDA is at most 900 MB/s. The main difference with pinned memory is that it is not leading anymore, the rCUDA-Xeon is and with the stunning score of 5700 MB/s, surpassing the XeonD by 1500 MB/s and the CUDA-Xeon by 2400 MB/s. The Atom reached 2500 MB/s. The most interesting case is shown with the ARM. It starts with the same growth as other rCUDA devices, but at around 12660736 bytes transfer size instead of increasing it starts decreasing down to 1000 MB/s.

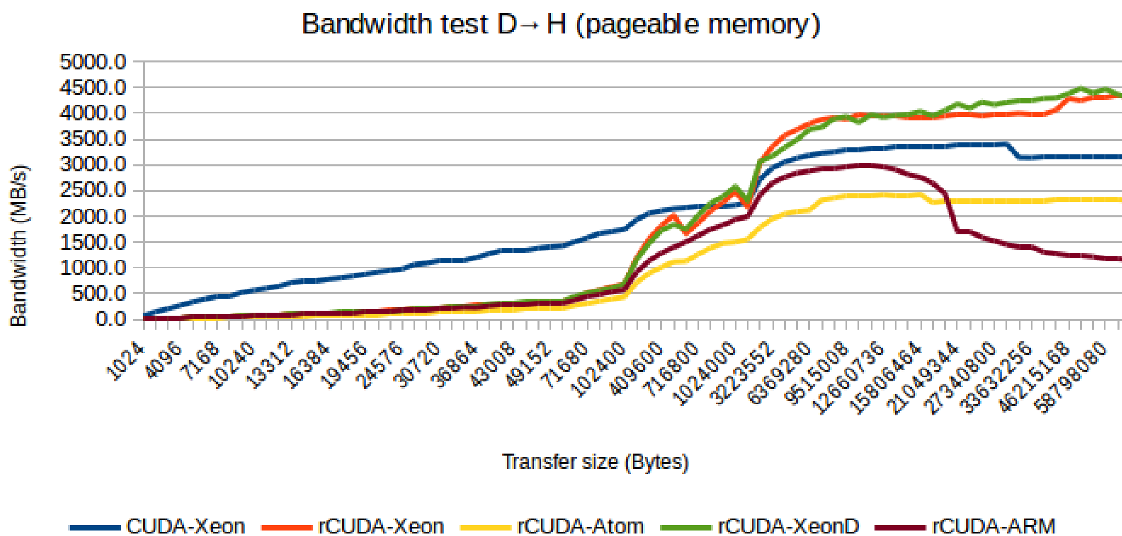

Figure 4.7: CUDA bandwidth test results from device to host with pageable memory. Higher is better.

Figure 4.7 is similar to Figure 4.6, but in this one, the XeonD scored the highest with 4500 MB/s peak bandwidth. The rCUDA-Xeon reaches 4350 MB/s, more than 1000 MB/s over CUDA-Xeon. The Atom continues on its usual scores, peaking 2400 MB/s this time. The ARM maintains the unusual behaviour: has an average rCUDA start with potential to surpass CUDA-Xeon's peak bandwidth, but instead it suddenly decreases to 1150 MB/s.

4.4 LAMMPS

In this section, we present our experience with the LAMMPS application. To that end, we first show how to install the application in both Intel and ARM based systems. Later, we present how to execute it and, finally, we present the execution results.

4.4.1 Installation

Before executing the application we need to compile it to create the required binary files. First, we need to download the application's tarball, which can be freely downloaded from the author's website. There is available a stable version, development version and older Fortran version of LAMMPS. We chose the stable version as we are not going to modify any source code.

The contents from the tarball can be extracted with the command:

```
$ tar zxf lammps-stable.tar.gz
```

After executing this command, the contents of the file will be extracted in a new folder named *lammps-31Mar17*, although the suffix might change depending on the stable version downloaded.

LAMMPS has different versions for distributed computing (using the MPI library) and single node computing (MPI library is not required). Even though we are using two nodes in some of our test configurations, the communication between them is handled by rCUDA, therefore we chose to use the single node version: *lmp_serial*.

To start compiling, we have to access the *src* folder, which includes the source code and the makefiles. For a default installation, we would just need to execute the command:

```
$ make serial
```

However, as we also need to cross compile the application for the ARM architecture. To achieve this, we need to compile the application a second time after changing the necessary parameters to use the necessary tools from the `cuda-cross-aarch64` cross compiling package. The file that needs to be modified is located inside the *MAKE* folder and it is called *Makefile.serial*. The following lines need to be changed in order to properly cross compile this application:

Default compilation:
```
CC = g++
LINK = g++
```

ARMv8 cross compilation:
```
CC = aarch64-linux-gnu-g++
LINK = aarch64-linux-gnu-g++
```

Once the change is done, we can execute the make command again to compile the application for ARM.

After the compilation succeeds, the binary file *lmp_serial* would be created. This file contains the whole application, it can be executed right away to start making the tests.

4.4.2 Execution

For this test, we have used as input the data sets that come with the source files in the *bench* folder. For a basic test run, the benchmark can be executed with a simple instruction:

```
$ ./lmp_serial < in.lj
$ ./lmp_serial < in.eam
$ ./lmp_serial < in.chain
```

Unfortunately, those runs are too small to use as benchmarks, therefore we scaled up the problem to test the systems with bigger workloads. The scaling system works as a three-dimensional grid (length variables x, y and z) that can be set as parameters. For the CHAIN data set exists a scaled version that helps to increase the problem size.

In this test we have used cubic problem scaling, meaning if P is the problem size: $P = x * y * z = x^3$ because $x = y = z$. Different values could be set to optimize the workload distribution in parallel configurations.

To ease the testing process, we automated the scaling from the benchmarks by creating a *for* loop. These are the commands used to execute the different problems:

```
$ for i in {1..5}; do ./lmp_serial -sf cuda -in in.lj -var x $i -var y $i
↪ -var z $i; done
$ for i in {1..5}; do ./lmp_serial -sf cuda -in in.eam -var x $i -var y $i
↪ -var z $i; done
$ for i in {1..5}; do ./lmp_serial -sf cuda -in in.chain.scaled -var x $i
↪ -var y $i -var z $i; done
```

After the execution of every instruction we collected the data relative to the benchmark and stored it. Once every node had already executed the three loops and all the data was gathered, we would transfer it to a spreadsheet and make charts according to the obtained results.

4.4.3 Performance results

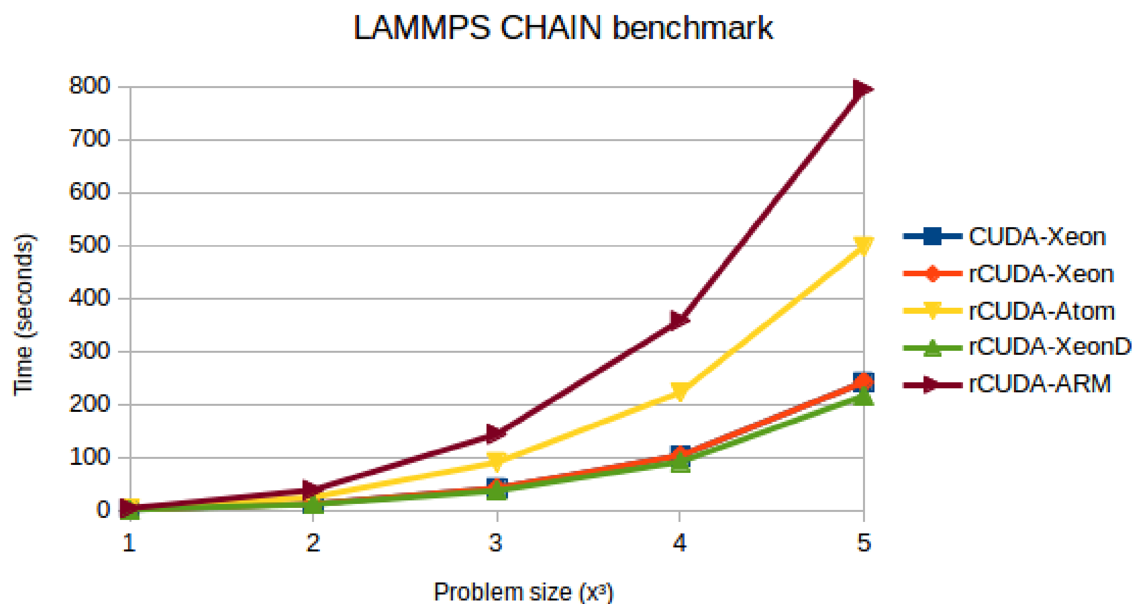

Figure 4.8: LAMMPS CHAIN benchmark results. Lower is better.

In Figure 4.8 we can observe that, for the small sizes, the difference between the several configurations is not necessarily big. They start to differentiate in the later stages when the problem is of considerable size.

In the end, ARM got the lowest score, even worse than the Atom. The list is headed by the XeonD followed at a very close distance by the Xeon, with almost identical timings with rCUDA and CUDA through every problem size.

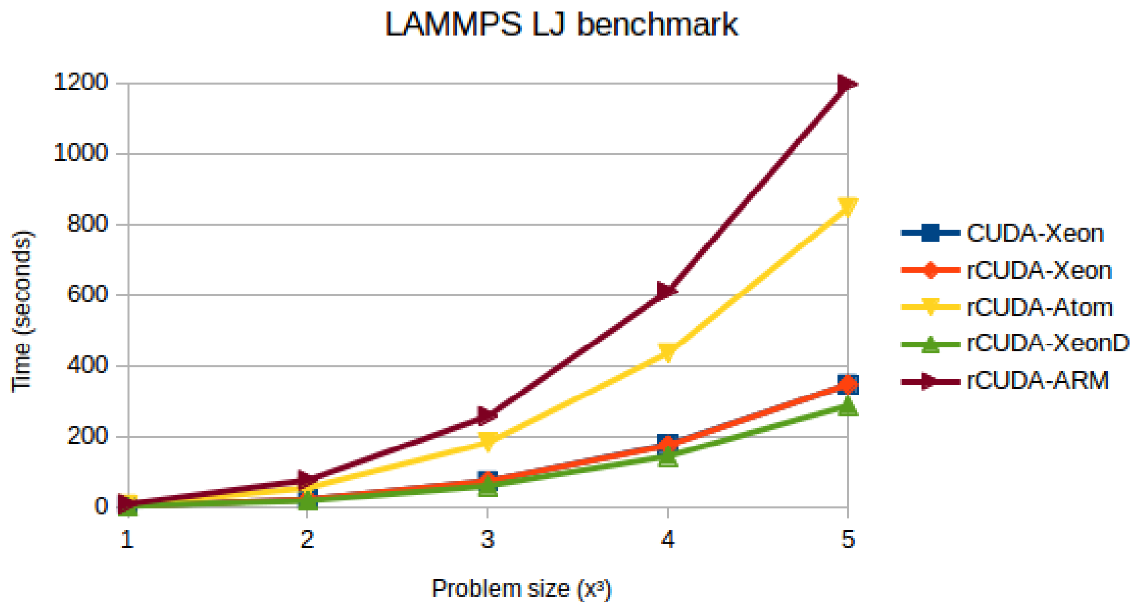

Figure 4.9: LAMMPS LJ benchmark results. Lower is better.

Figure 4.9 shows almost identical results as the previous one: little difference in the small sizes and slow results for the ARM and Atom while the Xeons scored nearly the same time.

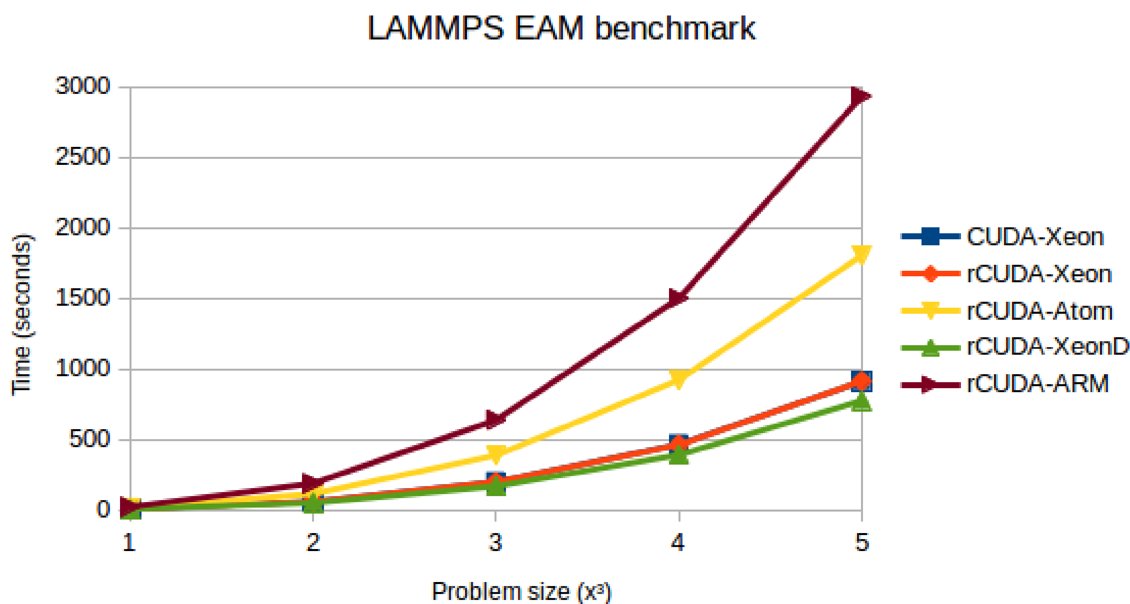

Figure 4.10: LAMMPS EAM benchmark results. Lower is better.

The EAM benchmark is the toughest one from the three, yet the results are virtually the same as in Figures 4.8 and 4.9. The Atom cuts some space from the Xeons, as now it is less than double their score, but the ARM keeps the distance by being more than three times slower than them.

4.5 CUDA-MEME

The CUDA-MEME application is studied in this section. To that end, as in the case of the LAMMPS application, we first present the installation and execution procedures and we finally show the performance result.

4.5.1 Installation

To install this application, first we had to download the application's tarball, which can be freely downloaded from the author's website. As of today, the latest version of CUDA-MEME is 3.0.16, therefore we downloaded the file *cuda-meme-3.0.16.tar.gz*.

The contents from the tarball can be extracted with the command:

```
$ tar zxf cuda-meme-3.0.16.tar.gz
```

After executing this command, the contents of the file will be extracted in a new folder named *cuda-meme-3.0.16*.

Before continuing any further, we have to make sure we fulfil the CUDA-MEME requirements stated in the *README* file included in the package:

```
1) Install CUDA 2.0 or higher SDK and Toolkits
2) Install MPICH/MPICH2. If using Makefile.gpu, do not need to install MPI
 ↪  library.
3) If the tool "convert" (installed in /usr/bin/) that changes EPS to PNG
 ↪  format is not installed
in your system, you might need to download and install ImageMagick first
 ↪  (http://www.imagemagick.org/script/download.php).
You can change the config.h file in the src directory to specify an
 ↪  alternative tool by changing the value of macro "CONVERT_PATH", and
 ↪  then recompiling the code. If ths converting tool does not exist, you
 ↪  need manually convert the ESP files in the output directory (default
 ↪  meme_out) to PNG files.
```

First, to know if our CUDA Toolkit would be compatible, we checked the version we have installed, which is stated in a file named *version.txt* in its installation folder:

```
$ cat /usr/local/cuda/version.txt
CUDA Version 8.0.61
```

In our case, the CUDA version installed is higher than the one required by CUDA-MEME, therefore we can continue. As in this work we are not distributing the workload between nodes but using one node at a time with GPU virtualization, MPICH/MPICH2 would not be required. We skipped this step. Finally, for the last requirement, we need to check if we have the *convert* tool installed:

```
$ which convert
/usr/bin/convert
```

If the *which* command returns a value it means that the convert tool is in scope. It is possible that the tool was installed but the command did not return the tool path. This could mean that the executable file is in a folder not included in the PATH environment

variable and it must be. Once we ensured that our system satisfies the CUDA-MEME requirements, we can proceed to the installation.

We can observe that in the installation folder there are two different makefiles:

- **Makefile.gpu** used to compile CUDA-MEME.
- **Makefile.mgpu** used to compile mCUDA-MEME. Uses the MPI library previously mentioned in the requirements.

Therefore, we are going to use *Makefile.gpu*. As we have to compile the application for two different architectures, we might need to make modifications to Makefile.gpu so it can compile properly. First, we need to take a look at the contents of the makefile:

```
all:
	-mkdir objs
	make -C src -f Makefile.gpu
	strip src/cuda-meme
	mv src/cuda-meme .

clean:
	make -C src -f Makefile.gpu clean
	-rm -rf cuda-meme
	-rm -rf mcuda-meme
```

In this makefile, two paths have been defined: *all* (compilation) and *clean* (remove all the files created in the compilation process). It can be seen that it delegates in another makefile located in the *src* folder. The default compilation should work for our Intel-based systems. However, the second compilation was a cross compilation for the ARM architecture. Thus, the strip command had to be replaced with its respective tool from the `cuda-cross-aarch64` package:

Default compilation:	`strip src/cuda-meme`
ARMv8 cross compilation:	`aarch64-linux-gnu-strip src/cuda-meme`

This second makefile is longer and more complex, but there are some parts that we needed to review:

```
[...]
NVCC = /usr/local/cuda/bin/nvcc
DEVICE_FLAGS = -arch sm_30  --ptxas-options=-v -Xcompiler -fopenmp
[...]
NVCCFLAGS += -cudart=shared
#NVCCFLAGS += -ccbin=/usr/bin/aarch64-linux-gnu-g++
[...]
```

In these last lines, we need to set our actual *nvcc* path. In *DEVICE_FLAGS*, as we are using a Kepler GPU, we set the `-arch sm_30` flag. Since we are going to use this application with rCUDA, it is mandatory to use the *nvcc* flag `-cudart=shared`. The second NVCCFLAG we added should only be uncommented in the second compilation when cross compiling for the ARM architecture.

The current directory, *src*, has two additional subfolders which are libraries included in the CUDA-MEME installation: *libxml2* and *libxslt*. These two libraries, when not cross

compiling, do not need modification. As we are compiling for two different processor architectures, we had to modify one line from each makefile from inside every library's folder when we had to cross compile the application:

Default compilation: `CC = gcc`
ARMv8 cross compilation: `CC = aarch64-linux-gnu-gcc`

After modifying all the necessary makefiles, we should already be able to compile the application without problems. For that end, we need to go back to the root of the installation folder and run the make command from there:

```
$ make -f Makefile.gpu
```

If the compilation succeeded, an executable file named *cuda-meme* would have been created in the same folder. This file is all we need to run the application.

4.5.2 Execution

We are using the DNA alphabet for these data sets and the *one occurrence per sequence* (OOPS) model which permits exactly one motif occurrence in each sequence. A more precise description of the parameters is given when executing the program like `./cuda-meme -h`:

```
USAGE:
cuda-meme       <data set> [optional arguments]

<data set>              file containing sequences in FASTA format
[...]
[-dna]                  sequences use DNA alphabet
[...]
[-mod oops|zoops|anr]   distribution of motifs
[...]
[-maxsize <maxsize>]    maximum data set size in characters
[...]
```

The CUDA-MEME package comprises three FASTA files that we used as benchmarks. Every benchmark was run once in each system configuration. Afterwards, when the execution of a benchmark finishes, the timing data is saved separately from each different test and gathered in a spreadsheet to generate the charts later.

Thus, the benchmark execution looked like this:

```
$ ./cuda-meme ./nrsf_500.fasta -dna -mod oops -maxsize 500000
$ ./cuda-meme ./nrsf_1000.fasta -dna -mod oops -maxsize 1000000
$ ./cuda-meme ./nrsf_2000.fasta -dna -mod oops -maxsize 2000000
```

4.5.3 Performance results

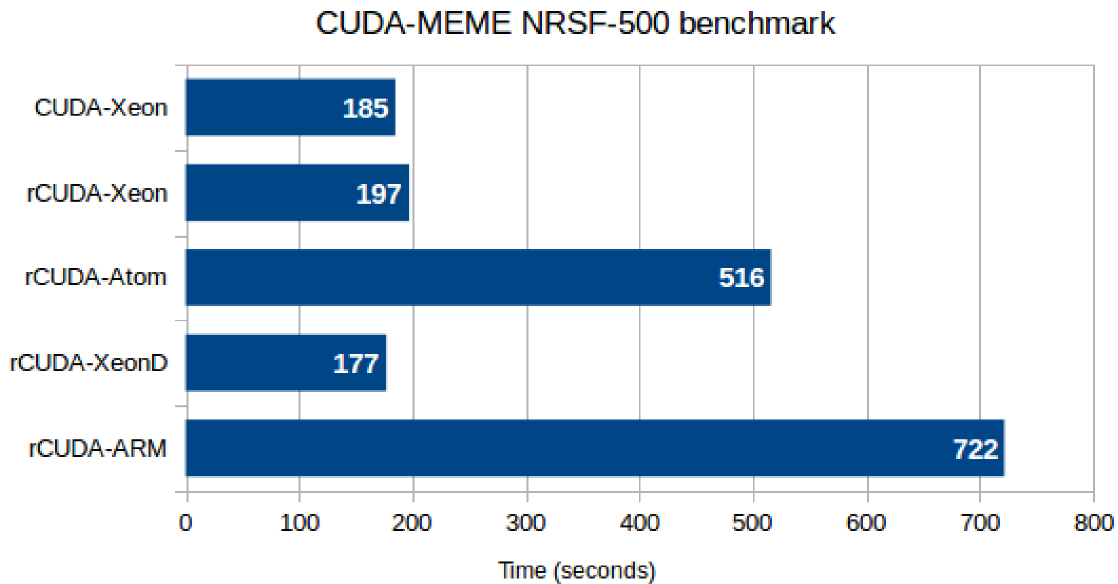

Figure 4.11: CUDA-MEME NRSF-500 benchmark results. Lower is better.

Figure 4.11 shows that in this application the Xeons are leading again. The first position is taken by the XeonD, followed by the Xeon with CUDA, which is 6.5% faster than with rCUDA. The Atom is far from them: it benchmarked two and a half times slower than the Xeon, whereas the ARM scored three and a half times slower than the Xeon as well.

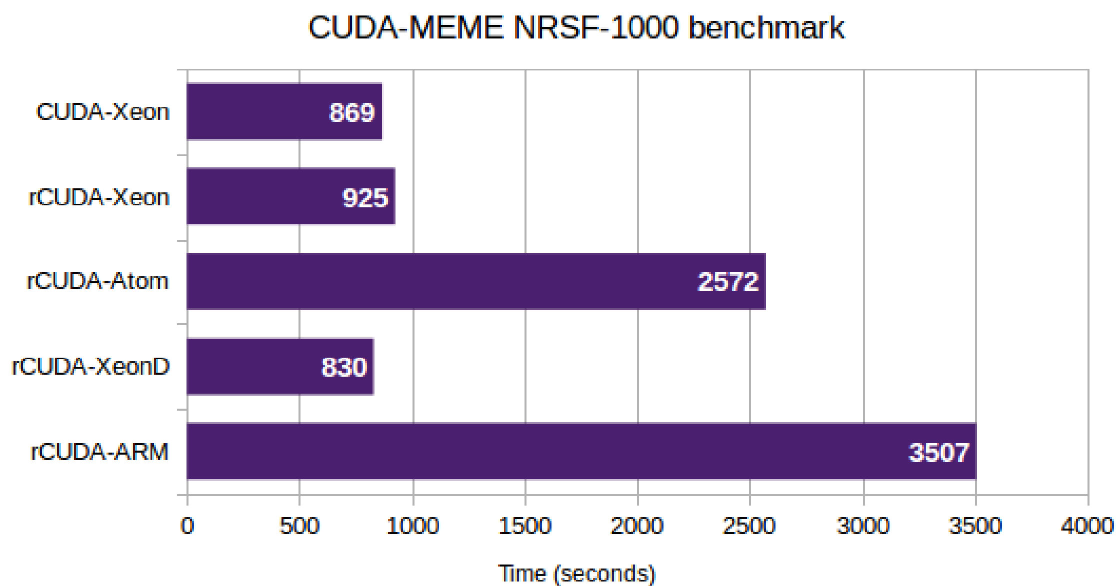

Figure 4.12: CUDA-MEME NRSF-1000 benchmark results. Lower is better.

Figure 4.12 shows very similar results compared to Figure 4.11. The XeonD has a small lead in this chart, followed by the Xeon with CUDA that is still faster than with rCUDA

by 6.4%. The Atom and ARM are at the bottom of the list, being significantly slower than the others.

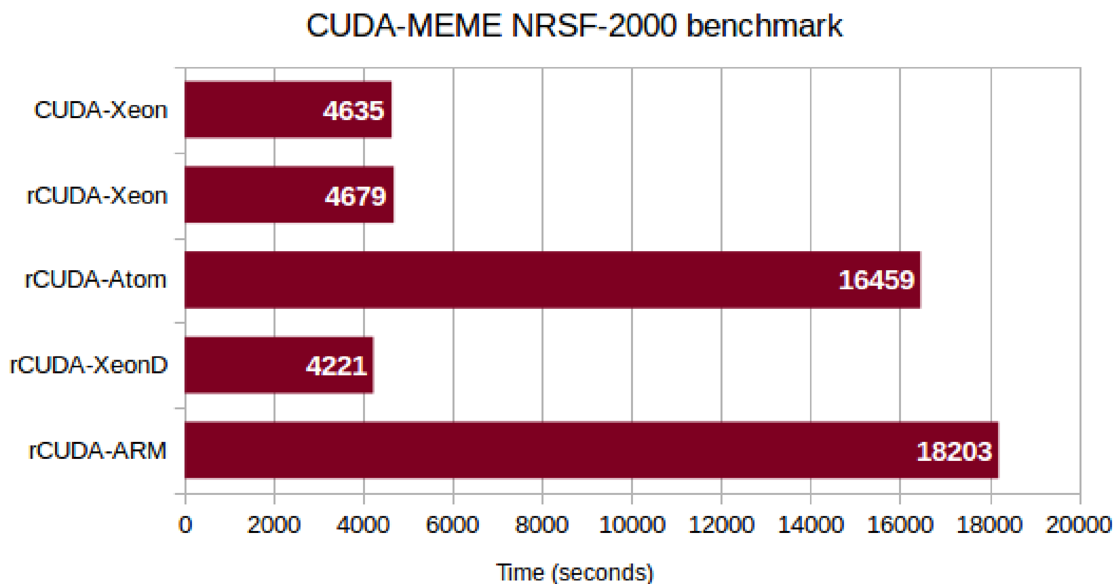

Figure 4.13: CUDA-MEME NRSF-2000 benchmark results. Lower is better.

The chart in Figure 4.13 continues with the expected results, except that the difference between rCUDA and CUDA from the Xeon became less significant ($< 1\%$). The XeonD also separates from the Xeon by 9.8%, while the slowest are again the Atom and the ARM by 255% and 293%, respectively, compared to the Xeon.

4.6 NAMD

In this last section we present the NAMD application. To that end, as with the previous applications, we first present how to install and execute the NAMD application and later show the performance results.

4.6.1 Installation

Despite the effort we put into the realization of this test, we were not able to make the executable to work under the ARM architecture. Therefore, this system has not been included in the charts or the installation instruction as we were unable to compile it.

First of all, we had to download the NAMD tarball in our system, which can be retrieved from the author's website for free. We downloaded the latest version: 2.12. To access the content we need to extract the files first, which can be achieved by executing the command:

```
$ tar zxf NAMD_2.12_Source.tar.gz
```

This would extract all the contents from the tarball into a new folder named NAMD_2.12_Source.

In this application we also need the TCL and FFTW libraries, which are not included in the tarball. These libraries must be downloaded separately. The University of Illinois at Urbana-Champaign has uploaded different tested versions from the TCL and FFTW

libraries on their server. To set up these libraries in our installation folder we need to execute the following commands:

```
$ wget
↪  http://www.ks.uiuc.edu/Research/namd/libraries/fftw-linux-x86_64.tar.gz
$ wget
↪  http://www.ks.uiuc.edu/Research/namd/libraries/tcl8.5.9-linux-x86_64.tar.gz
$ wget
↪  http://www.ks.uiuc.edu/Research/namd/libraries/tcl8.5.9-linux-x86_64-threaded.tar.gz
$ tar xzf tcl8.5.9-linux-x86_64.tar.gz
$ tar xzf tcl8.5.9-linux-x86_64-threaded.tar.gz
$ tar xzf fftw-linux-x86_64.tar.gz
$ mv tcl8.5.9-linux-x86_64 tcl
$ mv tcl8.5.9-linux-x86_64-threaded tcl-threaded
$ mv linux-x86_64 fftw
```

The first three commands download the library tarballs. The next three commands extract the files from the tarballs. The last three commands rename the created folders with the names that NAMD recognizes.

After the libraries are prepared, we can proceed to the next step: the compilation of Charm++. As NAMD is written using the Charm++ parallel programming model, we have to prepare the Charm++ binaries first before compiling NAMD. Charm++ source files are provided in the NAMD package in another tarball. The first thing to do is to extract it:

```
$ tar xf charm-6.7.1.tar
```

Once extracted, we access the folder and start building Charm++ with the following command:

```
$ ./build charm++ multicore-linux64
```

The option *multicore-linux64* specifies to the compiler that we are running the application in a single node with 64 bits Linux operating system. Therefore, MPI library is not required. This execution could be optimized by adding the `--with-production -j<N>` parameters, replacing the <N> with the number of threads that will be created to parallelize the compilation process. The *–with-production* parameter applies production build optimizations to the application, which is of our interest since we are not going to debug anything. After these optimizations the command would look like this:

```
$ ./build charm++ multicore-linux64 --with-production -j8
```

However, the building script has its own step-by-step tutorial on how to compile Charm++ according to the user's needs. To build Charm++ this way, the script has to be run without any parameter, and then the user has to answer the questions presented on the terminal. When the compilation process is successfully finished, the *charm++* binary would be created in the same folder.

The next step would be the NAMD compilation configuration. Exiting the Charm++ folder we would be in the main NAMD directory, which contains another assistant to configure how we would like to compile NAMD according to our needs. The assistant file is named *config* and it is a script from the *C shell*. Thus, might not be supported by the user's shell. The best alternative was to install *tcsh* (a Unix shell based on csh), as in the system we used for compiling the applications we had *bash* as the default shell. Once we had a compatible shell installed, we ran the *config* file as follows:

```
$ csh ./config Linux-x86_64-g++ --charm-arch multicore-linux64 --with-cuda
↪ --cuda-prefix /usr/local/cuda
```

Executing this command just creates the configuration parameters for the makefile in a different folder named as the NAMD target architecture: *Linux-x86_64-g++*. Notice that the `--charm-arch multicore-linux64` parameter is the same as the Charm++ previously compiler. It is also worth mentioning that without the `--with-cuda` flag, NAMD would make the computations in the CPU, even if the system has a GPU connected.

We need to add the shared cudart flag necessary by rCUDA. To achieve it, we navigate to the *arch* folder, which contains the makefiles from different available system architecture. There, we need to locate the CUDA configuration from our architecture: *Linux-x86_64.cuda*. This line has to be added at the end of the file:

```
CUDACC += -cudart=shared
```

At this point, we just need to change to that new directory and start the compilation process:

```
$ cd Linux-x86_64-g++
$ make -j8
```

When the compilation process finishes, it would create different binary files, but we are just going to use the one named *namd2*.

4.6.2 Execution

In the University of Illinois at Urbana-Champaign website, there are available different utilities related to NAMD. These utilities include related packages that can be used to extend NAMD functionality, scripts to ease different tasks and example simulations. In this last category, we can find the simulations that are commonly used as benchmarks. We downloaded the benchmarks we selected for the performance analysis and placed them in the same folder as the NAMD binary file.

We have sorted the three tests from smallest to the largest: ApoA1, ATPase and STMV.

The execution of *namd2* can be customized to balance the workload, tune it for specific networks and many other parameters. As we do not need any from the offered settings, we have just run the benchmarks out of the box:

```
$ ./namd2 ./apoa1/apoa1.namd
$ ./namd2 ./f1atpase/f1atpase.namd
$ ./namd2 ./stmv/stmv.namd
```

After every command execution, we saved the application output and timing. Once we collected the data from the execution of every benchmark in every node, we gathered the results in a spreadsheet to be able to easily create different charts.

4.6.3 Performance results

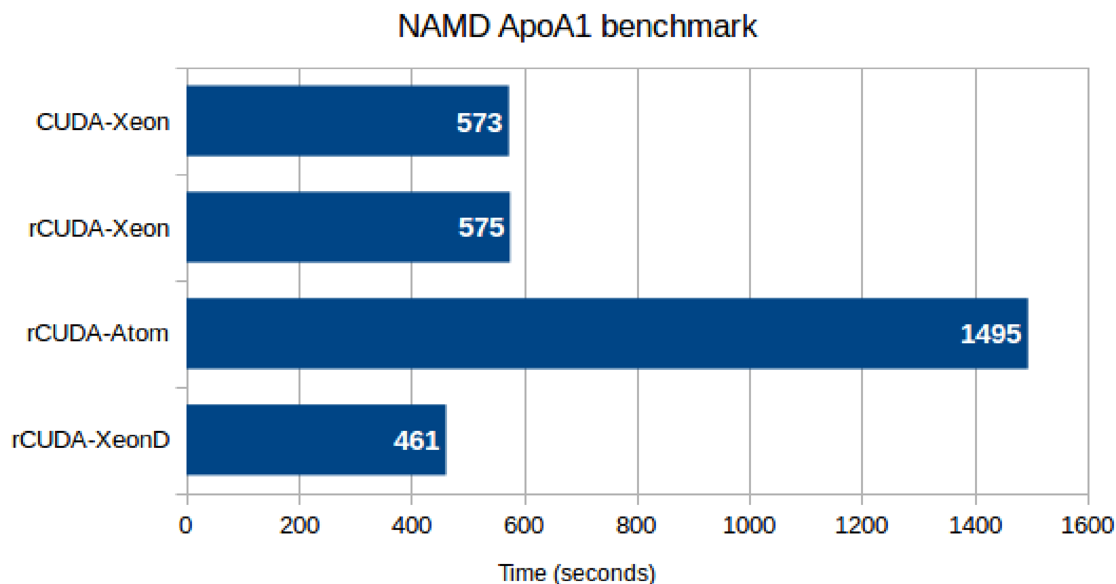

Figure 4.14: NAMD ApoA1 benchmark results. Lower is better.

Figure 4.14 shows the XeonD leading once more. The Xeon follows. Looking at CUDA-Xeon and rCUDA-Xeon can be observed that the difference between CUDA and rCUDA is insignificant. The Atom is the last one by a large margin.

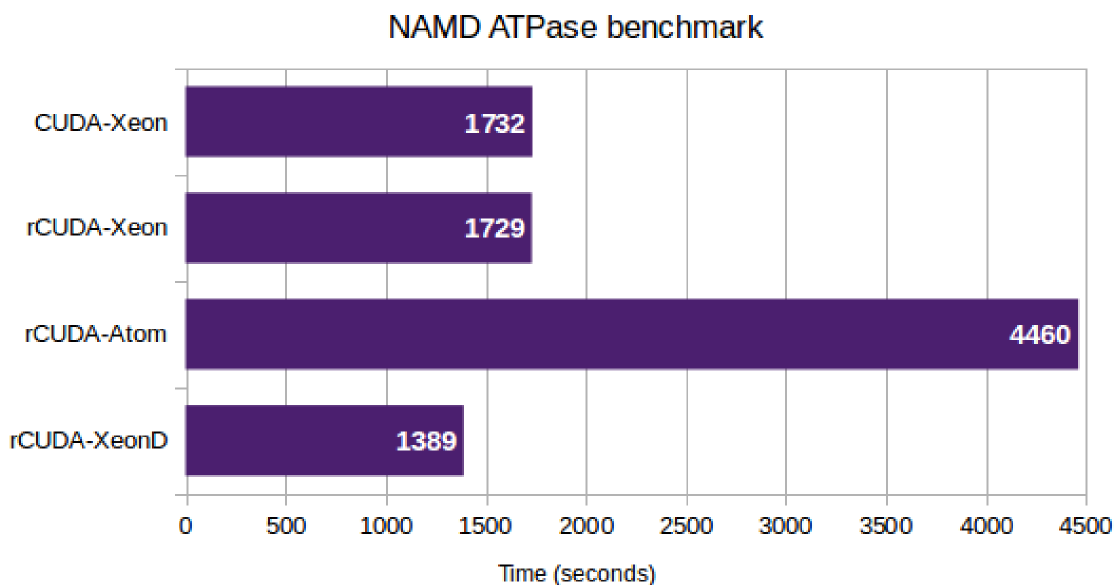

Figure 4.15: NAMD ATPase benchmark results. Lower is better.

Figure 4.15 is lead by the XeonD too. It is followed by the Xeon and, once again, showing the almost null overhead from rCUDA in this application. The Atom is very distant from the Xeons again.

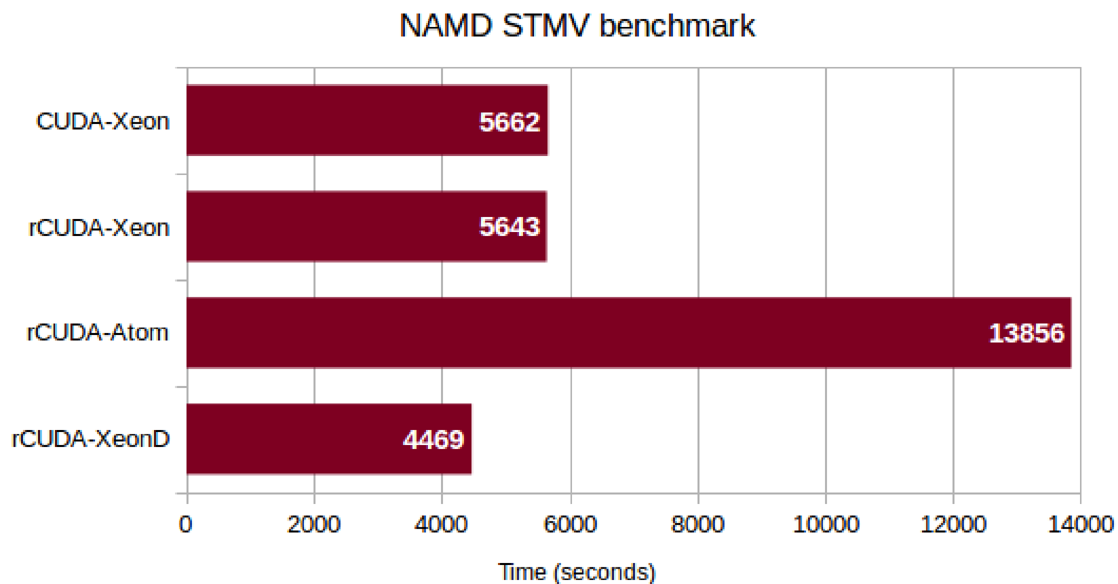

Figure 4.16: NAMD STMV benchmark results. Lower is better.

In Figure 4.16 we can see that the XeonD is still leading this last test, keeping a slight difference with both Xeon configurations: with rCUDA and CUDA. Atom is still the slowest with a time three times slower than the XeonD.
Worth mentioning that when the size of the problem increases, the difference between rCUDA and CUDA decreases (in a very small scale) and, at some point, the performance numbers favor rCUDA: in the ApoA1 benchmark, CUDA was faster by 0.0035%; in the ATPase benchmark, rCUDA surpassed CUDA's time by 0.0017%; in the STMV benchmark, rCUDA is still faster and with double the difference as in the ATPase benchmark: 0.0034%.

CHAPTER 5
Conclusions

In this thesis, we measured the performance of both physical (CUDA) and virtual (rCUDA) GPUs with scientific applications on systems with x86-64 and ARMv8 CPUs architectures and different levels of power consumption.

The goal of this study was to analyze the feasibility of using low-power processor based systems to run CUDA-accelerated applications. In this regard, given that this kind of systems usually lack the required PCIe lanes to connect a GPU and a high-performance network adapter, only the latter is installed in these systems and therefore the CUDA acceleration was provided by means of a remote GPU virtualization such as rCUDA. Notice, however, that in our study we have only considered performance results, but not energy consumption numbers due to the temporal limitations of the study carried out. Nevertheless, in order to perform a thorough study, power measurements should be gathered.

In general, it was a tedious task to test different processor architectures. The main reason was that CUDA 8.0 natively supports only x86-64 and POWER8, while ARMv7 and ARMv8 just have some library support for cross compilation. This has made the testing process troublesome to the point that some tests were unable to be performed in the ARM system, leaving this system not as exhaustively tested as the others.

We tested the InfiniBand network to observe how the different systems' connection perform, and we could see that the Atom is hardware-limited since the very first test. Using PCIe v2.0 severely affects bandwidth, which means that it would have around half the bandwidth of the other systems at most.

Afterwards, we tested the GPU capabilities by measuring its bandwidth with a simple test included in the CUDA toolkit. It demonstrated that computing with pinned memory without any middleware is the best option, as sparing the communications provides the highest bandwidth. However, pinned memory is very situational and in general should not be used by applications. Thus, pageable memory would be the ideal case given that the workload might take too many resources from the system if pinned memory were to be used. The results with pageable memory are very positive for rCUDA, as the Xeon and XeonD present higher bandwidth than with just CUDA.

ARM is not fully supported by rCUDA either. It can be spotted when, at some point, instead of increasing or stabilizing the bandwidth to the remote GPU, it unexpectedly decreases. The results are low, even lower than the Atom in some cases. After knowing that its GPU bandwidth is strongly limited, its other tests are expected to return lacklustre outcomes.

According to the previous paragraphs and the results from the scientific applications, we created an opinion about every system we studied:

- **XeonD:** outstanding performance. This is due to faster memory (2133 MHz versus 1600 MHz from the Xeon). It makes such a big difference that is even faster than the Xeon running programs locally. The price is $173 higher than the Xeon, but this an insignificant amount for data centres. Consuming only 40 W, and everything previously said, makes it the most solid candidate to be our CPU choice for GPU virtualization servers.

- **Xeon:** strong performance too. Scores high in all the results and somewhat close to the XeonD, but consuming 80 W places this CPU as our second choice.

- **Atom:** decent performance for what it is. It cannot be compared with the Xeons in any aspect, as it has slower PCIe, slower processor frequency, fewer cores and several other details that go against it. The upside is the 20 W that it consumes, which would make it a good choice for a low-budget low-end server or for storage, but the performance that a modern data centre expects is definitely higher than what the Atom can offer.

- **ARM:** poor performance when it was able to work. The CUDA and rCUDA support for ARM are flawed, making it not completely reliable even if its specifications are powerful. As of today, this would be our last option.

Despite the good results of the XeonD processor, it would not be an option if the purpose of acquiring it is to renew an older CPU. It is a SoC, meaning that it cannot replace or be replaced easily. Therefore, it would only be a valid option if a data centre needs to extend some system or to build a new one.

At the personal level, the process of making this thesis has positively influenced me. I have gained extensive experience in the GNU/Linux operating system, which is an essential skill on any computer engineer. Now, I would be able to work comfortably and fluently in a GNU/Linux environment if it were to be the workplace designated operating system.

It also helped me reach a deeper understanding of computer hardware, being able to recognize the different capabilities from a hardware specification and having the means for pointing out its strengths and weaknesses. This would help to judge hardware accordingly when acquiring new systems, both professionally and recreationally.

The process of making this thesis, and the whole degree study in general, has made me a more mature person, helping me develop a prepared mindset to start a professional career and to keep expanding my knowledge about technology.

Bibliography

[1] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili, "Red fox: an execution environment for relational query processing on gpus", in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, ACM, 2014, 44:44–44:54, ISBN: 978-1-4503-2670-4.

[2] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, "Rapid aerodynamic performance prediction on a cluster of graphics processing units", in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, Orlando, FL, Jan. 2009.

[3] D. P. Playne and K. A. Hawick, "Data parallel three-dimensional cahn-hilliard field equation simulation on gpus with cuda", in *PDPTA*, 2009, pp. 104–110.

[4] I. Yamazaki, T. Dong, R. Solca, S. Tomov, J. Dongarra, and T. Schulthess, "Tridiagonalization of a dense symmetric matrix on multiple gpus and its application to symmetric eigenvalue problems", *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2652–2666, 2014, ISSN: 1532-0634.

[5] D. Yuancheng Luo, "Canny edge detection on nvidia cuda", in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, IEEE, 2008, pp. 1–8, ISBN: 978-1-4244-2339-2.

[6] V. Surkov, "Parallel option pricing with fourier space time-stepping method on graphics processing units", *Parallel Computing*, vol. 36, no. 7, pp. 372 –380, 2010, Parallel and Distributed Computing in Finance.

[7] P. K. Agarwal, S. Hampton, J. Poznanovic, A. Ramanthan, S. R. Alam, and P. S. Crozier, "Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures", *Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, pp. 1356–1375, 2013.

[8] G.-H. Luo, S.-K. Huang, Y.-S. Chang, and S.-M. Yuan, "A parallel bees algorithm implementation on GPU", *Journal of Systems Architecture*, vol. 60, no. 3, pp. 271 –279, 2014.

[9] Netlib. *From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming.* Available: http://www.netlib.org/lapack/lawnspdf/lawn228.pdf. Accessed 21-08-2017.

[10] HPCwire. *Why 2016 is the most important year in hpc in over two decades.* Available: https://www.hpcwire.com/2016/08/23/2016-important-year-hpc-two-decades/. Acessed 31-07-2017.

[11] TOP500. *Top500 list.* Available: https://www.top500.org/list/2017/06/. Accessed 21-08-2017.

[12] TOP500. *Green500 list.* Available: https://www.top500.org/green500/list/2017/06/. Accessed 21-08-2017.

[13] NVIDIA. *Nvidia tesla p100 pcie datasheet.* Available: https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf. Accessed 21-08-2017.

[14] TOP500. *The linpack benchmark.* Available: https://www.top500.org/project/linpack/. Accessed 30-08-2017.

[15] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers: Particle-particle particle-mesh", *Computer Physics Communications*, vol. 183, no. 3, pp. 449 –459, 2012.

[16] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: Accelerating smith-waterman protein database search by coupling CPU and GPU SIMD instructions", *BMC Bioinformatics*, vol. 14, no. 1, 2013.

[17] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: Using graphics processors to accelerate protein sequence alignment", *Bioinformatics*, 2010.

[18] Y. Liu, B. Schmidt, W. Liu, and D. L. Maskell, "Cuda-meme: Accelerating motif discovery in biological sequences using CUDA-enabled graphics processing units", *Pattern Recognition Letters*, vol. 31, no. 14, pp. 2170 –2177, 2010.

[19] F. Silla, S. Iserte, C. Reaño, and J. Prades, "On the benefits of the remote gpu virtualization mechanism: The rcuda case", *Concurrency and Computation: Practice and Experience*, vol. 29, no. 13, e4072–n/a, 2017, e4072 cpe.4072, ISSN: 1532-0634. DOI: 10.1002/cpe.4072. Available: http://dx.doi.org/10.1002/cpe.4072.

[20] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and remote gpus perform similar with edr 100g infiniband", in *Proceedings of the Industrial Track of the 16th International Middleware Conference*, ser. Middleware Industry '15, 2015.

[21] C. Reaño and F. Silla, "Tuning remote gpu virtualization for infiniband networks", *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4520–4545, 2016.

[22] C. Reaño and F. Silla, "A performance comparison of cuda remote gpu virtualization frameworks", in *2015 IEEE International Conference on Cluster Computing*, 2015.

[23] *Data center power market:global forecast to 2021.* Available: https://www.researchandmarkets.com/reports/4035520/data-center-power-market-by-solution-power. Accessed 30-08-2017.

[24] Wikipedia. *Intel atom.* Available: https://en.wikipedia.org/wiki/Intel_Atom. Accessed 26-06-2017.

[25] Wikichip. *Intel xeon d.* Available: https://en.wikichip.org/wiki/intel/xeon_d. Accessed 28-06-2017.

[26] CAVIUM. *Cavium thunderx arm processors.* Available: http://www.cavium.com/ThunderX_ARM_Processors.html. Accessed 29-06-2017.

[27] Wikipedia. *Central processing unit.* Available: https://en.wikipedia.org/wiki/Central_processing_unit. Accessed 22-06-2017.

[28] Wikipedia. *Xeon.* Available: https://en.wikipedia.org/wiki/Xeon. Accessed 28-06-2017.

[29] Intel. *Intel xeon cpu e5-2620 v2.* Available: https://ark.intel.com/products/75789. Accessed 23-06-2017.

[30] Intel. *Intel atom cpu z500.* Available: https://ark.intel.com/products/35472. Accessed 23-06-2017.

[31] Intel. *Intel atom cpu c2750.* Available: https://ark.intel.com/products/77987. Accessed 23-06-2017.

[32] Intel. *Intel xeon cpu d-1541.* Available: http://ark.intel.com/products/91199. Accessed 28-06-2017.

[33] CAVIUM. *Thunderx rattles server market.* Available: http://www.cavium.com/pdfFiles/ThunderX_Rattles_Server_Market.pdf. Accessed 29-06-2017.

[34] Wikipedia. *Graphics processing unit.* Available: https://en.wikipedia.org/wiki/Graphics_processing_unit. Accessed 23-06-2017.

[35] NVIDIA. *Tesla technical brief.* Available: http://www.nvidia.com/docs/IO/43395/tesla_technical_brief.pdf. Accessed 29-06-2017.

[36] NVIDIA. *Nvidia tesla k20m datasheet.* Available: https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. Accessed 4-07-2017.

[37] NVIDIA. *Nvidia tesla v100 gpu architecture.* Available: https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf. Accessed 31-08-2017.

[38] Wikipedia. *Cuda.* Available: https://en.wikipedia.org/wiki/CUDA. Accessed 6-07-2017.

[39] NVIDIA. *Language solutions.* Available: https://developer.nvidia.com/language-solutions. Accessed 6-07-2017.

[40] NVIDIA. *Cuda 8 features revealed.* Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/. Accessed 27-08-2017.

[41] Wikipedia. *Opencl.* Available: https://en.wikipedia.org/wiki/OpenCL. Accessed 6-07-2017.

[42] ArrayFire. *Opencl on mobile devices.* Available: https://arrayfire.com/opencl-on-mobile-devices/. Accessed 6-07-2017.

[43] Wikipedia. *Infiniband.* Available: https://en.wikipedia.org/wiki/InfiniBand. Accessed 24-07-2017.

[44] Wikipedia. *Remote direct memory access.* Available: https://en.wikipedia.org/wiki/Remote_direct_memory_access. Accessed 24-07-2017.

[45] Archlinux. *Infiniband.* Available: https://wiki.archlinux.org/index.php/InfiniBand. Accessed 24-07-2017.

[46] Mellanox. *Introducing 200g hdr infiniband solutions.* Available: http://www.mellanox.com/related-docs/whitepapers/WP_Introducing_200G_HDR_InfiniBand_Solutions.pdf. Accessed 24-07-2017.

[47] Wikipedia. *Rdma over converged ethernet.* Available: https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet. Accessed 25-07-2017.

[48] S. N. Laboratories. *Lammps molecular dynamics simulator.* Available: http://lammps.sandia.gov/. Accessed 14-08-2017.

[49] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard", in *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*. 1994, pp. 213–218.

[50] Wikipedia. *Xeon phi.* Available: https://en.wikipedia.org/wiki/Xeon_Phi. Accessed 31-08-2017.

[51] W. M. Brown, P. Wang, P. S. Crozier, and S. Plimpton. *Lammps on gpus.* Available: http://lammps.sandia.gov/workshops/Feb10/Mike_Brown/gpu_tut.pdf. Accessed 31-08-2017.

[52] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with namd", *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005, ISSN: 1096-987X. DOI: 10.1002/jcc.20289. Available: http://dx.doi.org/10.1002/jcc.20289.

[53] U. of Illinois at Urbana-Champaign. *Namd 2.7 release notes*. Available: http://www.ks.uiuc.edu/Research/namd/2.7/notes.html. Accessed 31-08-2017.

[54] S. M. Computer. *Superserver 1027gr-tsf*. Available: https://www.supermicro.com/products/system/1u/1027/sys-1027gr-tsf.cfm. Accessed 31-08-2017.

[55] S. M. Computer. *Superserver 5018d-fn4t*. Available: https://www.supermicro.com/products/system/1u/5018/sys-5018d-fn4t.cfm. Accessed 31-08-2017.

[56] S. M. Computer. *Superserver 5018a-tn4*. Available: https://www.supermicro.com/products/system/1u/5018/sys-5018a-tn4.cfm. Accessed 31-08-2017.

[57] Avantek. *Arm server – cavium thunderx 1u 48 cores r120-t30*. Available: https://www.avantek.co.uk/arm-server-r120-t30/. Accessed 31-08-2017.