Studies on CUDA Offloading for Real-Time Simulation and Visualization

Edgar Josafat Martínez-Noriega

電気通信大学

情報・通信工学専攻

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Engineering

March 2020

Studies on CUDA Offloading for Real-Time Simulation and Visualization

Chairperson:	Prof. Narumi Tetsu	(成見　哲　先生)
Member:	Prof. Terada Minoru	(寺田　実　先生)
Member:	Prof. Nakatani Yoshinobu	(仲谷　栄伸　先生)
Member:	Prof. Yoshinaga Tsutomu	(吉永　努　先生)
Member:	Prof. Miwa Shinobu	(三輪　忍　先生)

概要

リアルタイムのシミュレーションと可視化のためのCUDA のオフロードに関する研究

マルチネス　ノリエガ　エドガー　ホサファット

電気通信大学

本論文では, GPU を使ったリアルタイムのシミュレーションを可視化する際に, 計算部分をネットワークの先にオフロードすることで計算効率を向上できることを示している. GPU はもともと3D グラフィックス用に開発されたものではあるが, 近年はGPGPU を呼ばれる汎用的な計算が行えるようになってきている. 様々なコンピュータシミュレーションもGPU 上で実行出来るが, その中でCUDA と呼ばれるアーキテクチャはGPU 業界の事実上の標準技術となっている. 一方タブレットやスマートホンのようなモバイルデバイスは, タッチ機能や加速度センサーのようなPC には無かった機能が追加されており, データを可視化し操作する際のやり方が以前とは変わってきている. 例えば分子シミュレーションの世界では, インタラクティブに操作可能な程シミュレーションが高速化されてきており, 特定の分子を人工的に動かすことで周りの分子の反応を見るなどシミュレーション技術の新しい方向性が生まれている. ただしモバイルデバイスには消費電力的な制約があり, PC 用のGPU 程の性能は期待出来ない. このようなモバイルデバイスの性能を補完するために, クラウド技術を使う方法がある. つまり計算の重い部分に関してはネットワークの先のGPU サーバーに処理を任せる. このようなやり方をCUDA のオフロードと呼び, GVirtus, ShadowFax, DS-CUDA, GPUvm, MGP, vCUDA, rCUDA 等のフレームワークが提唱されている. 本論文では, リアルタイムの分子動力学シミュレーションを対象のアプリケーションと定め,タブレット端末上で高速に実行するために有効なオフロードの方法を検討した. 最初にDS-CUDA を用いてCUDA の計算だけをGPU サーバーにオフロードするシステムを評価した. 特にこれまでサポートされていなかったAndroid タブレットからのオフロードシステムも開発した. この結果タブレット単体に比べて高い演算性能は達成できたものの, 画面表示のフレームレートが十分に滑らかには出来なかった. これはタブレットとGPU サーバー間の通信がボトルネックになっていたからである. このボトルネッ

クを無くすため, CUDA のDynamic Parallelism 機能を用い, rCUDA と組み合わせた. この結果高い演算性能と同時に高いフレームレートを実現出来る組み合わせを発見した. 更に, タブレットとGPU サーバーの合計の消費電力を測定し, 提案したシステムがGPU サーバー単体よりも高い電力効率を達成したことを示した. つまりタブレットの操作性を持ちながら高い計算性能を持つシステムが実現した. 最後に, オフロードの性能を更に向上させるための手法を提案した. CUDA 機能と描画機能でメモリを共有するInteroperability 機能や, 動画のエンコード/デコード機能を用いることにより, よりオーバーヘッドが減ることが期待出来る. 近年のゲームストリーミングサービスで同様の機能が使われていることから, コンピュータシミュレーションの世界でもこのようなオフロードの仕組みが有用になることが期待される.

Abstract

Studies on CUDA Offloading for Real-Time Simulation and Visualization

by

Edgar Josafat Martínez-Noriega

The University of Electro-Communications

Professor: Narumi Tetsu

The Graphics Processing Unit (GPU) is a co-processor designed to aid the Central Processing Unit (CPU) for rendering 3D graphics. The prompt development of these graphics chips due to the popularity of games and media design helped the GPU to evolve its ubiquitous parallel architecture. The programmability of these devices increased with the introduction of shaders, and thus using the GPU for more than rendering pixels. A new paradigm was introduced by General Purpose Computing on Graphics Processing Unit (GPGPU). At the present time, super computers in the top ten are powered by GPUs in order to accelerate physical phenomena simulations. Moreover, programming models such as Compute Unified Device Architecture (CUDA) and OpenCL have been proposed from major GPU manufactures. Nevertheless, CUDA has proven to be the first choice from the developer community due to its extensive support and applications.

On the other hand, post-PC devices such as smart phones and tablets have become elemental in our daily life. These mobile devices equipped with touch screen and many sensors, provide new ways to visualize and interact with data. Interactive modelling on Molecular Dynamics (MD) simulation, is one example where these devices can offer a better user experience. However, post-PC devices are designed for low power consumption, thus their computational power is not enough to perform such compute intensive applications.

Moreover, a new approach that can complement the low computing power of mobile devices is cloud computing. Implementing a server-client scheme, cloud computing allows to offload computational intensive routines and hookup with massive parallel accelerators such as GPUs. In order to have access to these hardware accelerators, tools such as GPU virtualization frameworks has been proposed: GVirtus, ShadowFax, DS-CUDA, GPUvm,

MGP, vCUDA, and rCUDA. These virtualization tools can handle a remote GPU in order to accelerate execution within applications and reducing code complexity.

In this dissertation, we study and analyse the rendering, computational power, and power efficiency when GPU virtualization tools are implemented to accelerate an MD simulation and visualization on a tablet device. We proposed to offload the most computational intensive routines to a remote GPU. Two cases are reported: In the first scenario, we used a low-powered GPU from a notebook as a server in order to keep power efficiency of the whole system. We selected DS-CUDA framework to enable the development of remote offloading using an Android tablet. Only CUDA kernels were offloaded since DS-CUDA preprocessor has the capability to wrap seamlessly CUDA code without modification. Calculation speeds are reported when the MD was compared between GPU and CPU implementation inside the tablet device. However, to get larger calculation performance, the visualization speed need to be decreased. The efficiency of GPU can be improved by decreasing the frequency of updating a frame to render. Nevertheless, this is not the optimal way to achieve real-time visualization of MD simulations. By the time of performing the experiments, we were one of the first attempts to bring GPU virtualization to an Android device.

In the second case, a novel idea to tackle communication reduction in the execution of real-time MD simulation and visualization using tablets is proposed by applying Dynamic Parallelism (DP) in the GPU. We switched to the rCUDA virtualization framework instead of DS-CUDA, since the first one is more up to date and presents better communication latency compared against the second one. We implemented DP in order to hide the latency to call a GPU routine from a CPU in our MD simulation and visualization. This technique allows our system to achieve better computational performance, more frames per second than a tablet powered by a CUDA capable GPU. Moreover, our results confirm that keeping the GPU saturated with more steps in the MD simulation per frame helped in the reduction of the latency from the client-side. However, using more steps affects the frame rate of the visualization. We found that 250 steps were optimal for our system achieving enough frame rate and better power efficiency when multiple clients were used.

Our system proposal is capable of real-time MD simulation and visualization. With a $dt = 2 \times 10^{-15}$ we can reach proximately 800 nsec/day with a frame rate of 20 fps for a 2,744 particles using our proposed system. We were able to achieve interactive frame rates by tuning parameters using a remote GPU from a tablet device. This is rather not conventional since offloading involves the communication bottleneck from the network. However, applying DP we were able to compensate computational and rendering speed.

Lastly, we set up the following research directions by reducing the communication overhead between the rendering and computation process using a remote GPU. We proposed to apply software capabilities such as Graphics Interoperability and take advantage of the

in-hardware modules of encoder/decoder for image processing. The main idea is to broadcast through the network the final frame buffer. Preliminary results demonstrated poor performance. However, customizing the communication routines with buffer techniques could lead to better execution. This research path presents huge expectations since the evolution of the GPU will be boosted by the incoming services such as game streaming.

Acknowledgements

I would like to express my special appreciation and thanks to my advisor, Professor Dr. Narumi Tetsu. For his continuous support, pieces of advice and encouragement, in both, life and research. For all his feedback, knowledge and leverage on the topics using the GPU. To provide me the chance to be in his laboratory for more than 7 years. For all his positive energy and patience when I was through rough times during my Master and PhD. course.

Special thanks go to Professor Dr. Syunji Yazaki for his encouragement, bits of advice and discussion on the preparation of the paper manuscript. As well as for all his feedback on my research topic. Special mention also goes to Professor Uehara Suwako for her unconditional support. For all her advice on English skills. Also for giving me the opportunity to work on the SAP and contributing to her research with numerous projects. For her friendship and advice during hard times. As well I would like to thank Professor Dr. Choo who is in charge of the JUSST program. For all the opportunities to teach and to be part of the staff. Also, for all his pieces of advice. I am in debt also with the JUSST program from this University. To all my friends in UEC and lab members during all these years. Especially, to Jairo, Edgarito, and Julio.

To my family for all their support. My mother Edith and my father Raul. Without their love, wisdom and support, this achievement in my life would not be possible. To my brother Raul for his encouragement, advice, and knowledge. For his support during hard times and a positive vibe, always thank you very much, brother. As well, to all my family in Mexico. Special thanks also to Dr. Trejo who provided me support in difficult times.

Finally, I would like to give a special mention to MengMeng who has been there always for me. For all her love, inspiration, patience, support and the encouragement necessary to conclude my PhD.

Contents

List of Figures

List of Tables

Listings

1

Introduction

At the beginning of the history of computers, models such as the Electronic Numerical Integrator Computer (ENIAC) and the Universal Automatic Computer (UNIVAC) occupied a whole room of a building providing only 1K Floating-point Operation Per Second (FLOPS). These machines were the ancestors of the supercomputers, introducing a new field called High Performance Computing (HPC) at the time. The applications for these big computers were only for military usage. With the advance of the TTL technology on the decade of the 70's, companies such as Intel, ARM, Zilog, IBM, and Motorola started the development of microprocessors. They welcome a digital era for computing. Since that time, the Central Processing Unit (CPU) was the core of the computers. The CPU evolved to become a sophisticated piece of hardware which is focused on dispatching work through the Operating System (OS) for modern computers. However, there has been the development of another kind of hardware accelerator that is dedicated to a special purpose. These devices are designed at a hardware level to solve a specific task, such as Molecular Dynamics (MD) simulations. Some of the characteristics on these devices are highly parallel architecture and multi-core implementation. Anton [1], ATOMS [2], FASTRUN [3], CSX600 [4], and MD-GRAPE [5] are some examples. Nevertheless, the development of these specialized hardware involves a huge budget, thus the price of each device is really high. Table 1.1 shows the estimated cost of these devices when they were released.

Developer	**Accelerator**	**Estimate cost per Unit**
CSX600	ClearSpeed	~ $10,000[1]
ATOMS	AT&T Bell	~ $186,000 (1990)
FASTRUN	Columbia University	~ $17,000 (1989)
MDGRAPE-3	Riken	~ $9,000,000
GPU	NVIDIA / ATI	~ $200-800[2]

Table 1.1: Unit price in USD for specialized computer accelerators.

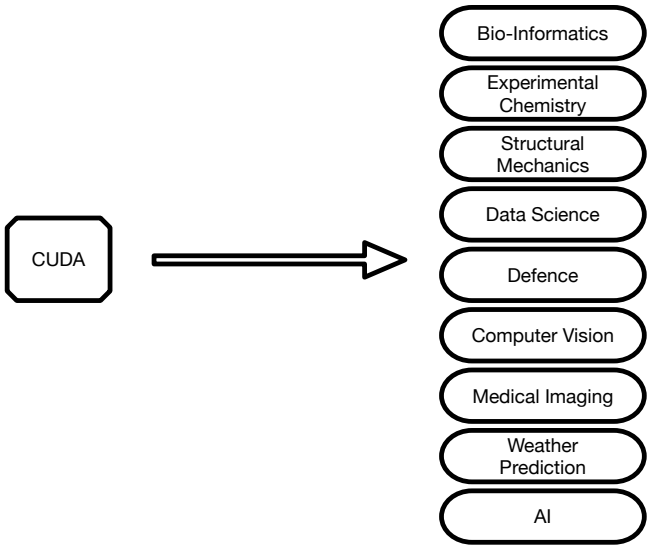

Figure 1.1: CUDA applications over different fields.

The Graphics Processing Unit (GPU) was born due to the need for rendering pixels and presents into a display that the modern OS requires. This is due to graphical applications and the window system that the OS implemented for a better user experience. As well, media, CAD design, and video games boosted the evolution of the GPU, making its massive production relatively cheap to develop. Yet, this is another specialized hardware that presents a parallel architecture design. The GPU is optimized for Floating-point calculation due to the primitive image processing operation for color output. This can be done using its massively programmable processors. During the 80's decade rendering machines such as Ikonas [6], Pixel Planes 5 [7], the Pixel Machine [8] were proposed for general-purpose computing. Hence, a new paradigm was introduced: General-Purpose computing on Graphics Processing Unit (GPGPU). On the first attempts of using this new paradigm in recent GPUs, advance knowledge of the graphics pipeline was necessary. Controlling buffers inside the GPU for data allocation was necessary, and programming shaders provided the ability to implement the algorithm. The final computation did not involve pixels or any image-related data. NVIDIA, the GPU company introduced Compute Unified Device Architecture (CUDA) in 2006. CUDA is an architecture and programming framework that enables dramatic increases in computing performance by extending shader units to general-purpose computing. Since its introduction, CUDA has successfully accelerated applications in some of the fields presented in Figure 1.1. Hence, top of supercomputers, in the list of TOP500 [9], are equipped with GPUs.

Moreover, in order to utilize a conglomerate of GPUs in the cloud environment, HPC virtualization tools have been proposed. These frameworks provide the ease for programming

[1]This cost is not the actual cost per unit rather reflects the cost of one node.
[2]This cost represents only the public unit for the consumer.

in multi-node heterogeneous computers by virtualizing GPUs on a distributed network, as if they were attached to a single node. Thus, using a remote GPU from another device as an accelerator of this kind is feasible with such virtualization frameworks.

On the other hand, since the introduction of the first iPhone from Apple in 2007, so-called Post-PC devices, came along to the scenario to define a new way to interact with mobile computers. Nowadays, these devices are essential in our main daily activities such as reading emails, taking pictures, playing games, using social networks and also creating our own content. However, its inherent mobile nature forces the design of these devices with low computation power.

Combining these two worlds, mobility (embedded devices) and GPUs have been blocked in the growth path. This is mainly due to the huge power consumption that GPUs required to work. Discrete or desktop GPUs have a range from ∼150 to ∼250 Watts presents a considerable constraint to be implemented in low-power environments such as embedded devices. However, for laptop PC computers *integrated* GPUs are implemented. These integrated GPUs are designed for power efficiency and its power consumption in teens of Watt. Even though integrated GPUs save a considerable amount of power consumption, they can deliver almost the same computing power of their desktop counterpart models when a parallel task is given [10]. In this dissertation the combination of mobile devices with these integrated GPUs is presented in order to achieve a better power efficiency for the whole system.

1.1 Research Purpose - Objective

The main idea in the early stages of this research was the conception of a prototype similar to that shown in Figure 1.2. The main motivation behind this study is merging high-performance machines with post-PC devices. These touching screen devices present different sensors and many user interface capabilities which lead to a new way to dive into the information presented to the user. Nevertheless, the mobile device itself is not equipped with enough computational power to perform heavy computational simulations. It presents a challenge that must be tackle taking into account the different scenarios that are already proposed.

In order to understand the offloading from client devices to cloud servers, we have to identify the different characteristics and capabilities that servers in the cloud offers. Narumi *et al.* [11] classify these combinations in three different categories:

- **A)** --→ Most of the calculation and rendering is performed in the server cloud.
- **B)** --→ Only rendering is performed in the server cloud.
- **C)** --→ Only calculations is performed in the server cloud.

On the A) side, we can define the client as zero-client since only the input from sensors is sent to the cloud. The server, retrieve only images in the form of video to the client.

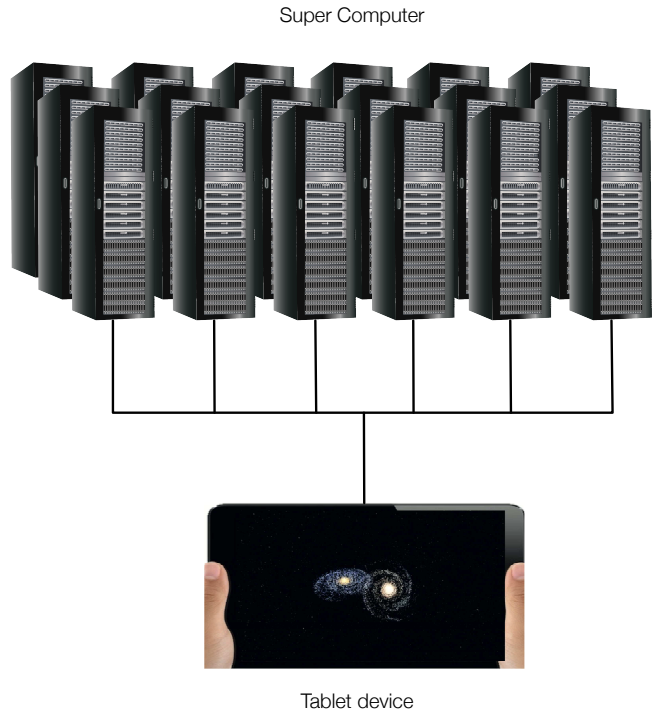

Figure 1.2: System prototype as main motivation of this study.

There are currently solutions of this type such as NVIDIA GRID and Amazon EC2. This kind of approach restrains the application development environment since they only provide popular ISV applications. Moreover, Special API or another kind of mechanism is needed if full tablet sensors are required. Finally, video transfer could become a bottleneck, thus special compression may be needed, pushing and consuming computational power from the client.

On the B) side, only rendering APIs such as OpenGL, Vulkan or Direct3D are viable to utilize. Approaches such as VirtualGL have been proposed. However, the missing APIs for high-performance computing such as CUDA are not supported which implies a a big disadvantage since we want to merge HPC applications.

On the C) side, rendering, and other light processes are performed on the client-side. The development environment is not a constraint here since only CUDA code is utilized for offloading. Users have full control and access to the native development environment. Thus, all access to sensors and other client capabilities. Finally, with this approach, the developer can benefit from high-end GPUs on a cloud server by hooking CUDA APIs in their applications.

Utilizing GPU virtualization frameworks are a feasible solution since they provide the ability to use remotely a GPU in a cloud environment.

We highlight, the main objectives presented in this dissertation:

First

We proposed a system composed of a server equipped with a GPU accelerator device in order to perform an MD simulation and to visualize on a tablet device. We used GPU virtualization tools in order to use remotely a GPU in a cloud environment.

Second

We used DS-CUDA framework in order to offload intensive parts of the MD simulations. Only kernel information is offloaded in this case. An analysis of communication, computational power, and rendering performances are presented.

Third

We utilized rCUDA framework to further enhance our proposed system implementing Dynamic Parallelism (DP) as a mechanism to avoid communication inside kernel launch. An analysis of computational power, rendering speed, and electric power performance is reported. Furthermore, results using various clients for better computational and electric power distribution is included as well.

Fourth

We proposed to enable GPU graphics acceleration in our server-client scheme by implementing graphics interoperability capabilities. These features are not available on the GPU virtualization frameworks due to their local execution nature. However, using in-hardware modules such as encoder-decoder, we give the first steps in broadcasting the final image to the client-side using frame buffer through the network.

We proposed a system capable of interactive MD simulation and visualization by using a remote GPU (server) and a tablet device (client). Offloading techniques are rather known to enhance capabilities on the client-side, especially computing power. However, a communication bottleneck may be a concern due to the network. Our proposal alleviates this problem by tuning parameters and using DP to hide latency when a remote GPU is used.

1.2 Related Work

As we mentioned in the section above, in our approach we proposed a system composed of a tablet device (client) and a power-efficient GPU (server) attached to a laptop PC in order to accelerate MD simulations. Other proposals in the field have been made similar to our idea.

Efforts to create new contents has lead a numerous variety of research topics such as visualization data, virtual reality, health-based applications, between others [12] [13] [14] [15] & [16]. Although these proposals use a mobile device for data visualization, they do not implement any kind of acceleration offloading nor local.

Ideas to include interactive simulations and visualizations have been proposed [17], [18], [19] & [20]. These proposals used the interactivity as a medium of facilitating the user a more

comprehensive and informative simulation. When MD is carried out, selected areas of the molecule can be enhanced by the user for example. These ideas are rather to be executed in normal PC machines, they do not support mobile architectures.

Several proposals including offloading from a mobile device have been made [21], [22], [23]. These proposals use the cloud in order to get better performance inside the application running in the mobile device. As well, they include patterns for better electric power use in order to save battery life. Nevertheless, they do not include CUDA support for the offloading part.

Furthermore, some ideas to take advantage of the parallel frameworks inside the mobile device such as RenderScript, OpenCL, and ParallDroid has been made [24], [25] & [26]. The authors on these proposals used a local acceleration, utilizing the GPU for a particle filter, synthetic radar imaging, and a benchmark. However, the corresponding reports do not include electric power measurements.

Ideas similar to ours have been proposed in [27], [28], [29] & [30]. Differences between these proposals ours are as follows: the first proposal used rCUDA GPU virtualization framework in order to offload part of the image filter using an expose fusion algorithm using a mobile device. However, the author claims a negative performance on the client-side. Moreover, they report battery power consumption to be negative when offloading is performed. In the second case, an image processing algorithm is applied running in a mobile device. They used the cloud for offloading intensive computational parts of the algorithm for acceleration using the OpenCL framework. They report gains in performance and power savings. However CUDA is not supported. The third case used GVirtuS framework to offload a matrix multiplication to several ARM GPU servers. Although the author reported performance gains and low latency as the size of the matrix increases, they do not include power analysis. Moreover, their application is not targeting any real-time visualization. In the last proposal, the author used rCUDA to offload MD simulations to an ARM server equipped with several GPU hardware. They characterized the execution using remote offloading and local one, mentioning that using a server guided a power-saving. However, they do not include power measurements nor visualization of the MD simulation.

Lastly, we mentioned some proposals that are similar to our approach in the future directions [31], [32] & [33]. These proposals implemented real-time visualization for remote simulations. They proposed to use in-hardware features of the GPU such as Ray-Tracing for photo-realistic rendering. This is rather important since interactive photo-realistic visualization will bring a better understanding of the physical phenomena. As well, they proposed to used in-hardware encoder/decoder for frame buffer streaming using virtual reality (VR) headset. Our idea is similar to their proposals for future directions. However, we propose to take advantage of whole GPU hardware for simulation and visualization, with the possibility

to include those features in GPU virtualization frameworks to facilitate the development of such applications.

1.3 Thesis Organization

The present work is divided into 7 Chapters. In Chapter 2 we talk about the GPU as a general-purpose computing device. As well, we introduce the CUDA programming model and architecture. We highlight those features on the GPU which are fundamental in this dissertation. Furthermore, we introduce the GPU virtualization frameworks which allow using GPUs in a cloud environment. In Chapter 3, we introduce the MD simulation and visualization which is the main application for our GPU offloading techniques. Relevant versions of Claret software are mentioned, as well as the port for Android tablets. Chapter 4 discusses our first approach to offload heavy computational parts from the MD simulation using a tablet by DS-CUDA GPU virtualization framework. We report speed up on computational power and rendering on the tablet side. On Chapter 5 we further optimize our MD simulation and visualization using tablets by applying DP. In this case, rCUDA GPU virtualization framework is used. Gains in computational power and reduction on latency were achieved by applying DP. Moreover, we report power measurements using multiple clients. On Chapter 6, we settle the first steps towards GPU virtualization frameworks that enable graphics acceleration on the server-side. Preliminary results of the broadcasting frame buffers over the network are presented. Finally, in Chapter 7 we provide final thoughts and conclusions about this dissertation.

2

General-Purpose Computing on the GPU

The Graphics Processing Unit or GPU was conceived to aid the CPU in rendering high-quality 3D images. This hardware accelerator gained popularity since the demand for rendering capabilities from the PCs was growing noticeably. This was mainly due to the graphical operating systems that appeared in the late 80's. With the arise of this new interactive paradigm on computers, more applications for visualization were developed, such as video games and CAD design among many others. Since then, graphics cards have become an intrinsic part of computers and indispensable tool for software visualization. Due to the large competitive market in this range of devices, the GPU has become powerful hardware for a comparatively low cost.

During the beginning of the 2000s, a new paradigm that allows the computation of any kind of data in GPUs were growing. This new paradigm has its origins based on the General Purpose Computing on Graphics Processing Units (GPGPU). GPUs at this point were designed to produce a color for every pixel using programmable arithmetic units called *pixel shaders*. In a general way, these shaders use the (x, y) position on the screen and some other additional information to combine various inputs in computing the final color that will be displayed. The additional information could be input colors, coordinates for textures, or other attributes that the shader needs in order to be executed. However, the arithmetic is performed on the input colors and textures were completely controlled by the programmer. It was observed that these inputs "colors" could be replaced by any kind of data. Although, this new shift for the usage of GPUs started promising with the idea of taking advantage of its ubiquitous parallelism, yet it was particularly known for their great programming difficulty due to the high level of knowledge in the graphics pipeline. Some of the first attempts on GPGPU were specific to intensive computing applications and frameworks compatible with

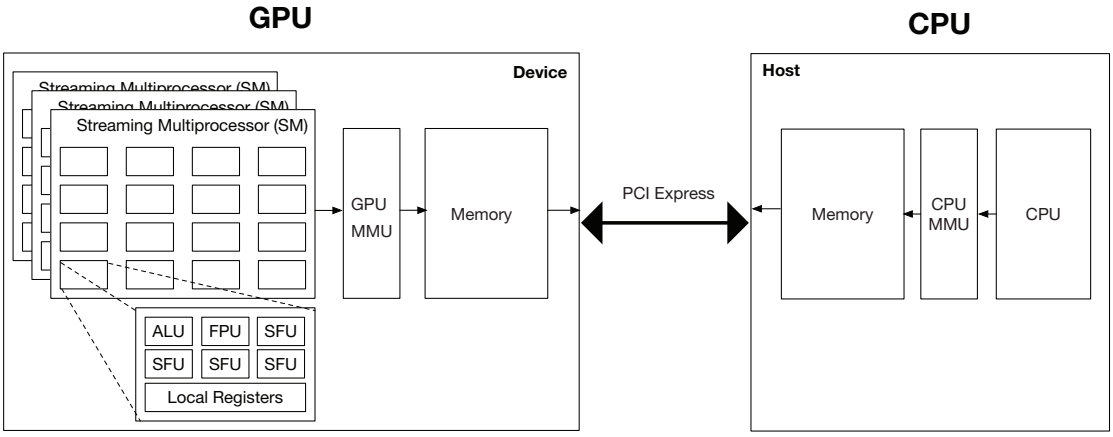

Figure 2.1: Basic architecture of a "Heterogeneous" system GPU-CPU.

OpenGL and Direct3D [34, 35, 36, 37].

2.1 General GPU Architecture

The Graphics Processing Unit is special hardware which is designed mainly to execute parallel applications being 3D graphics the fundamental one. This is rather different from the counterpart, the CPU [38]. The GPU is also designed to offer many thousands of single cores using a high bandwidth memory. As we can denote from these characteristics, this hardware maximizes the throughput inside the application by exploiting the data parallelism launching a large number of threads per call. In this scenario, memory access latency can be hidden using big chunks of computing [39]. This kind of technique is rather slow per single thread on execution performance. However, the total performance represents a gain in throughput.

Nowadays, heterogeneous systems composed of GPU and CPU are the common norm on PCs. Figure 2.1 shows the traditional system. Although the GPU architecture may be different from implementation and model, they all adopt a similar high-level implementation. The GPU is composed of several streaming multiprocessors (SM) that contain several computing modules or cores. Each core contains an integer Arithmetic Logic Unit (ALU), a Floating Point Unit (FPU), several Special Functions Units (SFU) and local registers. The GPU Memory Management Unit (MMU) grants virtual address spaces. A host can be connected by utilizing a PCI-Express interface. A large amount of data can be transferred between the host memory space and the GPU by the Direct Memory Access (DMA) engine. However, this can cause data transfer overhead due to the low transfer bandwidth of the PCIe interface when compared to the internal memory bandwidth of the GPU.

2.2 CUDA Overview

Compute Unified Device Architecture (CUDA) is a framework and a computing architecture developed by NVIDIA, first introduced in 2006 within the GPU GeForce 8800 GTX. This first GPU chip aimed to alleviate many of the limitations that prevent previous graphics processors from being legitimately useful for general-purpose computation. Before CUDA conception, an advanced degree of the 3D graphics pipeline knowledge was needed to handle GPUs. However, CUDA uses a base C like syntax and programming model. This makes CUDA more program-affordable for more developers. The chip in GeForce 8800 GTX was one of the first DirectX 10 compatible devices, bringing the speed up on science and start the revolution of GPGPU. NVIDIA uses the standard IEEE 754-1985 [40] for single floating point precision on the creation of the Arithmetic Logic Unit (ALU) inside the GPU chips. Also, these chips include many functions not oriented to graphics rendering. The new memory hierarchy inside of the device composed up to 5 levels were introduced.

Previously, GPUs were used primarily for the media design, high-end multimedia, and games sector. Nowadays, CUDA has an impact on the following practical applications:

- ☐ Fast Video Transcoding
- ☐ Video Enhancement
- ☐ Oil and Natural Resource Exploration
- ☐ Medical Imaging
- ☐ Computational Sciences
- ☐ Neural Networks
- ☐ Gate-level VLSI Simulation
- ☐ Fluid Dynamics

In recent years, companies such as NVIDIA and other major GPU manufacturers have implemented a much more easy way to reach and program GPUs for general-purpose computation. Thus, industry-standard frameworks and architectures have been developed such as CUDA and OpenCL.

2.3 CUDA Programming Model

The structure of a CUDA program is grouped in various phases that are executed in the *host* (CPU) or inside of the *device* (GPU). The sections of the application which presents a lot of parallelism are executed inside of the device. Contrarily, the serial parts are on the host side. Hence, a CUDA program is a code execution combination inside of the host and device. In order to compile and use CUDA with C/C++, NVIDIA provides a compiler called *nvcc* which separates and processes the code for each part. Figure 2.2 shows this flow.

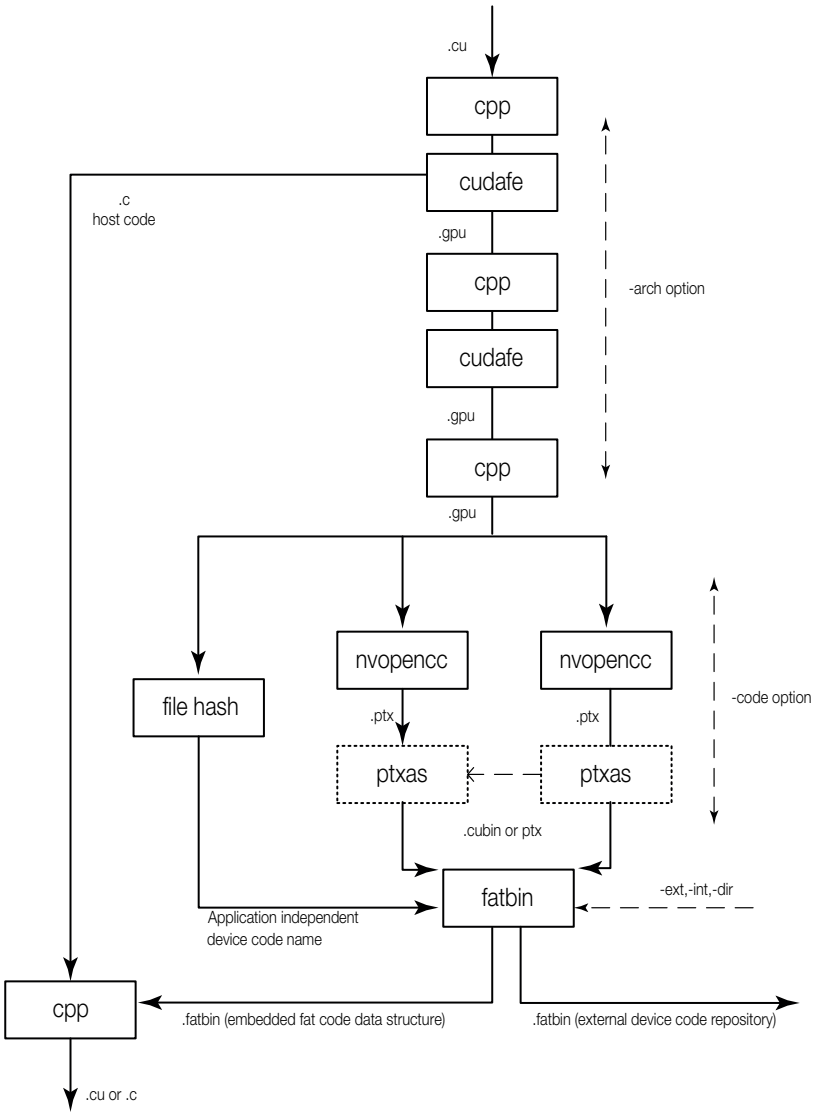

Figure 2.2: C/C++ compilation trajectory using *nvcc*.

Main CUDA files use *.cu* extension. The code that belongs to the host is ANSI C standard. This part of the code is processed by a normal C language compiler such as *gcc* or *clang*. The execution of this code is done in the CPU. The code executed in the device is processed in different ANSI C standard that extends "key-words" for parallel functions called *kernels* and its associated data structures.

2.3.1 Kernels

Subroutines that are executed inside of the GPU are called *kernels*. This GPU subroutines are able to call a massive number of *threads* per launch in order to process several amounts of data at the same time. Each GPU is composed of many Multiprocessors (MP) which are the recipients of the actual threads inside of the hardware. Depending on the compute

```
1  __global__ void MyKernel(float* x, float* v, float cons) {
2
3      int i = threadIdx.x;
4
5      x[i] = x[i] + v [i] * cons;
6
7    }
8    ......
9    ......
10   ......
11   int main()  {
12
13     //Kernel call from the Host
14     MyKernel<<<1,N>>>(X,V,Cons);
15
16   }
```

Listing 2.1: Simple kernel structure for CUDA C/C++ code.

capability[1], we can launch up to 1024 threads per MP or more. One thread does not process the same data at the same time considering that each thread have a different ID or *Index*. This special identifier will allow the thread to access different data from different memory regions. One simple kernel sample is shown in the List 2.1.

The definition of a kernel is done with the usage of a special identifier inside the code using the reserved word **__global__**. As the sample code shown above, these definitions are like normal C/C++ function declarations, with output and input type arguments. This is the actual code that is executed in the GPU. The special index for each thread is reachable by one *built-in* variable called **threadIdx**. In order to specify the number of threads to be launched per kernel, another identifier is introduced <<<....>>>. This pattern of code execution operates using the paradigm *Single Instruction, Multiple Data* (SIMD) which is used on the GPUs, on the opposite side to the CPU which uses *Single Instruction, Single Data* (SISD) paradigm. CUDA has implemented the concept of *Single Instruction, Multiple Thread* (SIMT) which consists of executing code depending on the parity of the index of a thread.

Implementing trivial kernels for GPU using CUDA is very straight forward for a C/C++ developer. However, to tune the GPU at maximum performance is rather complicated. We have to take care of every hardware-specific details such as so-called *warp*. This specification of the GPU is a set of threads that all share the same code, follow the same execution path with minimal divergences and are expected to stall at the same places. A hardware design can exploit the commonality of the threads belonging to a warp by combining their memory accesses and assuming that it is fine to pause and resume all the threads at the same time. Thus, the developer should handle and consider the conflict of memory between different indexes.

[1]The compute capability of a GPU determines its general hardware specifications and available features.

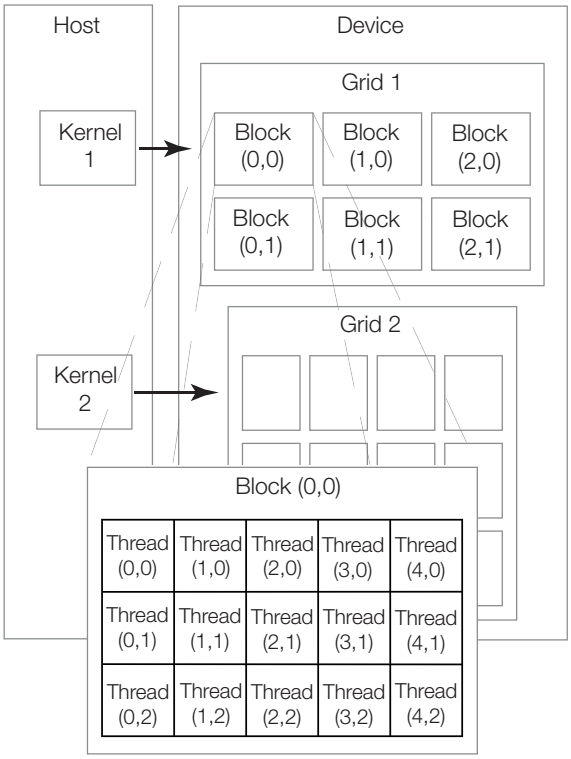

Figure 2.3: Thread, Block and Grid organization inside of CUDA architecture.

2.3.2 Thread Management

The built-in variable **threadIdx** is a vector with 3 components that is able to identify threads by an Uni-dimensional (1D), Bi-dimensional (2D) or Tree-dimensional (3D) arrangement.

- ☐ `threadIdx.x`
- ☐ `threadIdx.y`
- ☐ `threadIdx.z`

A bunch of threads can be grouped into *blocks*, which at the same time are collapsed by 1D, 2D and 3D index variable **blockIdx**. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

- ☐ `blockIdx.x`
- ☐ `blockIdx.y`
- ☐ `blockIdx.z`

Blocks are organized as well into a one-dimensional, two-dimensional, or three-dimensional. A group of blocks is called *grid*. The number of thread blocks in a grid is proportional by the size of the data to be computed for the processors in the system. Figure 2.3 shows the complete organization.

Memory	Global	Constant	Texture	Shared	Local
Access	W/R	R	R	W/R	W/R
Size	$\geq$ 1 GB	64 KB	$\geq$ 1 GB	32 KB	$\geq$ 100 MB
Scope	Application	Application	Application	Per Block	Per Thread

Table 2.1: CUDA memory attributes. W/R = Reading and Writing. R = Read only.

There is a limit of threads that are able to be launched per block. Actual GPUs can handle over 1024 threads per execution. However, this limit is constrained to a special memory segment shared for all threads inside of the same SM. Moreover, a kernel is able to execute a multiple amounts of blocks per time. Thus, the total amount of threads to be launched inside the GPU is equal to the number of threads per block multiplied by the number of blocks.

2.3.3 Memory

CUDA capable GPUs are integrated with 5 different memory regions. Each of them has different characteristics, size, and functionality. In order to squeeze all the computing power from the GPU, the understanding and management of these different memory spaces are crucial. Table 2.1 shows the main characteristics of these types of memory. Depending on the hardware, the size of this region may be bigger, especially with the newest GPU generation.

Following, we add a brief description and usage of these 5 different memory spaces.

Global Memory

This is the main memory region as its name suggests on the hardware. It is the biggest zone that a kernel is able to write and read data. The usage of dynamic memory allocation is not allowed, it must be handled before the application starts. According to the GPU model, the size may vary rounding the $\sim$ 1GB or more. During the kernel call, this memory space is persistent.

Constant Memory

Constant memory is relatively small compared to other regions, reaching sizes of 64KB and with an attribute of "read-only". This space is persistent along with the kernel calls. The host is able to load any kind of data inside of this region of memory. The attribute "read-only" refers to the ability of a kernel for no modification on this region inside the application by the device.

Texture Memory

Specialized memory to load, mapping, and modeling elements in 2D and 3D, which is fast and "read-only". This memory region offers the ability to communicate with graphics pipelines such as Direct X and OpenGL. This could lead to time-saving when reaching objects in memory space delivering faster rendering outputs.

Shared Memory

Shared memory is the smallest memory region among others. The size is about 32KB and it is the closest similar to cache in CPUs. Shared memory is not persistent along with the kernel's call. The host (CPU) can not load data on application time. However, when the device performs a kernel call, this can specify up to 32KB read and write zone for all the threads within a block. Furthermore, all the threads inside of a block share this memory space. After the last execution of the last thread, this space is deallocated. Performing memory operations inside this space are faster than the global memory for the same threads within a block.

Local Memory

Local memory has similar attributes and functionality to global memory. Differences are the life time and the variable scope. For this memory region, the scope is limited to one single thread. The main reason for this is that if every SM can run up to 1024 threads concurrently and there are only 16384 registers, each thread can only use 16 of them with a full load. If more different variables are needed at the same time, these will be allocated in the local memory. Unfortunately, this choice is left for the compiler in order to save register spaces.

In Figure 2.4 we show the different memory types in CUDA architecture. As we can denote, the closest access to the threads is faster memory but smaller in size. It is not a trivial task to use them and manage. However, the proper handling of CUDA memory regions may impact directly to the performance of the final CUDA application.

2.4 CUDA Capabilities

The CUDA platform, architecture, and programming ecosystem have been evolving since its conception in 2006, adding new hardware and including new libraries to get exceptional performance. Some of the libraries that are packed in the CUDA SDK are the followings:

- **cuBLAS** --→ CUDA Basic linear algebra subroutines
- **cuFFT** --→ CUDA Fast fourier transform
- **cuRAND** --→ CUDA Random number generation

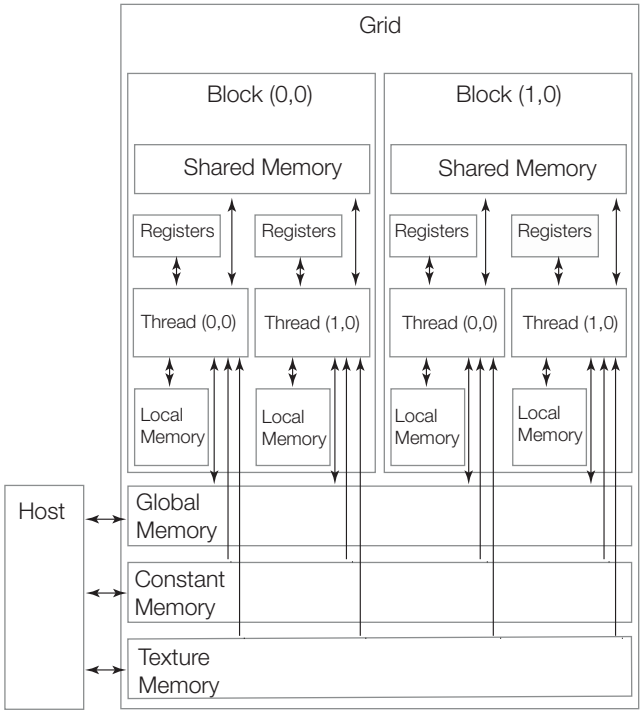

Figure 2.4: Different memory regions on CUDA architecture.

- **cuSOLVER** --→ CUDA Based collection of dense and sparse direct solvers
- **cuSPARSE** --→ CUDA Sparse matrix
- **CUTLASS** --→ CUDA Custom linear algebra algorithms
- **nvJPEG** --→ CUDA Hybrid JPEG processing

The libraries mentioned above provides good performance and it provides the developer easy-to-handle functions, data types, and structures for each field. Although, there are many features inside CUDA architecture, in the following sections we add a brief description of the most important points inside this dissertation.

2.4.1 Dynamic Parallelism

Dynamic Parallelism (DP) is the capability inside the programming execution model that CUDA provides in order to create and synchronize new nested workload. This can be explained as follows: the ability of a CUDA kernel parent to create new CUDA kernel child invocation and synchronization. The parent kernel has the ability to get the output from the child kernel without having to involve Host operations. A simple example is shown below:

Naturally, recursion methods are supported by Dynamic Parallelism. Additional, parallelism can be exposed to the GPU's hardware schedulers and load balancers dynamically, adapting in response to data-driven decisions or workloads. Now, programming patterns such as recursion, an irregular loop structure, and single-level of parallelism can be more easy to

```
// GPU code execution
__global__ Child_K(void* data){
    //Operate on data
}
__global__ Parent_K(void *data){
    Child_K<<<16, 1>>>(data);
}

// CPU code execution
Parent_K<<<256, 64>>>(data);
```

implement. Generally, using Dynamic Parallelism is convenient for implementing algorithms that includes computing adaptive grids, performing recursion, and splitting the work among different and independent threats and batches.

2.4.2 Graphics Interoperability

The graphics interoperability functions are related as its name suggests to the interconnection between CUDA space and rendering API's space. These functions allow CUDA to write and read from OpenGL or Direct3D memory space. This is mainly to alleviate bottleneck on applications that creates a lot of memory traffic between Host and Device. For the best practice and performance effect, it is desirable that applications keep the data inside the GPU as much as possible. Implementing the graphics interoperability function with CUDA gives the kernels the ability to write data inside images and textures that are inside into the graphical frame buffer output from OpenGL or Direct3D.

2.4.3 Hardware-Based Video Encoder and Decoder

From the beginning of Kepler architecture, NVIDIA provided an on-chip video encoder and decoder named *NVENC* and *NVDEC* respectively. This hardware feature provides fully accelerated video encoding and decoding capabilities supporting the most popular codecs. This feature is independent of the graphics engine making the encoding/decoding process suitable to be offloaded to the GPU. This provides the CPU and GPU free to perform other operations. Some of the encoding capabilities are listed as follows:

- ☐ **Formats** --→ H.264, H.265 and Lossless
- ☐ **Bit Depth** --→ 8 and 10 bit
- ☐ **Color** --→ YUV 4:4:4 and YUV:4:2:0
- ☐ **Resolution** --→ Up to 8K

Some of the decoding capabilities are listed as follows:

- ☐ **Formats** --→ MPEG-2, VC1, VP8, VP9, H.264, H.265 and Lossless
- ☐ **Bit Depth** --→ 8,10 and 12 bit

- **Color** --→ YUV 4:4:4 and YUV:4:2:0
- **Resolution** --→ Up to 8K

This hardware accelerator engine for video encoding and decoding on the GPU is faster than real-time video processing using CPU, which makes this feature suitable for video playback and transcoding applications.

2.4.4 Tensor Cores for AI

The tensor cores are specialized hardware execution units designed specifically to perform the tensor and matrix operations that are the core in computing function for Deep Learning algorithms. These cores provide significant performance in speed for matrix computations on deep learning neural network training and inferencing operations. The tensor cores add new INT8 and INT4 precision modes for inferencing processing that tolerate quantization and do not require FP16 precision. These new cores add new deep learning-based AI capabilities to gaming on PCs such as a technique called Deep Learning Super Sampling (DLSS). This new technique allows a deep neural network to extract multidimensional features for rendering a scene and smartly combine details from multiple frames to build a final image. This rendering technique uses fewer input samples than traditional Texture Anti-Aliasing (TAA).

2.4.5 RT Cores for Ray Tracing

The RT cores introduce ray tracing in real-time. These new cores enable a single GPU to render visually realistic 3D scenes. Different from a common rendering algorithm such as rasterization, the ray-tracing algorithm builts complex professional models with physically accurate shadows, reflections, and refractions. RT cores can accelerate ray-tracing by computing on hardware triangle intersections which are a fundamental operation. NVIDIA provides interfaces such as NVIDIA's RTX ray tracing technology, and APIs such as Microsoft DXR, NVIDIA OptiX, and Vulkan ray tracing to deliver a real-time ray tracing experience.

2.5 CUDA on Mobile Devices

Due to the increased usage of smartphones, tablets, and other gadgets, new processor architectures were developed such as ARM. In order to follow the special computing and power demand that these new devices require for daily task, NVIDIA company introduced a new branch of mobile processors called Tegra. This system on chip (SoC) is aimed for mobile architectures such as smart phones, digital cameras, personal digital assistants and internet mobile devices. There are many iterations of this new SoC, Tegra APX, Tegra 2, 3 and 4. However, all of these chips are not CUDA capable. It was in April 2014 when NVIDIA finally released one mobile chip capable of CUDA architecture, the one called Tegra K1. This new

ARM cortex general purpose 32-bit processor includes a CUDA capable GPU. This processor is also capable to run OpenGL ES 3.1, CUDA 6.5 and OpenGL 4.4. Some of the motivations to use this new chip are solutions for compute-intensive embedded projects like autonomous robotic systems, advanced driver assistance systems, mobile medical imaging and intelligent video analytics.

2.6 Remote GPU through Virtualization

Cloud computing is a platform that can help to ease the access to huge compute nodes and to reduce the total cost of the ownership meanwhile achieving high performance and saving energy. The cloud allows users to deploy computational intensive applications without maintaining or acquiring large computational systems. Especially, heterogeneous systems equipped with GPUs are the main focus on big types of equipment [41]. This has to lead to major GPU manufacturers to develop and enhance programming environments [42]. Several HPC applications have been benefited from this approach, such as particle simulation and MD simulations [43, 44]. However, in order to handle remote GPUs, virtualization of some sort is needed to achieve this task. Virtualization techniques allow the creation of elastic components that are used by methods multiplexing system resources. Most of these resources include processors and peripheral devices. The area of virtualizing hardware is not rather new [45]. Nevertheless, virtualizing the GPU is just recently developing due to GPU driver implementations which are not standardized and they are not open for modifications. Thus, standard virtualization techniques can not be applied.

2.6.1 GPU Virtualization Techniques

According to the literature [46], there are basically 3 groups of GPU virtualization techniques. These, are based on their implementation approach:

- API remoting
- Para and Full virtualization
- Hardware supported virtualization

On the first approach, API remoting provides a wrapper communication library between the GPU and the guest machine. This library is in charge of intercepting GPU calls on the guest machine which are redirected to the host machine. The host machine includes the actual GPUs where the remote calls are executed. The results from the request are back to the guest machine. This approach is rather at a higher level of the GPU in the execution stack. However, this technique solves the difficulty of the virtualization of the GPU at the driver level.

On the second approach, para and full virtualization happen at the driver level. As we mentioned before, this is rather difficult since most GPU vendors do not provide the source code of their driver implementation. Nevertheless, some architecture documentation has been opened recently by some manufacturers as an open driver [47]. As well, some efforts from the development community have done with reverse engineering [48] for research purposes.

Third and last approach, hardware-supported virtualization uses a guest OS to access a GPU through the chipset on the motherboard. These capabilities are specified by individual GPU vendors. The access occurs by remapping the DMAs for each call in the guest OS. Some of the most important vendors such as NVIDIA, AMD and Intel support this kind of virtualization [49, 50, 51]. However, one of the main problems of this approach is the lack of supporting multiple GPUs.

Each GPU virtualization technique presents advantages in execution and also some difficulties with the implementation. Nevertheless, in this dissertation, we focus on the API remoting. Next subsection, we present a more detail explanation on this approach.

2.6.2 Remote GPU using API

GPU virtualization presents similar challenges as other virtualization I/O devices. API remoting is up to date and the most useful GPU virtualization technique, specially from GPGPU computing developers. API remoting provides a wrapper library which is used from a guest machine in order to intercept and forward GPU calls. This approach can emulate a GPU execution as if the GPU where physically attach to the guest machine.

The main scheme for API remoting is shown in Figure 2.5. Here, we can denote a guest machine which is able to issue a request to a GPU in another host machine. This virtualization scheme is known as a split device model; the frontend and backend implementation for the GPU drivers are placed inside guest and host machine respectively. The wrapper library located on the guest side awaits for any calls from inside of the application. Once a call is performed, the wrapper library transports the request to the front-end driver. Here, the message is packed and prepared in a suitable format to be sent to the back-end driver in the host machine which will parse the message and convert it to the original API call. Finally, the call handler performs the request to the physical GPU and gets the result back using the reverse path to the guest machine. The main advantage of this approach is the ability to use GPUs without the need of recompiling the code since the wrapper library can be linked at run time. As well, the virtualization presents a negligible overhead as bypasses the hypervisor and other hardware related difficulties. On the other hand, this virtualization approach requires updating the wrapper library constantly in order to cover new hardware features on GPUs. This can be rather a daunting task. Moreover, since API remoting bypasses the hyper-visor, basic virtualization techniques such as live migration, check point, and fault tolerance are

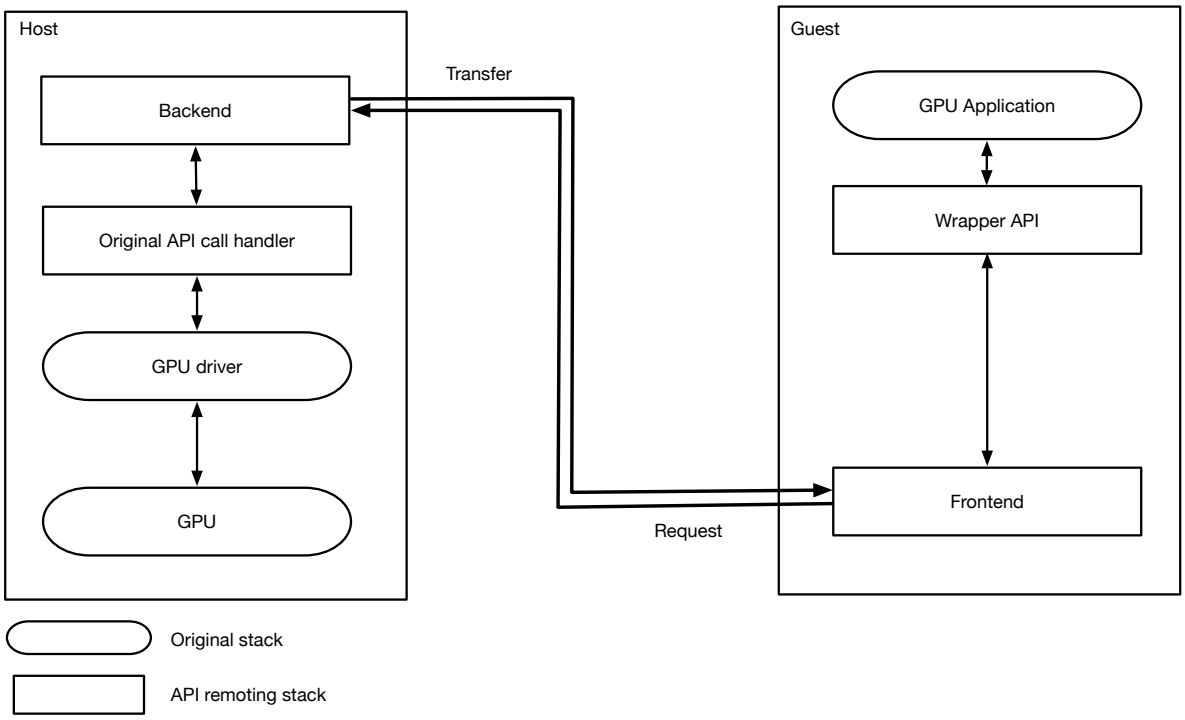

Figure 2.5: API remoting scheme.

difficult to implement in this scheme.

API remoting virtualization technique can be classified according to target of acceleration provided by the wrapper library; acceleration methods for graphics rendering and acceleration methods for GPGPU computing. Inside the first category, the wrapper library consist in the implementation of OpenGL or Direct3D render libraries. Implementations supporting this method of acceleration have been proposed such as VMGL [52], Blink [53], Chromium [54], Parallels Desktop [55] and VADI [56]. On the second category, the wrapper library supports GPGPU computing APIs such as CUDA and OpenCL. Some implementations supporting this method of acceleration include the following proposals: GViM [57], vCUDA [58], GVirtuS[59], GVM [60], Pegasus [61], Shadowfax [62], VOCL [63], rCUDA [64] and DS-CUDA [65].

3

Molecular Dynamics Simulation and Visualization - Claret

The term "Visualization" or visual data exploration plays an important role inside the scientific process. Looking at or analyzing data from experiments is a crucial part of the process of discovering and producing new science. At first, term "Visualization in scientific computing" was used in a report inside the computer graphics and visualization community [66]. Through a series of operations and processing steps, a visualization pipeline transforms abstract data into comprehensible images. Today, scientific visualization plays a central role in the description of computer simulation involving physical phenomenons.

A molecular dynamics simulation (MD) is a computer simulation of the natural phenomena on the matter structure and composition. We can simplify the description as the interaction between atoms. This kind of computer simulation is performed in order to achieve a better understanding and interpretation of certain material structures. The MD simulation is possible due to the advances in Physics Theory, Chemistry, Mathematics, and Computer Science. The MD simulation and visualization is able to render information about the evolution and behavior of the system. Furthermore, this physical computer simulation produces results on many microscopic properties of the structure and dynamics that are difficult to obtain by merely experimental methods in the lab. The main characteristic of this kind of simulations is computationally intensive, which pushes the power to the limit inside of the machine. This heavy workload is due to heavy and many computations per particle in the system.

3.1 General Description of MD Simulations

An MD simulation comprises the integration of Newton's motion laws, as well as the description of approximate force field generated based on the particle interactions. There are several

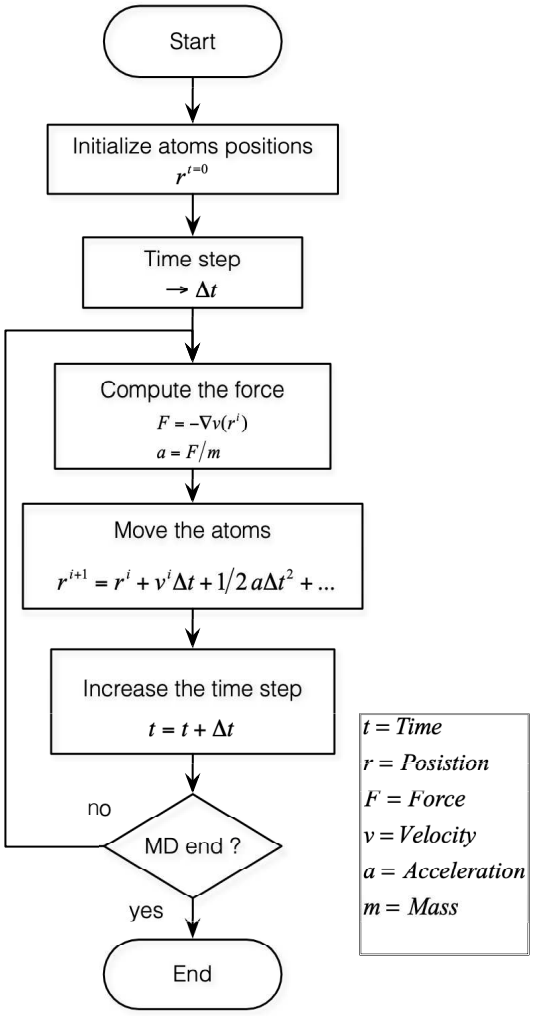

Figure 3.1: A general flow for a Molecular Dynamic simulation.

constrains in an MD implementation, such as the different force fields and limits of the system. Thus, there are many software implementations that offer many different capabilities according to the their specific algorithm. Some of them are ACEMD [67], OpenMM [68], NAMD [69], Amber[70], and CHARMM [71] to mention some of the most developed and up to date. Although all of them offer different capabilities, they follow a similar process which is described in Figure 3.1.

As we can denote, the MD simulation includes a numeric solution of the motion equations. This is performed by solving present forces that are residing on the atoms derivative from the potential energy of its 3 spatial components (x, y & z). The time between each interaction or *time step* is very small. Going from the order of $t \sim 10^{-3} - 10^{-6}$ seconds per step, which represents a few nanoseconds in real life.

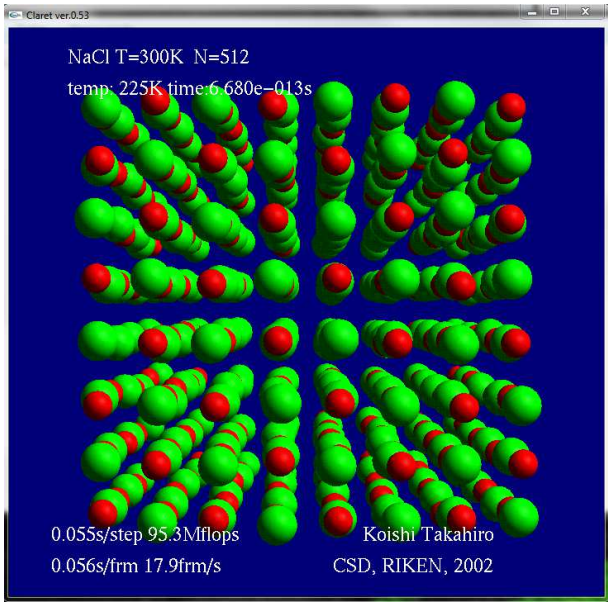

Figure 3.2: Image sample of Claret MD simulator.

3.2 Claret MD Simulation Software

It was first developed by Dr. Takahiro Koishi as an education purposed software. However, it was used to show the massive computational power of the *Molecular Dynamics Gravity Pipe* (MD-GRAPE 2) [75]. This special-purpose hardware allows a parallel implementation of the MD by using several processor units. This specialized hardware was first developed in The University of Tokyo [76] and lately taken by the Institute of Physical and Chemical Research (RIKEN) for further iterations.

Claret uses C/C++ as the implementation language and OpenGL as a rendering framework. The software includes MD-GRAPE libraries. However, as educational software, the versions and capabilities were changing gradually. Nowadays, claret is mainly used to understand basic MD between particles and also to learn parallel computing techniques. It is the main testbed for this dissertation. The original code is open source and it can be downloaded from the site of the author [77].

Claret MD simulation and visualization software include interactions between sodium (Na+) and chloride (Cl-) particles. This is basically a salt crystal in real life. However, in Claret, we can visualize its behavior at the atomic level. This can be appreciated in Figure 3.2. As we can denote, all the particles reside at the vacuum level, delimited by a cubic subspace. Some of the information in this version of the package includes a variation of the temperature and pressure. A limiting capability of the software is the particle boundary: if the crystal reaches its boiling or fusion steps, the particles are not able to escape from the wall, instead, the movement and force are changed in the opposite direction.

Real-Time Visualization of the interaction and behavior of the particles are done in Claret.

Input key	Description
q	Exit the program
v	Hide information on/off
t	Increase temperature 100K
g	Decrease temperature 100K
y	Increase temperature 10K
h	Decrease temperature 10K
!	Restart
z	Pause or Continue
s	Increase Time step by 10
c	Background color
M	New 27 ion for collision
N	New 4 ion for collision
m	New 1 negative ion for collision
n	New 1 positive ion for collision
1-9	Collision velocity OR number of particles
space	Shoot ion for collision

Table 3.1: Keyboard input list for Claret.

The software includes various capabilities such as the following:

- ☐ Visualization of the evolution of the system in Real-Time
- ☐ Different angle view
- ☐ Temperature change (- +)
- ☐ Adding collision from new Ion
- ☐ Rendering using textures and polygons
- ☐ System Status: Force performance computation and frames/sec
- ☐ Stereoscopic[1] view

These are the capabilities of the first version of the software. Thus, these features are enabled during compilation time through `# define` C directives. Extra rendering options such as rendering with polygons or textures, stereoscopic vision, and the usage of an external accelerator are enabled in the same way. Some other features inside the program can be modified once the program is launched. These options are managed by the keyboard. A list of these options is shown in Table 3.1. As we can denote, for the numeric keys 2 options are provided:

1. Select the velocity for collision on the newly generated ions
2. Select the number of ions present in the system

The number of particles in the system is computed as follows: If the key $P = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ then the total amount of particles, is $n = P \times P \times P \times 8$. Hence, 8 is the minimum and

[1]For 3D vision a special high frequency display and special glasses are needed.

```
1  void main (int argc, char** argv){
2
3  // OpenGL settings
4      glutInit (&argc, argv);
5      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
6      glutInitWindowPosition (100, 0);
7      glutInitWindowSize (500, 500);
8      glutCreateWindow ("Claret ver0.53");
9
10 // Main variables, constants, and memory allocation
11     init();
12     keep_mem();
13     set_cd( );
14
15 // OpenGL functions
16     glutDisplayFunc(display);
17     glutReshapeFunc(reshape);
18     glutMouseFunc(mouse);
19     glutMotionFunc(motion);
20     glutKeyboardFunc(keyboard);
21     glutIdleFunc(md_run);
22
23 // Main loop function
24     glutMainLoop();
25 }
```

Listing 3.1: C code for Claret main routine.

5832 is the maximum amount of particles. Claret was initially coded using C and OpenGL. In order to facilitate the implementation, we show all the main process and functions used in the main loop in the List 3.1.

As auxiliary library to handle windowing system, keyboard and other important functions inside the visualization, the auxiliary library *glut* [2] is used [78]. At first, the OpenGL state is initialized, creating an appropriate window. Settings such as windows size and buffer type are selected. Next, the initial state of the constants and variables are set: for example position, pressure and temperature. Main OpenGL functions can be described as follows:

- ☐ **Display** --→ Is in charge of all the rendering of polygons/textures that represents the whole simulation system
- ☐ **Reshape** --→ Computes the actual deformation, size and angle of the camera inside of OpenGL
- ☐ **Mouse** --→ Enables the mouse input which makes the camera to rotate
- ☐ **Motion** --→ Computes a new frame according to the new angle provided by the mouse motion
- ☐ **Keyboard** --→ Implements the actions provided in Table 3.1.
- ☐ **Md_run** --→ The core of the MD simulation, where the computation of the force, velocity and other constants are performed. This follows the general MD process, the one depicted in Figure 3.1.
- ☐ **Mainloop** --→ Keeps the simulation alive

[2] GLUT is an OpenGL auxiliary library that handles all the system-specific implications required for creating windows, initializing contexts, and handling input events.

	A $(10^{-19}J)$	$\sigma_i + \sigma_j$ $(\mathring{A})$	C $(10^{-79}\mathrm{Jm}^6)$	D $(10^{-99}\mathrm{Jm}^8)$
$++$	0.4225	2.34	1.68	0.80
$+-$	0.3380	2.75	11.20	13.90
$--$	0.2535	3.17	116.00	233.00

Table 3.2: Parameters of Tosi-Fumi potential for Na Cl MD Simulation. $B = 3.15\mathring{A}^{-1}$

The description above represents the implementation in the original version of Claret software.

3.2.1 MD Core Function

In order to describe the behavior between the particles inside of Claret, a force computation using a direct method is performed. The inter-ionic potential of a rigid-ion model proposed by Tosi and Fumi [79] is used as a force field between ions.

$$\phi_{ij}(r) = \frac{q_i q_j}{r} + A_{ij} B \exp\left[\frac{(\sigma_i + \sigma_j - r)}{\rho}\right] - \frac{C_{ij}}{r^6} - \frac{D_{ij}}{r^8} \tag{3.1}$$

This potential comprises a Coulomb term, a repulsion term, a dipole-dipole term, and a dipole-quadruple term, where q_i and q_j are electric charges and r represents the distance between them. It uses the constant parameters of Eq. 3.1 given by Tosi and Fumi. These constants are shown on Table 3.2. A wall boundary condition is adopted.

Initially, the system at vacuum level is equilibrated at $T = 300K$. The number of floating operations per time-step is given by $n \times n \times 78$, where n is the number of particles, and 78 is the total operations inside Equation 3.1.

Although the core function used in Claret is not as complex as other MD simulators such as Amber[70] or CHARMM [71], we are able to visualize the crystal structure evolution of Na Cl ions.

3.2.2 Interactive Capabilities

Claret presents the behavior of Na and Cl particles at a vacuum level integrating simulation and visualization at the same time. This approach is commonly referred to as computational steering. Generally, data analysis and visualization of computer simulations are performed after everything else is done. This leads in some cases to discover invalidating results or errors during the simulation just after the pre-processing is performed. In this way, combining visualization of the simulation at the same time not only presents the advantage of looking at the evolution of the system but as well make an adjustment on the way. Computational steering has been studied and used since the computational graphics become more accessi-

ble [72, 73, 74]. However, its need for extra computational power for rendering presents a challenge compared to conventional computer simulations.

Claret was developed in order to interact with the particle system in both ways, visualization, and simulation. A description of these capabilities are listed below:

- **Simulation Interaction** --→ On the simulation side, Claret is capable of of changing variables of the system such as the temperature. This allows changing the state on the conglomerate of Na Cl particles. As well, changing the number of particles and the possibility to shoot ions to observe a collision.
- **Visualization Interaction** --→ Claret offers the capability of changing the camera view angle in order to navigate to different spots in the simulation. We can freeze the simulation which is useful to look at the state of the whole system. Also, an effect on the temperature is visible on each particle. We can increase the time-step in order to delay visualization for longer simulations as well.

These capabilities allow the user to observe a phase transition between different temperatures on Na Cl. As well, we can observe the crystal formation in different angle views. On Claret, the visualization of this phenomenon is feasible by using accelerators such as GPU as we will discuss in further sections.

The frequency ratio of updates between simulation and visualization is fixed to be 'step'. If step = 100, the visualization is performed every 100 MD steps. Though, the camera itself can be changed independently to simulation steps. In principle, we used a fixed ratio between them for simplicity. Therefore, frames per second are important for visual interaction. Note that fps are also important for simulation interaction since the smooth steering of simulations requests it.

3.3 Claret Versions

Given that Claret was initially conceived as an educational software package, it has changed its original source code implementing new features. Some of these new changes include:

- New keyboard actions
- Adding information to the visualization
- Different force algorithm implementation
- Different hardware accelerator e.g. PlayStation 3 or GPU.
- Different rendering methods

There is no actual official record of the branching, but in this dissertation, we consider to include 5 major versions. The latest ones serves as the testbed for the experiments in Chapter 4 and 5.

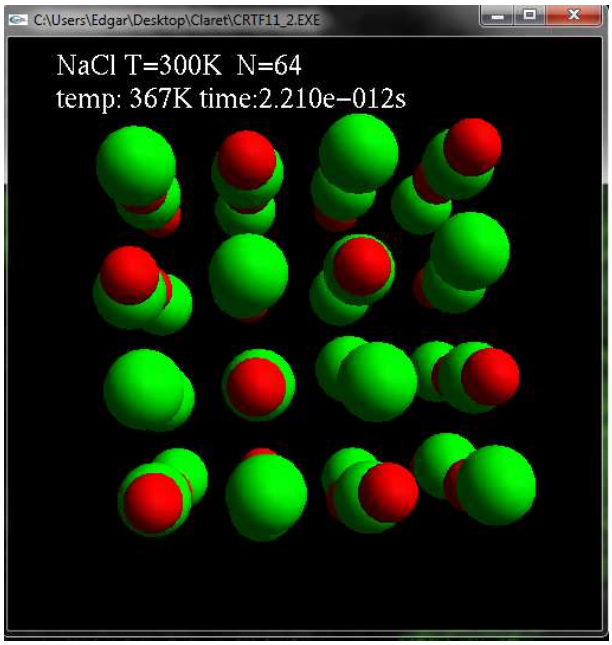

Figure 3.3: Sample image of version 0.11

3.3.1 Version 0.11

This version is the original one created by Dr. Takahiro Koishi. Figure 3.3 shows a sample image of the MD simulation. Version 0.11 includes some of the capabilities such as :

- ☐ Temperature in K scale
- ☐ Number of particles
- ☐ Time step

This version does not implement the cubic subspace, in other words, the wall is not present. The keyboard actions listed in Table 3.1 are the same. The rendering method uses polygons and the detail level can be changed by pressing the "R" key. All calculation process is done through CPU or MD-GRAPE devices. Nevertheless, the original repository for the source code is not available.

3.3.2 Version 0.53

In this iteration of the Claret MD simulator, the cubic sub space wall is present. Also, more information is added to the display. Version 0.53 was developed by its original author. The software can be found on this site [77]. Figure 3.4 shows a sample picture of this version. Some of the main capabilities of this version are shown below:

- ☐ Temperature on K scale
- ☐ Number of particles in the simulation
- ☐ Time step

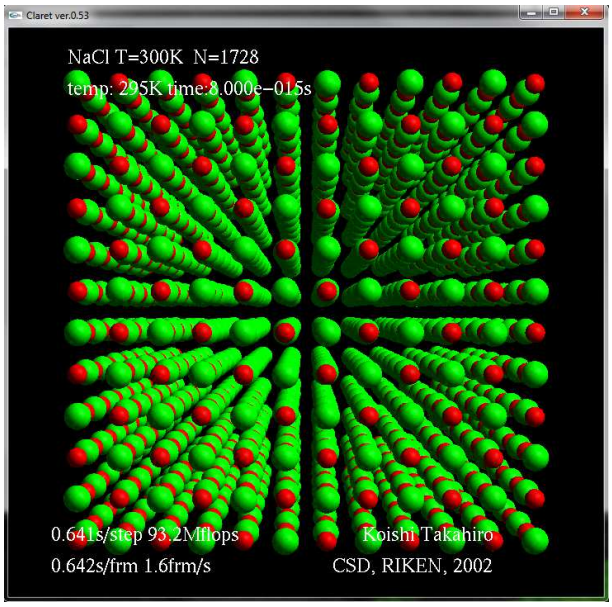

Figure 3.4: Sample image of version 0.53

- ☐ Flops measurement
- ☐ Frames per second

As for the rendering method in this version, polygons and textures are enabled. This version includes a stereoscopic view if the special hardware is present. The collision of new ions is possible as well. The force field between atoms can be performed by MD-GRAPE devices or CPU hardware.

3.3.3 Version 1.0

Version 1.0 was developed in Narumi laboratory from The University of Electro-communications. This version adds the GPU as a hardware accelerator using CUDA. Figure 3.5 shows a sample image of this version of Claret. Some of the new capabilities in this version are:

- ☐ Hardware accelerator.
- ☐ Temperature on K or C scale.
- ☐ Number of particles present in the simulation.
- ☐ Flops measurement.
- ☐ Time step.
- ☐ Rendering speed.
- ☐ Ion type.
- ☐ Ion charge.
- ☐ Pressure information.

Major changes in this version are the ability to switch between hardware acceleration:

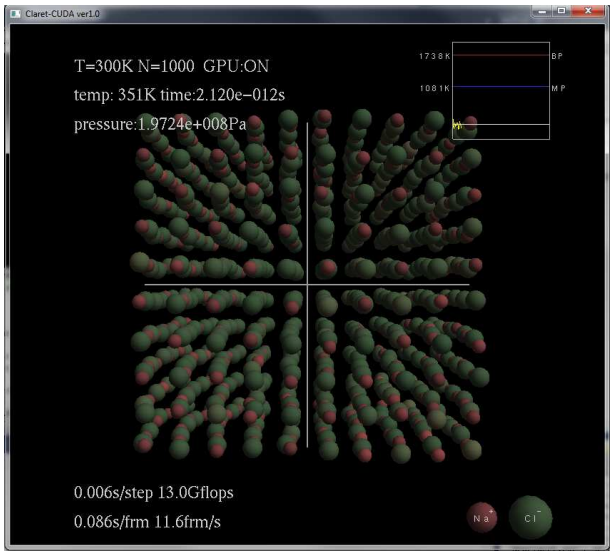

Figure 3.5: Sample image of version 1.0

GPU or CPU.

3.3.4 Version 2.0

The last version of Claret software is 2.0. This was developed by the author of this dissertation and it is the main application for the testbed in Chapter 4 and 5. This version was re-written in pure C++ code. As of version 1.0, this one supports GPU to compute the force between particles using CUDA. Figure 3.6 shows a sample image of Claret version 2.0. New capabilities include the following list:

- ☐ CPU implementation using OpenMP.
- ☐ GPU implementation using CUDA.
  - ◊ OpenGL interoperability for rendering.
  - ◊ Dynamic Parallelism for kernel launch type.
- ☐ Remote GPU execution.
  - DS-CUDA 2.5 compatible.
  - rCUDA 18.8 compatible.
- ☐ OpenGL 4.1 implementation.
  - ◊ Vertex and Fragment shaders.
  - ◊ GLFW as auxiliary library.
  - ◊ Render to custom frame buffer.
- ☐ Number of particles present in the simulation.
- ☐ Flops measurement.
- ☐ Time step.
- ☐ Rendering speed.

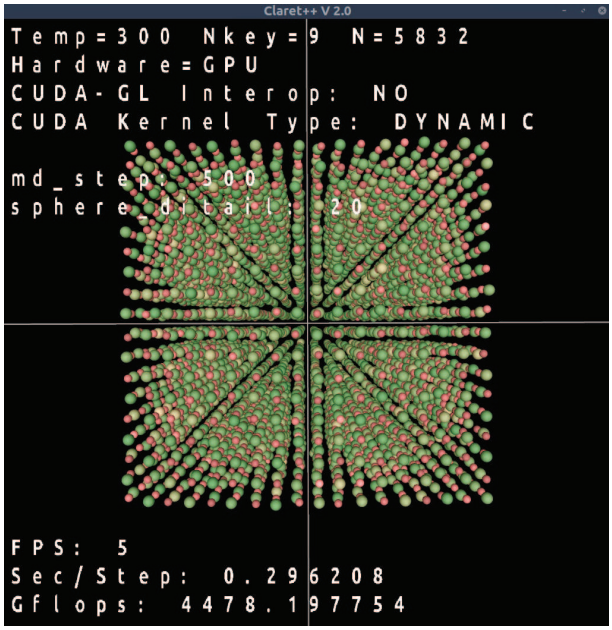

Figure 3.6: Sample image of version 2.0

- ☐ Polygon count per sphere.
- ☐ Ion charge.
- ☐ Pressure information.

We enhance this version of Claret with many new features. A basic idea of the force computation on CUDA is depicted in Figure 3.7. As well, on the CUDA side, we implemented OpenGL interoperability. This CUDA feature allows sharing memory space between OpenGL and CUDA context without double memory copies to the host. Thus, we keep the particle memory space shared between both contexts to alleviate the transfer bottleneck. Dynamic Parallelism over kernel launch was implemented over this version: this technique as is reported in Chapter 5, allows to reduce communication between host and client. We tested this version with DS-CUDA 2.5 and rCUDA 18.8 in order to use a remote GPU.

On the OpenGL side, we re-write the entire rendering engine. Before, Claret software it used OpenGL 1.x specification which does not allow to implement shaders or custom matrix states. This new version of Claret uses OpenGL 4.1, with the implementation of shaders in the vertex and fragment side. As well, we replaced GLUT for GLFW [86] which is a more capable and up to date utility library. Rendering particles were fixed to polygons and points. Lastly, we implemented a custom frame buffer to obtain the final image. This was mainly due to exploration for future works using coder/decoder inside of the GPU.

3.3.5 Android Version

The MD simulation is an interesting application that can benefit the user experience on a tablet [81, 82, 83, 84, 85] due to its touching capabilities and many sensors. A more dynamic

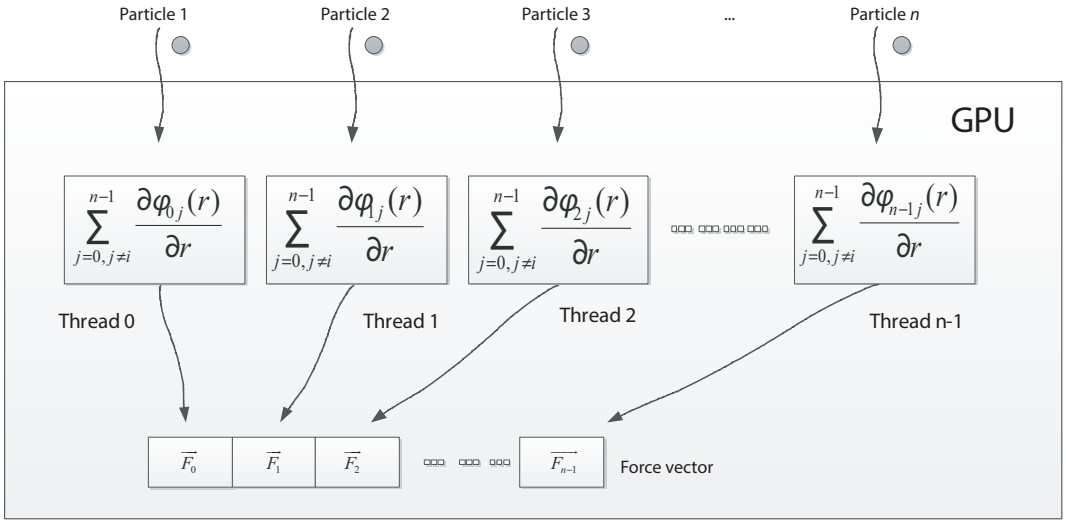

Figure 3.7: Force implementation on CUDA.

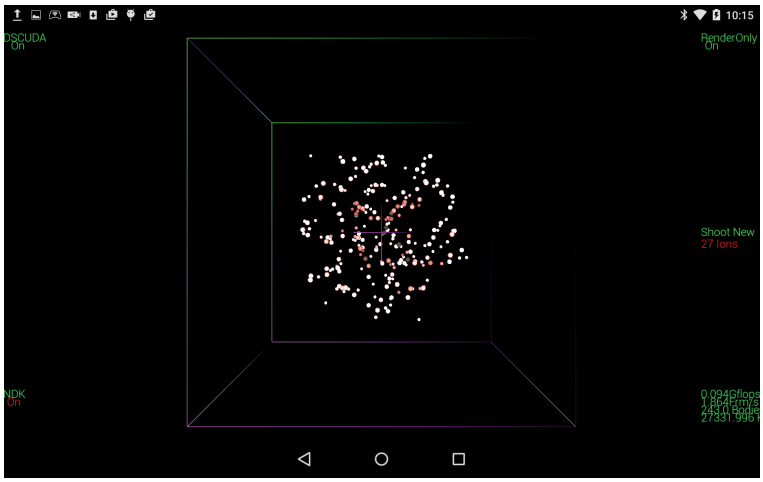

Figure 3.8: Sample image of Android version.

and immerse interface to interact with atoms is the aim of this version, as one of the main objectives of this dissertation is to enable compute-intensive applications on mobile devices.

In order to achieve a port from Claret PC version, we need to understand the technicalities involving rendering routines and software development. Claret for PC is a C/C++ based software that uses OpenGL as a rendering framework. Most of the original implementations are based on the OpenGL 1.x specification which lacks the *shaders* usage. Instead, it uses the fixed pipeline to render. Also, a freeglut library toolkit is utilized to handle windows and other interactive functions.

In this version of Claret for Android, we included 2 options as for medium of acceleration when the force between particles is computed: CPU, and remote GPU with DS-CUDA. We used the native tool NDK in order to port all the C code from the PC version. OpenGL is selected to render in this version as well. Specifically, OpenGL ES 1.1 is utilized due to the similarity of implementation against the PC version. Thus, the porting process becomes

Feature	OpenGL	OpenGL ES
Interface	WGL - Windows GLX - X11 Linux CGL - Mac OS	EGL
Utility library tool-kit	freeglut glut	glut - Java only
Rendering Routines	glBegin-glEnd glDrawArray	glDrawArray
Types supported	Float Double	Float
Main loop	glutMainLoop( )	onCreate( ) onPause( ) onResume( )
Font rendering	yes	no

Table 3.3: Technical differences between OpenGL / OpenGL ES on Claret port process.

more transparent and seamless. However, some minor differences between the implementation using OpenGL and OpenGL ES are noted. Table 3.3 shows these differences.

In the Android development ecosystem a class *opengl.GLSurfaceView* is provided in order to handle the content view inside the App. This auxiliary library is used to connect the OpenGL ES state to the Application state. Routines such as onCreate(), onStart() and onResume() from the Figure 3.9 are implemented within its equivalent *onSurfaceCreated()*, *onSurfaceChanged()* and *onDrawFrame()* on C through its proper interface using NDK. Next, we describe the process flow for each important routine in our port for Claret on Android.

- **onSurfaceCreated ( ) :** variables and constants are initialized. These variables include initial temperature, time step, velocity, force and position of the particles. State matrices for OpenGL and colors are initialized as well. Memory space is allocated.
- **onSurfaceChanged ( ) :** resizing of the canvas for the actual size of the Android tablet is performed here. On tablets, you may use it as portrait and landscape mode, which changes the total size for the main window buffer in OpenGL. Nevertheless, we restricted the usage as a landscape. The matrix model for OpenGL is defined here as well as the initial perspective. Buffer depth for color and spatial depth are cleared in this instance in order to generate a new frame.
- **onDrawFrame ( ) :** here we included all the rendering part. Basically, we implemented two main functions: One which is the core for the MD simulation where the force of the particles is computed and another function that renders all the position of the particles. The visual information such as the amount of floating operations per second is performed here as well.

Finally, for this version, we decided to use polygons and points to render the particles in the system. In the original code, textures and polygons are available for drawing. However,

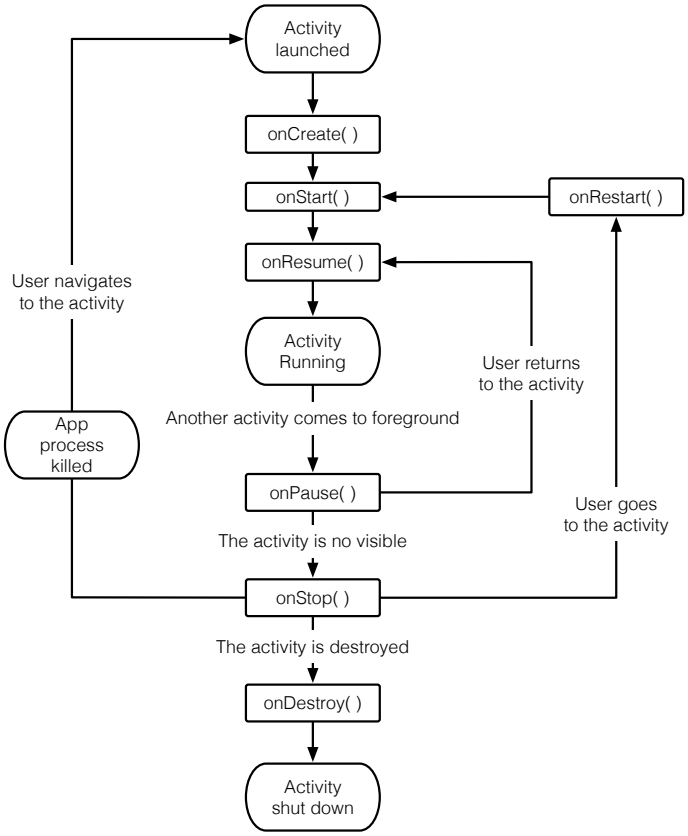

Figure 3.9: Life cycle of an Android application.

initially a function called *glutSolidSphere ()* from the GLUT library was used. Moreover, if we want to render a massive number of bodies without sacrificing performance in OpenGL [87], functions such as *glBegin-glEnd* should be avoided. Instead, *glDrawArrays* functions must be implemented.

4

Offloading with a naive approach: DS-CUDA case

Post-PC devices such as tablets and smartphones have become part of our daily lives. These mobile devices are changing the way users interact with computers and view data due to their many capabilities. By using such technology, interactive simulations have become a new way to artificially accelerate simulations by manually interacting with them. Mobile devices are suitable for such simulations because they have touch capability and multiple sensors. Nevertheless, mobile devices require more computational power to deliver the best user experience for such an intensive computational task. Cloud computing is another approach that can complement the low computing power of mobile devices. This is achieved by offloading intensive computations to a resource inside the same network. Cloud computing provides the ability to remotely connect with accelerators such as GPUs. To use graphics processors for GPGPU in a cloud environment, virtualization tools have been proposed, such as MGP [105], rCUDA [89] and DS-CUDA [108]. These tools can handle remote GPUs to accelerate applications and reduce code complexity. Specifically, DS-CUDA has proven to be a reliable and simple solution to handle remote GPUs while providing a fault-tolerant mechanism [94].

The main motivation leading this research is explained as follows: commonly, computer simulations are carried out without visualization. After the computation is done, all generated results are visualized and analysed on different work stations. The mobile computing devices, such as tablets, have shown better capabilities to interact with computers due to their touch screen capabilities and a variety of many other sensors. Nonetheless, the computing power of these devices is not sufficient enough to perform complex simulations such as Molecular Dynamics (MD). Consequently, we propose the implementation of a client and server scheme using a tablet and a remote GPU in order to perform real-time MD simula-

tion and visualization. We execute the entire simulation inside of the tablet and only the most computationally intensive parts are offloaded using a remote GPU through DS-CUDA framework [92].

Some other efforts have done to offload data and an intensive portion of computation from mobile devices to the cloud. Lin *et al.* [21], Elgendy *et al.* [22] and Kolb *et al.* [23] have proposed frameworks to offload computation from a mobile device to a server. Their frameworks consider different patterns to decide for offloading in order to save battery. However, they do not support CUDA for offloading. There have been some proposals to implement intensive applications on mobile devices held by parallel programming paradigms. Acosta *et al.* [24] implemented a particle filter running on Android using several parallel frameworks on such as RenderScript, OpenCL and ParallDroid. We used CUDA since its presence in HPC is clear [91] and DS-CUDA is able to handle CUDA code with mobile devices.

Our test system is composed of NVIDIA's "SHIELD" tablet, a notebook equipped with GeForce 970M GTX GPU, and an 802.11ac WiFi router. We also included NVIDIA's Jetson K1 an embedded system for comparison purposes. At the time of performing the experiments, this was the first CUDA capable chip for ARM devices. Details are described in a further section.

The rest of the Chapter is organized as follows. Section 4.1 includes a brief description of DS-CUDA as well as how we enable this virtualization framework on Android. Also, we include in detail each component of the system we used for the performance comparison. Section 4.2 is about the detail for each test we performed. In section 4.3 we present the results obtained from some experiments. Finally, in section 4.4, we discuss and summarise the contents of the Chapter.

4.1 Method

In this section, we present a general overview of DS-CUDA virtualization framework. We also include the procedure to enable DS-CUDA on Android tablets. Furthermore, a detailed description of each part of the test system is described.

4.1.1 DS-CUDA Overview

DS-CUDA is a framework that simplifies the usage of GPUs on a distributed network, rather than using native CUDA APIs. A single client node and one or more server nodes compose one DS-CUDA system, as shown in Figure 4.1.

The server nodes have one or more CUDA capable GPUs that are handled by server processes. An application on the client-side can use GPU devices to process data without having a physical GPU. The client program sees all GPUs contained in the server nodes as

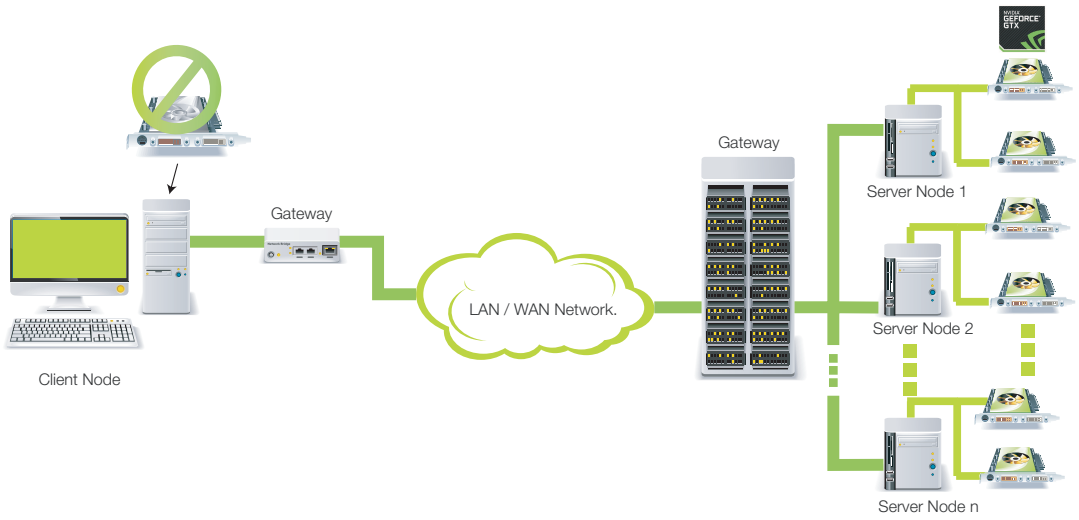

Figure 4.1: Diagram of a typical DS-CUDA system.

if they were actually attached to the client node. Therefore, DS-CUDA is a kind of GPU-virtualization tool at the source code level.

When the client program is compiled, native CUDA APIs are handled by a DS-CUDA pre-processor. The DS-CUDA pre-processor replaces them with corresponding wrapper functions. The substituted functions communicate with the server nodes through InfiniBand (IB-Verb) or TCP socket. The wrapper functions send the proper arguments and data to the server nodes and each server call the actual native CUDA APIs. Detailed implementation is explained in other papers [92, 93].

DS-CUDA has demonstrated good performance when multiple GPUs are used for MD simulation. Oikawa *et al.* [94] has conducted MD simulation with a replica-exchange method using more than 1000 GPUs. They concluded that increasing the number of MD steps lead to a better parallel efficiency even when Gigabit Ethernet was used.

4.1.2 DS-CUDA for Android

As we mentioned in the previous section, DS-CUDA is a GPU virtualization framework that works in the client-server scheme. On the server-side, where the physical hardware is located (the GPU), a *daemon* process is always listening for requests from the client. In order to generate the executable from the client-side, DS-CUDA pre-processor *dscudacpp* is used instead of *nvcc* compiler. This pre-processor is a Ruby script that replaces normal CUDA API calls to DS-CUDA ones. Figure 4.2 shows a simplified example of output files.

The *sample.cu* file includes the CUDA code of our application. This file is inserted into *dscudaccp* preprocessor. The output is composed by several files: the *sample.ptx* which corresponds to low-level code inside of the kernel and the *sample.ds.cup* which is a similar

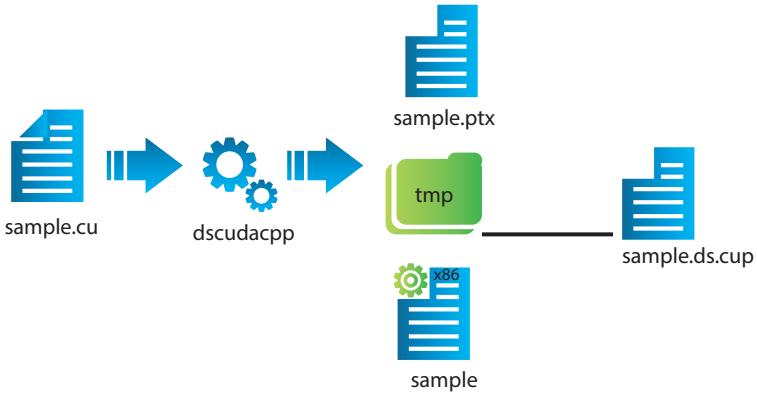

Figure 4.2: DS-CUDA pre-processor output example.

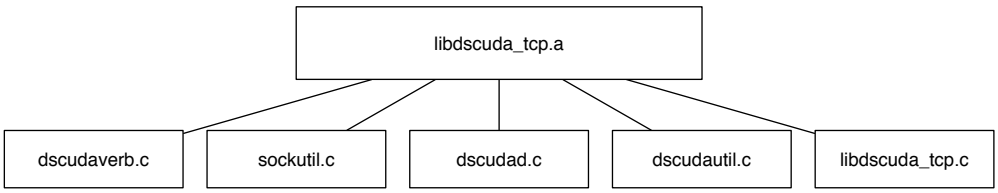

Figure 4.3: DS-CUDA client library code structure for socket communication through TCP protocol.

version of the original code but wrapping all the native CUDA functions with the DS-CUDA ones.

In order to generate the final executable, a static library is needed to be linked in the final phase. This library is the implementation of the CUDA APIs through socket calls. Figure 4.3 shows its code composition. In a normal scenario, this final phase will be handled by *dscudacpp* through *gcc* compiler. However, to generate an executable for the Android platform different tools are needed.

A native development tool is necessary to enable DS-CUDA for Android clients: the Native Development Kit (NDK) [95] allows the usage of C code inside of the Java main based program on Android devices. This framework and toolkit allow the usage of *gcc* compiler for ARM devices. Hence, we can use the compiler to generate the client library and handle the pre-process GPU code from *dscudacpp* as Figure 4.4 illustrate.

Two main *make* like files are required to generate and configure properly the NDK tool inside of the Android project. The first one, *Android.mk* is used to include source files, headers and some flags for the compilation phase. A sample is shown in List 4.1. The second one, *Application.mk* is used for platform-specific configurations, type of library to generate, architecture and some exceptions for the compiler. A sample of the file is included in List 4.2.

Finally, we can access the CUDA APIs from the Java code through the Java Native

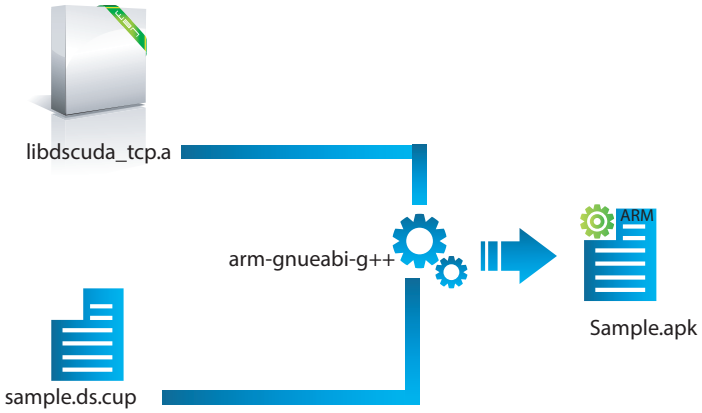

Figure 4.4: Final client compilation phase for Android application using NDK.

```
## Android.mk
## Static Library libdscuda_tcp.a
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE  := dscuda_tcp1.5.2

LOCAL_CFLAGS := -O0 -g -ffast-math -funroll-loops -I. \
-I/usr/local/cuda/include \
-I/usr/local/cuda-6.0/NVIDIA_GPU_Computing_SDK/C/common/inc \
-I/usr/local/cuda/samples/common/inc -DTCP_ONLY=1
LOCAL_SRC_FILES :=  dscudaverb.cpp dscudautil.cpp \
sockutil.c libdscuda_tcp.cpp \
LOCAL_LDLIBS := -ldl -llog
include $(BUILD_STATIC_LIBRARY)
## Static Library DS-CUDA Routine
```

Listing 4.1: Configuration file (Android.mk) sample to generate DS-CUDA static library.

Interface (JNI) [96] which can load C/C++ functions.

4.1.3 System Description

In Figure 4.5 our testbed system for simulations is shown. We utilized a mobile GPU GeForce 970M GTX from a notebook as a server. There are two methods to communicate between the client and the server: Gigabit Ethernet or WiFi 802.11ac. As for the router and access point, we used a Buffalo AirStation MZR-1750. The full characteristics of the server and client are listed in Tables 4.1 and 4.2, respectively.

For comparison purposes, we also included an embedded system powered by a mobile CUDA capable GPU. The full characteristics of the system are shown in Table 4.3.

4.2 Test Description

In this section, we present the details for each configuration test. Three different assessments are proposed in order to evaluate different metrics. First, a bandwidth test to measure communication performance between a client (Tablet) and a server (GPU). When DS-CUDA is

```
1 ## Application.mk
2 APP_MODULES    := dscuda_tcp1.5.2
3 APP_ABI        := armeabi
4 APP_PLATFORM   := android-18
5 APP_STL        := gnustl_static
6 APP_GNUSTL_FORCE_CPP_FEATURES := exceptions rtti
7 APP_OPTIM      := debug
```

Listing 4.2: Configuration file (Application.mk) sample to include DS-CUDA static library.

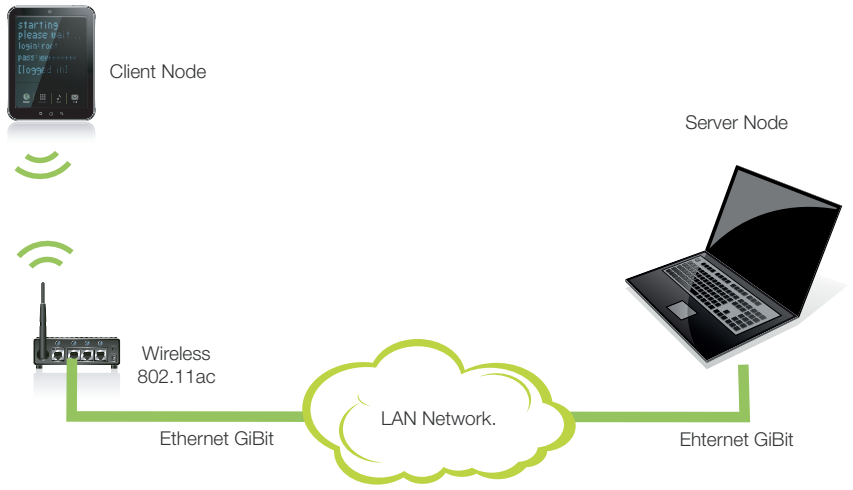

Figure 4.5: Test bed system for a DS-CUDA proposal.

utilized instead of native CUDA, there is some overhead in communication since DS-CUDA's wrapper functions substitute original CUDA functions. Furthermore, different mediums to communicate CPU and GPU are used, e.g. PCI Express in the case of a Notebook using native CUDA, and Ethernet and WiFi in the case of using DS-CUDA wrapper functions. The second test consists of a simple matrix multiplication to measure a simple latency when a GPU kernel is launched. Also, it is used to verify the computation saturation point of the GPU. Finally, for the third test, MD simulation and visualization are performed. This test aims the measurement of computation performance, communication overhead between client and server, and graphics rendering bottleneck.

Element	**Description**
CPU	Intel Core i7-4720HQ, 2.60 GHz, 8 Cores
GPU	GeForce 970M GTX , 1920 CUDA Cores
OS	Ubuntu 16.04 LTS x86
CUDA	Driver 352.55, Toolkit 6.0, SDK 6.0

Table 4.1: Server specifications. Notebook powered with NVIDIA's 970M GTX GPU.

Element	Description
CPU	NVIDIA Tegra 4, 1.912 GHz, 4 Cores
GPU	NVIDIA AP, 72 Custom Cores
OS	Android 6.0, Tegra for Android 3.0r3

Table 4.2: Client specifications. NVIDIA tablet "Shield Portable".

Element	Description
CPU	ARM cortex A-15, 2.32 GHz, 4 Cores
GPU	Tegra K1 , 192 CUDA Cores
OS	Linux for Tegra - Ubuntu 16.04 for ARM
CUDA	Custom Jetson K1, Toolkit 6.0, SDK 6.0

Table 4.3: Embedded system Jetson K1 powered with NVIDIA's Tegra GPU.

4.2.1 Bandwidth Test

We performed tests to measure data transfer speed between client (tablet) and server (GPU) via *cudaMemcpy* function. Two options for memory copy functions are considered, i.e. from Host to Device (H2D) and from Device to Host (D2H). The size of transfer data is increased from 1 KB to 268 MB. We tested for four different settings: 1) Native CUDA on a notebook, 2) Native CUDA on a K1 embedded system, 3) Ethernet connection on a DS-CUDA system and 4) WiFi connection on a DS-CUDA system.

4.2.2 Matrix Multiplication

A simple matrix multiplication code was implemented. Two matrices A and B are full with random floating-point numbers and matrix C is the result of their multiplication. The CUDA code for the kernel used in this test was taken from Nvidia's SDK CUDA 6.0 as a reference. The most naive implementation which does not use cuBLAS [1] library was used. Nevertheless, this kernel implementation uses shared memory and it is optimized for GPUs with 192 CUDA cores in SM. In our test, both devices equipped with a GPU have a multiple numbers of 192 as shown in Tables 4.1 and 4.3. The matrix size (width and height) for each input matrix (A and B) is set as follows: $W_A = 128 * i$, $H_A = 192 * i$, $W_B = 128 * i$, $H_B = 128 * i$. W_x means width of the matrix X, and H_x is height of matrix X. Here we defined $i \rightarrow \{1, 5, 10, 15, 20\}$ as the scaling factor. In this test, only the time for kernel execution is measured.

4.2.3 Molecular Dynamics Simulation and Visualization

As we mentioned in Chapter 3, MD simulation is used in computational science to describe physical phenomena at the atomic level. From the computational point of view, these kinds of

[1] The cuBLAS library is an proprietary implementation from NVIDIA of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA runtime.

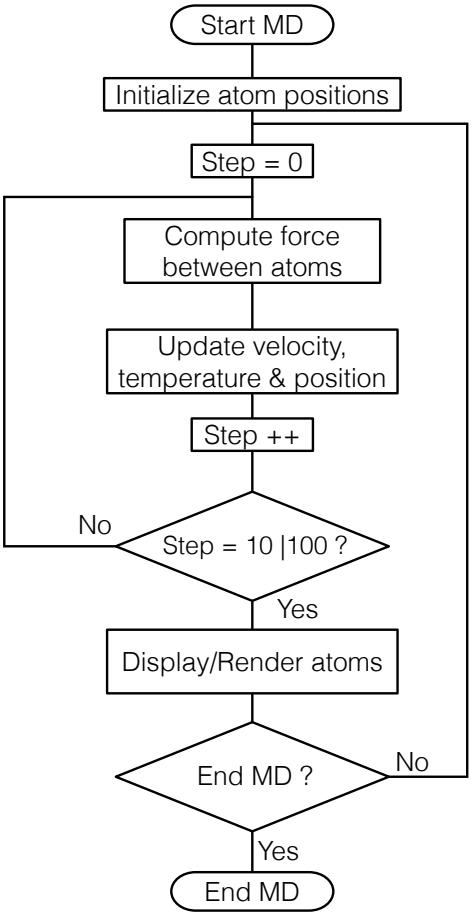

Figure 4.6: Simplified schematic algorithm of MD simulation. Step for simulation before rendering can be switched to 10 or 100.

simulations are very intensive due to its $O(n^2)$ complexity where n is the number of particles. A simplified MD algorithm used in this test is shown in Figure 4.6.

We implemented this algorithm for the tablet, notebook and the embedded system. Initially, a particle conglomerate of NaCl is shown and its behavior under the vacuum level is simulated. Tosi-Fumi potential [117] is used to describe the interaction between atoms. This potential, as shown in Eq. (3.1), describes a Coulomb term, a repulsion term, a dipole-dipole term and a dipole-quadruple term.

When we convert a serial version [77] of the MD program to GPU version, a general idea of CUDA implementation is as follows. In order to compute Eq. (3.1) for all bodies in the system, we allocate all constant parameters inside of the constant memory and send a fraction of positions of particle j and charge q_j to the shared memory. Thus, we update the partial force for particle i within each block of threads and keep this result in the shared memory as well. Finally, we apply a reduction sum in each thread block to obtain the complete force for each particle. However, we do not send back the results to CPU every step. Instead, we send back the results every 10 or 100 steps for each rendering.

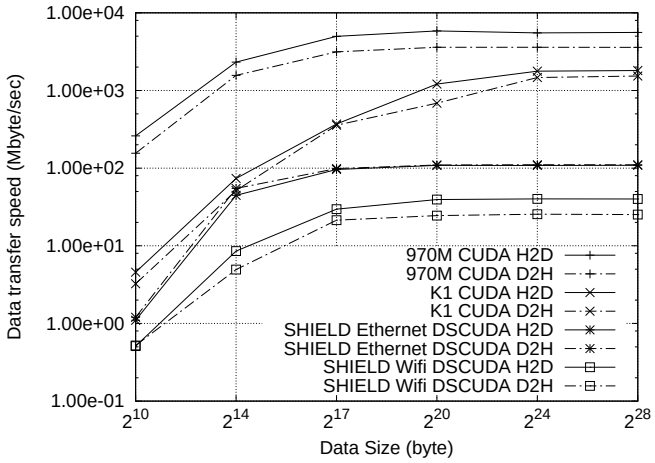

Figure 4.7: Data transfer speed using CUDA's *cudaMemcpy* function over different types of connection. H2D means Host to Device direction and D2H is opposite.

To implement the visualization side, we used OpenGL 3.0 for Linux based machines and OpenGL ES 1.1 for Android. A single dot is used for the representation of each atom in the simulation. An important thing to denote is that we disable vertical synchronization (Vsync) on OpenGL in order to print out the actual amount of frames per second for the application. This was only possible in Linux based systems through an variable *vblank_mode* set to 0. For the implementation of Android, we could not disable the Vsync because the control of this function is fixed by the specific display vendor.

4.3 Results

This section presents the results obtained from our test over DS-CUDA system using tablets for offloading. Computation performance from a matrix multiplication and MD simulation and visualization are shown. As well, communication throughput using different physical media is included.

4.3.1 Bandwidth Performance

Figure 4.7 shows the data transfer speed between Host and Device performed by CUDA API *cudaMemcpy()*. Data transfer speed ($Throughput$) is calculated from data size ($DataSize$) divided by the time ($Time$) for data transfer.

First, we report on the performance of the notebook. In this case, the internal GPU 970M communicates with the CPU using PCI Express Gen 3x16. The top speed on H2D is 5.5 Gbytes/sec, and 3.5 Gbytes/sec on D2H. These numbers are rather expected to take into account the bus connection from the PCI Express. The second case is the embedded system Jetson K1 which uses on chip communication for sharing resources between CPU and GPU.

	H2D latency (sec)	**D2H latency (sec)**
970M CUDA	3.7×10^{-6}	6.3×10^{-6}
K1 CUDA	2.2×10^{-4}	3.2×10^{-4}
SHIELD Ethernet DSCUDA	9.2×10^{-4}	8.4×10^{-4}
SHIELD WiFi DSCUDA	2.0×10^{-3}	1.9×10^{-4}

Table 4.4: Memory copy latency of CUDA and DS-CUDA.

It reaches a top speed of 1.8 Gbytes/sec for H2D configuration, and 1.5 Gbytes/sec when D2H is performed. Here, we can denote that the speed in both ways is similar, compared to the notebook in which case D2H presents slower performance. Third is the case for the tablet using DS-CUDA over Gigabit Ethernet and WiFi. Implementing Ethernet we reached a top speed of 108.8 Mbytes/sec on H2D, and 110.3 Mbytes/sec on D2H. Utilizing WiFi we got a top speed of 40.1 Mbytes/sec on H2D, and 25.2 Mbytes/sec on D2H. Comparing the results using DS-CUDA against native CUDA, we can see almost 50 times slower against the case of Ethernet, and almost 100 times slower communication compared with WiFi implementation.

To estimate communication time within the DS-CUDA application, latency is relatively important because the GPU is connected through a network. For this purpose, we assume the data transfer $Time$ as follows:

$$Time = Latency + \frac{DataSize}{Bandwidth} \tag{4.1}$$

where $Bandwidth$ is the maximum data transfer speed when the data size is large enough. $Latency$ is the time needed to initiate or finalize the communication. As $Bandwidth$ is roughly the same as the maximum data transfer speed ($Max.Throughput$) in Figure 4.7, $Latency$ can be calculated as follows:

$$Latency = \frac{Min.DataSize}{Min.Throughput} - \frac{Min.DataSize}{Max.Throughput}. \tag{4.2}$$

Table 4.4 includes the latency of *cudaMemcpy* for both cases, H2D and D2H. In communication performance, CUDA achieves higher transfer speed and less latency in both the notebook with 970M and the embedded system K1. Using DS-CUDA through Ethernet and WiFi has a penalty in transfer speed and latency. However, as shown in Table 4.4, latency between Host and Device is similar to CUDA on the SHIELD tablet when DS-CUDA is used through Ethernet and WiFi.

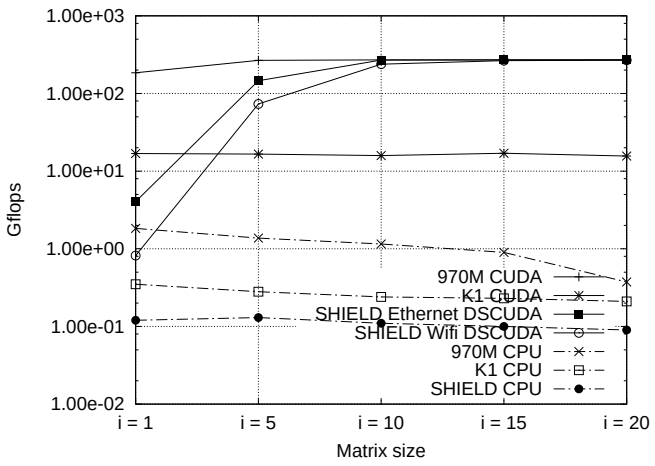

Figure 4.8: Computation performance for Matrix multiplication test. Horizontal axis shows the i scaling factor which defines the size of the matrices. Results are shown using Giga floating point operations per second.

4.3.2 Matrix Multiplication Performance

In this test we consider the amount of floating-point operations per second (flops) in our matrix multiplication sample. This is given according to Eq. (4.3):

$$flops = 2 * WA * i * WB * i * HA * i/time. \tag{4.3}$$

We show the complete results in Figure 4.8. The notebook and Jetson K1 using native CUDA on the GPU achieve a maximum of 271.2 and 16.90 Gflops, respectively. In both cases, constant performance is noticed because of the full usage of multiprocessors in the GPU at all times. The SHIELD tablet with DS-CUDA using Ethernet and WiFi achieves the same performance as the notebook for large matrix calculation. A performance difference is perceived between the notebook and DS-CUDA cases for smaller matrix sizes ($i < 10$).

The best CPU results from the notebook, K1, and SHIELD tablet are 1.8, 0.34, and 0.12 Gflops, respectively. We used only a single thread for CPU implementation in this test. These results are considerably lower than those utilizing the GPU.

On the DS-CUDA cases, the performance presented is lower for smaller matrix sizes because of communication latency takes a longer time than the actual computation. Calling the kernel over Ethernet and WiFi took 1.6 ms and 7.7 ms, respectively, while the matrix calculation itself took only 23 μs for the smallest matrix size of $i = 1$. For a medium-sized matrix, where $i = 10$, the calculation took 23 ms, greatly reducing the latency effect.

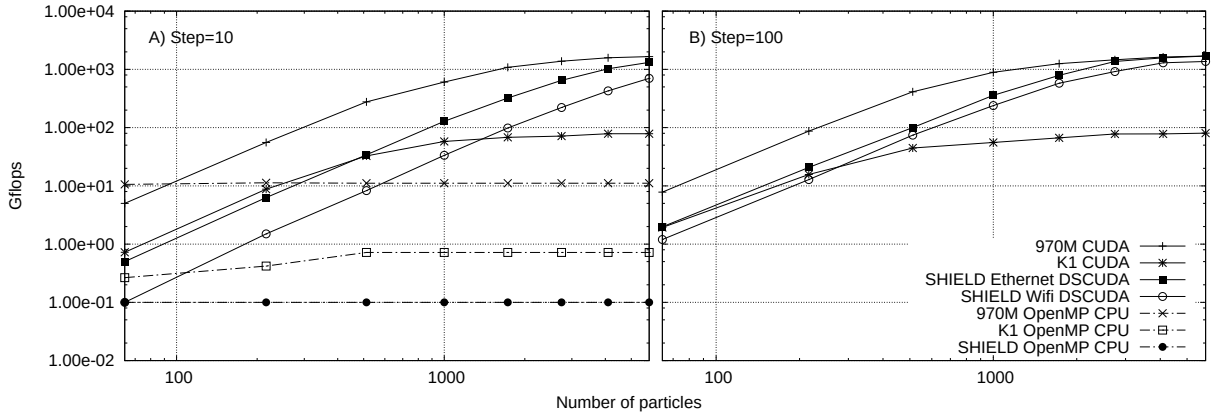

Figure 4.9: Computation performance for MD simulation and visualization test. Performance to compute force between particles for every 10 steps A) and 100 steps B) are reported. Results are shown using Giga floating point operations per second.

4.3.3 MD Simulation and Visualization Performance

Two kinds of results are presented for the MD simulation and visualization; performance of calculating force between particles and frame rendering performance.

Computation Performance

The first section shows the number of flops when solving Eq.(3.1). The positions of atoms are internally updated every step and rendered to the screen every 10 or 100 steps. To calculate the number of operations per second inside the MD simulation, Eq.(4.4) is used.

$$flops = (n * n * 78 * step)/time, \tag{4.4}$$

where n represents the number of particles in the system. There are 78 operations required to solve the potential between a pair of particles. *Step* represents how often the system is updated to render one frame, as shown in Figure 4.6.

First, we present the computation performance (Gflops) for $Step = 10$ in Figure 4.9 A). The notebook and K1 embedded system using CUDA achieve a maximum of 1,655.3 and 78.5 Gflops, respectively, for a large number of particles. The SHIELD tablet using DS-CUDA with Ethernet and WiFi accomplished 1,319.6 and 701.2 Gflops, respectively, for a large number of particles. The SHIELD tablet outperforms the K1 embedded system when the number of particles exceeds 1,728. Fewer particles affect the performance of DS-CUDA owing to communication latency between Host and Device.

Second, Figure 4.9 B) shows the performance of computing force between particles when $Step = 100$. In this case, only the GPU results are plotted because it is expected that CPU

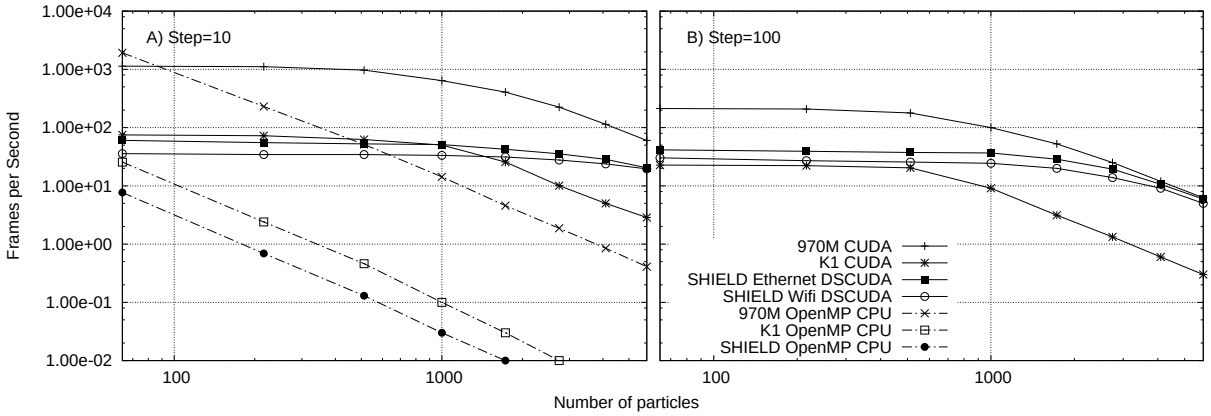

Figure 4.10: Visualization performance for MD simulation. Performance to render one frame for MD is reported. The number of steps to update the system was set to 10 steps A) and 100 steps B). Results are shown using frames/second.

results would be similar to Figure 4.9 A). The notebook and Jetson K1 using CUDA achieve 1,698.7 and 80.47 Gflops, respectively. SHIELD tablet using DS-CUDA with Ethernet and WiFi reaches 1,692.4 and 1,368.4 Gflops, respectively. As we can observe, the results for both CUDA implementations remain similar when we change the number of steps. However, for the DS-CUDA implementation, communication between Host and Device is reduced by increasing the number of steps from 10 to 100.

Frames per Second

The following section shows the number of frames per second. The main difference between this test and the computation performance is as follows: this includes computation time and also time to render the particles in the system.

Figure 4.10 A) shows the performance to visualize the MD simulation for $Step = 10$. The notebook and Jetson K1 using CUDA reached 60.24 and 2.86 frames/sec, respectively for a large number of particles. The SHIELD tablet using DS-CUDA with Ethernet and WiFi achieved 20.46 and 19.61 frames/sec, respectively.

Figure 4.10 B) shows the rendering performance for $Step = 100$ configuration. Only the GPU results are included at this time. The notebook and K1 using CUDA achieve 6.25 and 0.30 frames/sec, respectively, for a large number of particles. The SHIELD tablet using DS-CUDA with Ethernet and WiFi achieve 5.92 and 5.00 frames/sec, respectively. In this case, increasing the number of steps from 10 to 100 causes the GPU to take more time to compute the force between particles. Thus, the rendering process for each frame becomes relatively slow. Communication and rendering become less of a bottleneck compared with the actual MD simulation. Results with $Step = 10$ and $Step = 100$ were compared in this study. The main reason is to show the effect of reducing communication between Host and Device.

This is a well-known technique among experts on GPGPU for CUDA programming because copying data from the CPU to the GPU is a very expensive time-consuming operation.

Next, we report the numbers from the CPU implementation. In this case, OpenMP is used to compute the force between the particles. The outcome of this experiment is plotted in Figure 4.10 A). The notebook reaches 0.41 frames/sec for a large number of particles. The Jetson K1 and SHIELD tablet accomplish only 0.026 and 0.011 frames/sec for 1,728 particles. For a smaller number of particles, using the CPU as a force accelerator is the best visualization option because it excludes the communication bottleneck between the CPU and GPU. However, for a larger number of particles computing force between atoms becomes the bottleneck. In this case, GPU becomes the optimal solution.

Effects of the communication can be observed in Figures 4.9 and 4.10. Here, we denoted that the communication frequency is reduced to 1/10 when $Step = 100$ is used, compared with $Step = 10$. As we can note, the DS-CUDA performance is low for a small a number of particles because of network overhead. Nevertheless, hiding this latency was possible by increasing the number of steps in the MD simulation to keep the GPU busy on the server-side. From the results, we showed that the number of steps to update the system directly affects the frames per second. For $Step = 10$, the frames per second for the DS-CUDA system reached more than 19 frames/sec. However, increasing the number of steps to 100 directly affects rendering time. Importantly, the Jetson K1 could not handle more than 3 frames per second for the larger number of particles. This was owing to a combination of fewer flops and poorer rendering performance compared with the tablet-notebook combination.

4.4 Conclusion

In this Chapter, we demonstrated that intensive computations are accelerated on a non-GPU tablet using a remote low-power integrated GPU through a DS-CUDA framework. This was possible because of DS-CUDA's ability to virtualize GPUs in a cloud environment. However, communication between a client tablet and a server notebook with a GPU might become a performance bottleneck. Therefore, we compared the performance results of our system against the Jetson K1 which was the first embedded system equipped with a mobile GPU. Our system achieved better computational performance and better frames per second.

We tested the DS-CUDA framework to facilitate and enable the development of remote offloading using mobile devices. Using the same code as native CUDA, the DS-CUDA pre-processor replaces the CUDA APIs with wrapper functions that implement the connection between client and server. In this sense, a remote GPU located in the cloud can be pulled and looked at as if it were attached to the mobile device.

We show an MD simulation and visualization including several hundred particles in the system. However, in order to increase the size of the simulation, a similar approach from previous DS-CUDA implementations could be followed. It has been proven that DS-CUDA can be used in a multiple GPU environment for MD simulation. Nevertheless, latency and communication between nodes could become a bottleneck in our proposed system.

Our heterogeneous system proved to be suitable for executing an interactive molecular dynamics simulation. Using the DS-CUDA virtualization framework, only kernels for intensive computation are offloaded to the server-side. Mobile devices are not expected to perform intensive computations due to saving battery life and low powered CPU. However, cloud computing or similar systems like ours are an interesting approach to simultaneously achieve more computational power on mobile devices.

5

Reducing communication latency through Dynamic Parallelism: rCUDA case

Interactive modeling, such as interactive Molecular Dynamics (MD) simulations [100, 101], enables the artificial acceleration of simulations through manual interaction. Mobile devices are suitable for such simulations because they have touch capability and multiple sensors. Nevertheless, mobile devices require more computational power to deliver the best user experience for such intensive computational tasks, because simulations like these are characterized by high frame rates and processor-intensive routines.

Cloud computing provides the ability to remotely connect with other machines and hook up accelerators like GPUs. In a cloud environment, virtualization tools such as GVirtuS [102], Shadowfax[103], GPUvm[104], MGP [105], vCUDA [106], GridCuda [107], DS-CUDA [108, 11], and rCUDA [109, 110] have been proposed in order to use remote GPUs. These tools and frameworks are able to manipulate remote GPUs to accelerate applications in a cloud environment. In particular, rCUDA has proven to be a reliable, simple, and up-to-date solution for handling remote GPUs [111, 112, 113, 114]. In the previous Chapter, we were able to use DS-CUDA as a medium of connection between an Android tablet and a remote GPU from a notebook. However, as we will see in section 5.5, the performance delivered from rCUDA overpass the one from DS-CUDA.

We analyze the computing, rendering, and power efficiency when the rCUDA framework is used to accelerate computations on a mobile device, offloading most of its intensive computations to a notebook leveraged by a low-power GPU. As we can denote, despite their great acceleration and high performance, desktop GPUs are considered non-green computing solutions since they consume around 270 W [115], substantially more than the low-power

GPUs present on notebook computers which are in order of the 170 W. This is because of they are designed for energy efficiency and low power use [10, 116].

Compared to the previous Chapter, here we present a performance evaluation of a heterogeneous system composed of a non-CUDA-capable tablet device and a notebook powered by a low-power GPU. We show the effectiveness of using GPGPU techniques such as Dynamic Parallelism (DP) to reduce the kernel call latency. As well, we investigated using a server/client scheme. Moreover, the possibility and outcome of increased power efficiency using various clients are shown.

There have been some proposals that implement a paradigm similar to ours. Fatica *et al.* [25] implemented a synthetic aperture radar imaging application using a Tegra K1, which is a CUDA-capable GPU. In both cases, speed improvements were achieved implementing the GPU compared against CPU implementation. However, the authors on their study did not include any outcomes on performance per watt or battery life. Heungski *et al.* [28] and Kemp *et al.* [27] conducted a set of a test similar to ours. The main difference between our approach and that of Heungski *et al.* is the API used for offloading. They chose OpenCL, because it is open source and covers more devices to offload, whereas we use CUDA because of its presence in HPC is clear [99] and rCUDA is able to handle CUDA code. Kemp also used rCUDA to offload intensive computations to mobile devices. Our proposal is related to theirs in the sense that we both claim speed gains when heavy parts are offloaded for certain applications. Additionally, both proposals present results about energy usage. However, their study shows that for exposure fusion algorithm on images there is no lead to better execution or saving power consumption. The main reason is that they used CPU on the client side (Tablet) for image compression and thus the amount of data sent to remote GPU is reduced. We were able to tackle the communication problem in a different way. We implemented Dynamic Parallelism to reduce GPU kernel calls. Also, they consider only client-side power consumption, whereas we include both client-and server-side consumption for performance per watt measurements. Furthermore, we also examine the power efficiency for combinations of multiple clients. Another study related to low-power systems is that of Reaño *et al.* [30], who investigated the performance of rCUDA on a combination of low-powered CPUs such as ARM, Atom, and Xeon D. They used the GROMACS package to conduct MD simulations and concluded that the acceleration and handling of the virtual GPUs by the Xeon D processor was superior to that using the ARM or Atom. However, they did not present any power consumption results. Montella *et al.* [29] proposed to use offloading for heavy computations from an ARM cluster (Client) composed by 3 NVIDIA Jetson TK1 utilizing GVirtuS framework to a remote GPU TITAN X (Server). Although, Jetson TK1 contains on SoC with a CUDA capable GPU, they offload several sizes of matrix multiplications to a server and compared the results against the local execution, claiming

gain in performance when offloading. Furthermore, they report that latency is neglected as the problem size increases. Despite the similitude to our proposal, this study does not tackle a real-time application, including several copy memory functions or kernel calls. Moreover, they do not include power metrics between server or client, even thought GPUs such as TITAN X are very power-hungry, consuming around ∼250W. In our study, we selected a low-power client and server since we want to squeeze every Gflop/Watt delivered from the system.

The contents of this Chapter is organized as follows. Sections 5.1 and 5.2 provides an insight into how the strategy for implementing DP was decided to compare results from the previous Chapter. Section 5.3 provides a brief description of rCUDA, before Section 5.4 describes each component of the test system. In Section 5.5, we present the results obtained from a series of tests. Finally, in Section 5.6, we discuss and summarize the contents.

5.1 Communication Optimization Policy

In the previous Chapter, we were able to offload intensive computations from a mobile device to a remote GPU through virtualization framework DS-CUDA. Our results showed that the communication between the server and client still presents a constraint in order to achieve enough frame rate even though a significant gain of computation speed was observed. Optimizing communication can lead to various approaches. Our target application is designed for real-time visualization and simulation. Thus two factors are important:

- **Bandwidth** ⇢ This factor is related to the total information transfer between server and client. Upgrading the medium of connection is always a reasonable solution. Furthermore, reducing the data size in the application is crucial.
- **Latency** ⇢ This factor refers to the amount of communication between server and client including the number of kernel calls and the memory copies. Although latency is directly related to the medium used to communicate both entities, we can alleviate this constraint by reducing the number of calls.

In our system proposal, the MD simulation and visualization from the tablet device needs to overcome these two factors. A better way to understand the strategy for optimization in our system proposal is shown in Table 5.1. This reference to optimization strategies shows the total communication between GPU and CPU in a local machine environment. However, we can observe the reduction for communication and data transmission which is our primary target. In the first case, only force is performed in the GPU, thus only one kernel call per MD step is performed. However, the amount of transfer data is set to 18 (9 from CPU to GPU and 9 from GPU to CPU) for each MD step. This is due to the Integration part of

Optimization	**Kernel call**	**cudaMemcpy**
Only Force	$step \times 1$	$step \times (9 + 9)$
Force + Integration	$step \times 4$	2
Force + Integration + DP	1	2
Force + Integration + DP + Interop.	1	0

Table 5.1: Communication optimization strategy for Claret using GPU. The number of Kernel and memory copy calls are reported. Variable *step* refers to how often the MD simulation is executed during one frame. In our experiments it is set to few hundreds.

velocity, temperature and other variable calculations are done in the CPU. Therefore excessive communication is observed between CPU and GPU.

The first optimization is to perform the Integration part in the GPU, thus communication for memory copies are reduced. However, the number of kernel calls per MD step is increased to 4 per MD step. We can denote that here, all the data necessary to compute an MD step still remains in the GPU. Only data necessary to render is sent back to the CPU. In this case, only position and velocity variables are sent back.

The second optimization, which is the main objective of analysis in this Chapter is including DP. This will allow reducing the kernel call per MD step to only 1. DP allows to wrap the kernel inside the kernel, then our MD step loop is located inside the GPU code. This approach executed locally in a GPU presents a slow execution. However, we expect to increase performance when the server and client scheme is used since the number of communication for kernel call is reduced.

The last optimization technique is to avoid any copy memories from the GPU to the CPU. This can be achieved by sharing memory space between OpenGL and CUDA through Graphics Interoperability. This allows making zero memory copy calls since information for rendering still resides in the GPU. Nonetheless, this capability is not yet supported in the virtualization frameworks.

We can add that the computational power does not represent a vital constraint in our approach since the proposal system achieves enough computational performance to allow a few thousands of particles present in the simulation. Nevertheless, one of the most important capabilities of using virtualization frameworks such as DS-CUDA and rCUDA is the ease of multi-GPU. In this sense, the size of the system could be increased.

Considering the optimization techniques mentioned below we can denote that our approach is to use as minimal information as possible for rendering each particle. The total amount of data transferred between the server and client is on the order of KB. For example, for $n = 2744$, 66 KB are returned to the server. This amount of data does not impact severely the performance of our proposed system, even when WiFi is used as a medium of connection. However, the last factor which is the latency represents an issue in our system. This is due to

the high number of kernel calls inside of the MD simulation. We can expand this explanation with the following example: for $n = 2744$, the data transfer takes 1msec, while 400 times of kernel calls (the number inside of MD simulation) need 0.5msec $\times$ 400 = 200msec only for latency. Note that 200msec is over estimate since a series of kernel calls can hide latencies of following kernel calls. Even taken into account such effect, the reduction of kernel calls is important to leverage our proposed system.

5.2 Analysis

As we mentioned in the previous section, the latency and bandwidth are important factors to overcome in order to provide better performance to our proposal system. From previous results on Chapter 4 we could observe an impact on the performance due to the high communication generated from the kernel launch inside of MD simulation. In this section, we present the direct impact of implementing DP as a better way to understand the strategy to reduce latency. Figures 5.1 and 5.2 show the percentage of time spent on the kernel, data transfer, and latency. Kernel time refers to the actual computation of the MD simulation inside of the GPU. The data transfer takes into account memory copies from CPU to GPU. In both scenarios, DS-CUDA and rCUDA, the data transfer is only performed when MD simulation is finished, sending back only position and velocity of particles to CPU for rendering purposes as mentioned in Chapter 3. In order to calculate the percentage, we take the total time to generate one frame for each configuration of particles, $n = 1000$, and $n = 5832$. We selected this number of particles in order to compare and observe the latency effect when DP is present. Kernel percentage was computed from Eq. 4.4. Data transfer and latency were calculated using Eq. 4.1

In Figure 5.1 we can observe for 1000 particles in the system the huge part of the latency generated from several kernel calls. It is almost the same as the computing time itself, nearly 46%. As for the data transfer, we can denote only 12%. As we increase the number of particles to 5832, we can denote that the time for kernel is increased to 73%. However, the latency still presents 24% whereas the data transfer is not a constraint. This is the first insight in order to reduce latency by applying DP.

When we apply DP using rCUDA a reduction of the communication between server and client can be achieved. Figure 5.2 shows the results. Using 1000 particles we can denote a latency percentage of 14% which is significantly reduced compared to DS-CUDA is 46% for the same configuration. Utilizing DP allows wrapping the kernel calls to one single call which explains this reduction. Moreover, when we used 5832 particles the latency is reduced to 11% of the total time making the actual computation time to 88%.

Even when we are using different GPU architecture from DS-CUDA and rCUDA cases,

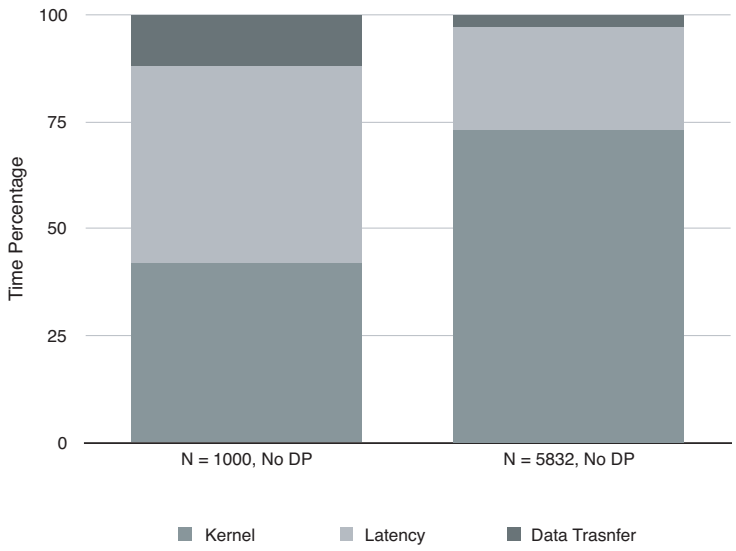

Figure 5.1: Total time percentage from kernel, data transfer and latency time of Claret using DS-CUDA. MD step is set to 100. No DP is implemented.

our strategy still consistent since the connection medium (Gigabit Ethernet) still presents the same latency. On the rest of this Chapter, we will conduct more experiments in order to show the effectiveness of our strategy when using DP for our MD simulation and visualization.

5.3 Methodology

This section provides a general overview and introduction of rCUDA virtualization middleware, including a description of how it works. We also include the motivation for using this virtualization GPU middleware. Additionally, each component of the proposed system for our test is described in detail.

5.3.1 rCUDA Virtualization Framework Overview

Different approaches for the virtualization of GPUs have been proposed [105, 106, 107, 108, 109]. The ones wrapping the original APIs share the same principle: provide the same CUDA interface to ensure the ease and re-usage of code when using GPUs in the cloud environment. A typical architecture for systems using these tools is shown in Figure 5.3. The main advantage of using virtualization frameworks is the reduction in code development time. The normal way of writing client and server applications would be to implement one application for each side, implementing a socket or some other type of communication protocol. Nonetheless, with virtualization frameworks such as rCUDA, we can simply link our CUDA GPU code on the compilation phase with the corresponding library to create a connection without developing other parts. Furthermore, we can still have the advantage of the whole development, and tools environment for the remote device in order to facilitate the

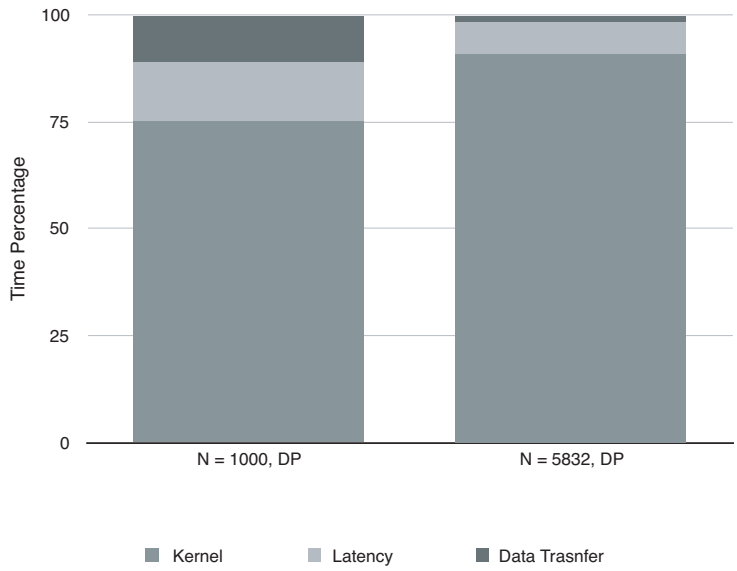

Figure 5.2: Total time percentage from kernel, data transfer and latency time of Claret using rCUDA. MD step is set to 100. DP is implemented.

access of other sensors and resources.

The virtualization framework rCUDA is a middleware that simplifies the usage of GPUs on a distributed network. One system using rCUDA is composed of a number of client nodes and server nodes. Each server node has one or more CUDA-capable GPUs that are handled by a server process. Applications on the client-side can use GPU devices to process data without having a physical GPU. The client program recognizes GPUs contained in the server nodes as if they were attached physically to the client node. The client utilizes the *nvcc* compiler to generate GPU code. Thus, during the linking process, the option flag **--cudart** *=shared* is passed to the compiler to generate the final executable. This allows loading at run-time the rCUDA dynamic library. This library implements the communication between the client and the server code. rCUDA supports communication via TCP sockets and InfiniBand verbs. For implementation details, readers are referred to [111].

From the previous Chapter, we also conducted a test between rCUDA and DS-CUDA in order to certify the better performance from the rCUDA framework. As Figure 5.4 shows, the execution of an MD simulation and visualization from rCUDA is faster against DS-CUDA. This is due to better communication implementation in the library. This lead to less latency when remote API calls are performed from the client-side. In this study, we selected the rCUDA framework due to its good maintenance and the ability to achieve better performance than other similar approaches [110], as well as the compatibility of newest CUDA 8.0 and other CUDA libraries such as cuDNN and CUBLAS. The weak point of rCUDA is that it cannot support Android since NVIDIA does not support it now.

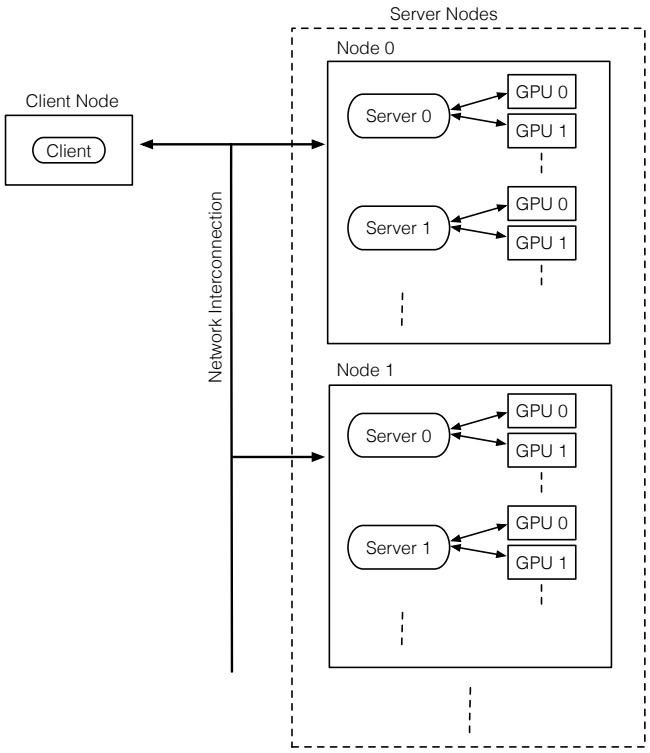

Figure 5.3: Typical architecture for virtual GPU systems.

Element	Description
CPU	Intel Core i7-6700HQ, 2.60 GHz, 8 Cores
GPU	GeForce 1070 GTX , 2048 CUDA Cores, PCIe Gen3
OS	Ubuntu 18.04 LTS x86-64
CUDA	Driver 410.48, Toolkit 8.0, SDK 8.0

Table 5.2: Server specifications. Notebook powered with NVIDIA's 1070 GTX GPU.

5.3.2 Proposed System Overview

In Figure 5.5, we show the system used to perform our tests and simulations. We used a notebook powered by a 1070 GTX GPU as a server. We choose the most recent (at the time of conducting this study) NVIDIA GPU architecture (Pascal). As for the client, we utilized Microsoft's Surface Pro 4 tablet. Full characteristics of each item are listed in Tables 5.2 and 5.3.

As the main hub, we choose Buffalo AirStation MZR-1750 for the server and client. For communication between the client and the server, two choices are available: Gigabit Ethernet through a USB 3.0 adapter and WiFi 802.11ac/n, which supports 867 Mbps over 5 GHz. For comparison purposes, we included a desktop computer powered by a 2080 RTX. The main specifications of the system are listed in Table 5.4. Section 5.5.2 includes results from a 1080 GTX GPU, another notebook with a 970M GTX, and NVIDIA SHIELD tablet powered by a Tegra K1. Full characteristics of those devices are listed in Tables 5.5, 5.6, and 5.7

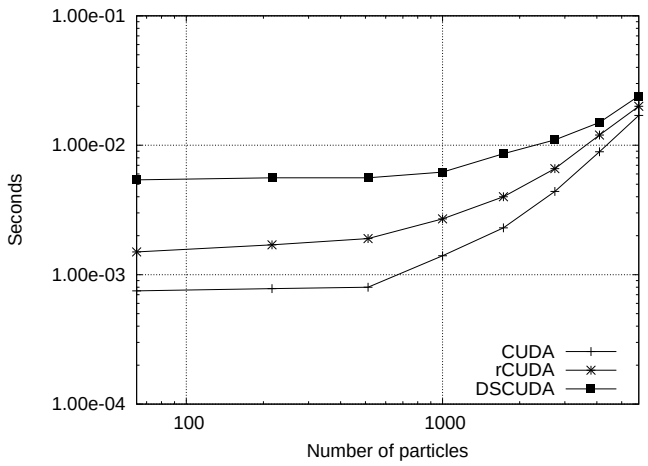

Figure 5.4: MD simulation performance between DS-CUDA and rCUDA frameworks.

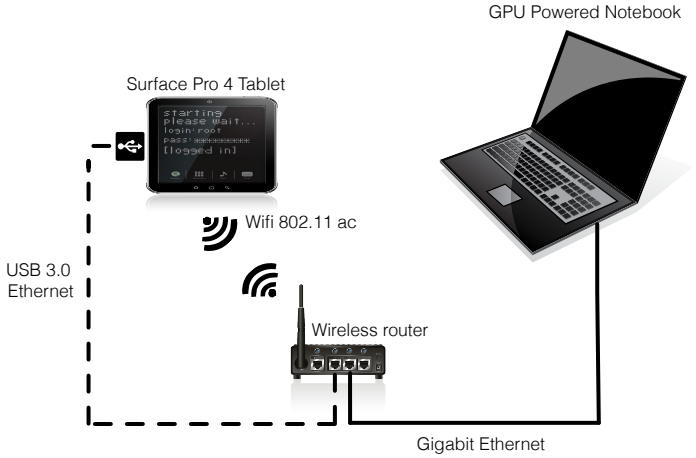

Figure 5.5: Test system.

respectively. CUDA 8.0 of the *nvcc* compiler was used to ensure compatibility with the rCUDA library.

5.4 Test Description

In this section, we show the details for each test. Two main experiments were conducted to evaluate and measure the performance of our system. We designed a bandwidth evaluation to measure the communication performance between the server and the client. Using rCUDA, some overhead from the usage of different mediums (Ethernet and WiFi) of communication exist. Furthermore, we describe details of the MD simulation, a method for collecting floating point operations per second (flops), and power measurements. These sets of experiments are useful to examine the computation performance, the impact of DP when using rCUDA, graphics rendering, and power efficiency.

Element	Description
CPU	Intel Core im3-6Y30, 0.90 GHz, 4 Cores
GPU	Intel HD graphics 515
OS	Ubuntu 18.04 LTS x86-64

Table 5.3: Client specifications. Surface Pro 4 tablet.

Element	Description
CPU	Intel Core i5-6400HQ, 2.70 GHz, 4 Cores
GPU	GeForce 2080 RTX , 2944 CUDA Cores, PCIe Gen3
OS	Ubuntu 18.04 LTS x86-64
CUDA	Driver 410.48, Toolkit 8.0, SDK 8.0

Table 5.4: Desktop powered with NVIDIA's 2080 RTX GPU.

5.4.1 Bandwidth Test

In order to measure the data transfer speed between the server and client, we used the *cudaMemcpy* function with pageable memory. Two configurations for memory copy are available: Host to Device (H2D) and Device to Host (D2H). In this experiment, the size of the data transfer increased from 1 KB to 268 MB. We also included measurements using native CUDA calls. In total, three different scenarios were considered: 1) Native CUDA, 2) Ethernet connection, and 3) WiFi connection using rCUDA.

As for the communication time within the rCUDA application, we calculated the latency which is relatively important because the GPU is connected through a network. Furthermore, we also provide kernel latency measurements for comparison purposes. This test aims to measure the time required for a kernel to be executed.

5.4.2 Molecular Dynamics Simulation and Visualization

MD simulations from a computational point of view are very intensive due to their $O(n^2)$ complexity, where n is the number of particles in the system. Another important challenge in conducting MD simulations is to achieve real-time visualization.

In this study, we implemented the algorithm shown in Figure 5.6 which describes the crystallization process of Na+ Cl- particles using a direct method. Including the Bandwidth

Element	Description
CPU	Intel Core i5-2500HQ, 3.30 GHz, 4 Cores
GPU	GeForce 1080 GTX , 2560 CUDA Cores, PCIe Gen3
OS	Ubuntu 18.04 LTS x86-64
CUDA	Driver 410.48, Toolkit 8.0, SDK 8.0

Table 5.5: Desktop powered with NVIDIA's 1080 GTX GPU.

Element	Description
CPU	Intel Core i7-4720HQ, 2.60 GHz, 8 Cores
GPU	GeForce GTX 970M, 1920 CUDA Cores, PCIe Gen3
OS	Ubuntu 18.04 LTS x86-64
CUDA	Driver 410.48, Toolkit 8.0, SDK 8.0

Table 5.6: Notebook powered with NVIDIA's 970M GTX GPU.

Element	Description
CPU	ARM cortex A15, 2.2 GHz, 4 Cores
GPU	Tegra K1, 192 CUDA Cores
OS	Android 5.0.1, ARM-32 bit
CUDA	Driver 6.0 custom, Tegra Android Development Pack 3.0r3

Table 5.7: NVIDIA's SHIELD Tablet specifications.

test, pageable memory is utilized. The behavior of a conglomeration of sodium chloride particles at the vacuum level is shown. We consider a similar GPU implementation as explained in Section 4.2.3. Furthermore, the *step* variable is changed to select the saturation GPU level for the experiments, also controls the evolution of the MD simulation, as well as the frequency of rendering. Thus, we vary this parameter to a few hundred in order to acquire the desired frame rate. Moreover, by increasing this variable we can reduce the communication overhead between the CPU and GPU. Another important technique used in GPGPU programming is Dynamic Parallelism (DP). It was first introduced on CUDA 3.5. This capability is inherently born from the need for nested parallelism for GPUs. DP allows a kernel to be invoked inside of a kernel. Nevertheless, compared with normal kernel launch, this may reduce performance due to threads from child kernels synchronization with the parent kernel. DP is suitable to implement in algorithms that compute adaptive grids, perform recursion, and split the work among different and independent threats and batches. However, in our approach, we applied DP for a different reason. We want to reduce the communication between the client(host) and server(device) through the virtualization of GPUs. Applying DP for communication reduction in our approach can be explained as follows. Normal kernel invocation case we have up to four kernel calls for each MD simulation step. If we set the number of MD steps to 100, there would be 400 kernel calls. Executing this number of kernel calls using native CUDA over PCI Express will not generate too much latency. However, using Gigabit or WiFi communication, the latency could increase severely. To implement DP in our original MD simulation we wrapped our 4 original kernels into one single parent kernel call. This allows the reduction of kernel calls from the client-side since the MD loop is situated inside of the GPU. Thus, the use of DP could reduce the communication load, as running many MD simulation steps, would only require a single kernel call from the client.

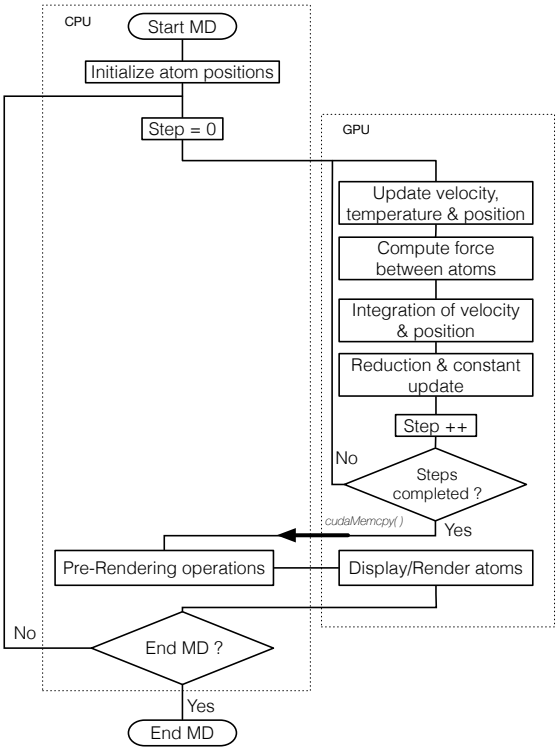

Figure 5.6: Simplified schematic algorithm of the MD simulation. The number of simulation steps before rendering can be set to a few hundred.

For visualization, we implemented OpenGL 4.2 and GLFW 3 in our MD simulation. A single dot is used to represent each atom in the simulation. Consequently, only we need position and velocity variables information from the GPU. The amount of data sent back to the CPU is in the order of KB, as we want the minimum information to visualize the MD simulation.

On the experiments, we disabled vertical synchronization (Vsync) in OpenGL to get out the actual number of frames per second inside the application. To achieved this mode, we set the variable *vblank_mode* to 0. The number of operations per second (flops) was computed using Eq. 4.4.

As for the power measurements, we used a watt meter attached to the electrical terminal of both the client and the server. However, we do not include power measurement from the access point. This can be explained as follows: our system is proposed implying the fact of the usage of mobile devices and the internet/network seamlessly. We want to show the performance of the server and client without restraining the type of network used. Additionally, we configured the NVIDIA *PowerMizer Settings* on the test machines to *Prefer Maximum Performance*. In the normal mode, this tends to reduce GPU performance to save power, especially on notebook equipment.

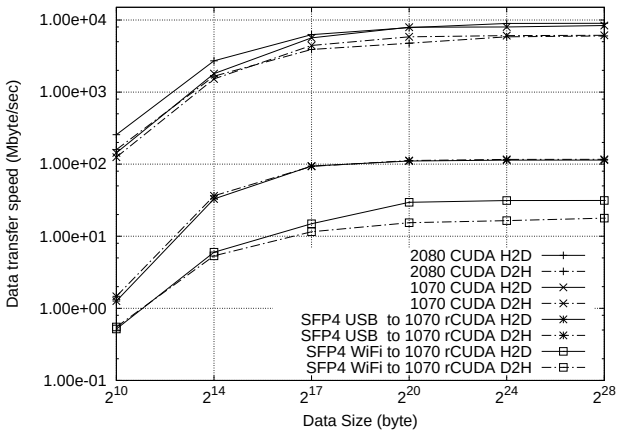

Figure 5.7: Data transfer speed using CUDA's *cudaMemcpy* function over different types of connection. H2D: Host to Device; D2H: Device to Host. Pageable memory is used.

5.5 Performance Results

In this section, we report the outcome obtained from our tests to evaluate CUDA offloading. First, we report the communication performance and latency measurements from the different mediums communicating with a GPU. Second, the raw computation power and rendering performance results are presented, as well as the impact of using DP on the system. Third, we show electric power metrics from three sets of MD simulations: computation performance vs frame rate, power efficiency vs frame rate, and power efficiency utilizing multiple client configurations.

5.5.1 Bandwidth Performance

On Figure 5.7, we show the data transfer speed between the host and device achieved by *cudaMemcpy* memory function. We calculated *latency* using Eq. 4.2.

The desktop GPU 2080 using native CUDA uses PCI Express Gen 3x16 as a communication medium. Maximum speed of 9.0 Gbytes/s was achieved for H2D, and 6.0 Gbytes/s in D2H mode.

Utilizing the notebook using the 1070 GPU, the medium of communication is again PCI Express Gen 3x16. A top speed of 8.3 Gbytes/s for H2D and 6.1 Gbytes/s for D2H was achieved. As we expected, the 2080 presents slightly better performance than the 1070 for H2D data bandwidth. However, their performance of D2H is similar.

The third and fourth cases are using the Surface tablet using rCUDA's function to transfer data through the network. Using Gigabit Ethernet, the Surface tablet reached 114.8 Mbytes/s for H2D and 116.9 Mbytes/s for D2H. With WiFi 802.11ac as the communication medium, the top speeds were 31.4 Mbytes/s for H2D and 17.8 Mbytes/s for D2H.

Latency results for copy memory function and kernel launch are shown in Table 5.8. In

	H2D latency (s)	**D2H latency (s)**	**Kernel latency (s)**
RTX 2080	3.9×10^{-6}	6.3×10^{-6}	2.8×10^{-6}
GTX 1070	7.0×10^{-6}	8.1×10^{-6}	2.6×10^{-6}
SFP4 USB to GTX 1070	8.0×10^{-4}	6.9×10^{-4}	5.2×10^{-4}
SFP4 WiFi to GTX 1070	2.0×10^{-3}	1.8×10^{-3}	1.1×10^{-3}

Table 5.8: Memory copy and kernel latency.

terms of communication performance, using native CUDA inherently achieves higher transfer speeds and lower latency for both cases: the notebook and desktop. The latency between memory copy and kernel functions are in the order of microseconds. Utilizing rCUDA through Ethernet and WiFi incur a drop in transfer speed. The latency using rCUDA with Ethernet is at least a hundred times greater than native CUDA, and a thousand times larger in case of using rCUDA with WiFi. As mentioned in the previous section, our MD simulation performs more kernel calls compared with *cudaMemcpy* before rendering a frame. This points that reducing the number of kernel calls is the most important factor in attaining high performance.

5.5.2 MD Simulation and Visualization Performance

Two main aspects of the MD simulations were examined: raw computation and power-related performance. To investigate the raw computation power, we explored the impact of using DP through rCUDA to reduce communication between the client and the server. As well, we report the number of flops and frames per second obtained on each configuration test. To evaluate electric power-related performance, we compared the power consumption against the computational power. We also included multiple client configurations to search for the best arrangement for power efficiency.

Computational Performance and Frame Rate

On the following set of tests, we set the number of particles in the simulation to $n = \{64, 216, 512, 1000, 1728, 2744, 4096, 5832\}$. The number of simulation steps is switched between 100 and 500. This was fixed to observe the DP effect on communication of the GPU during kernel calls. As is shown in Figure 5.6, step variable controls the MD loop. For comparison purposes, the MD simulations were also performed using native CUDA. The set of tests were conducted both with and without DP. Thus, for each GPU combination, the following combinations were tested:

- Steps = 100, No DP

- Steps = 500, No DP
- Steps = 100, DP
- Steps = 500, DP

First, we present the number of flops. We measured the performance of each MD simulation using the *cudaEventElapsedTime* function. The rendering phase was omitted from this test, and the copy memory functions were discarded as well. Only GPU time is measured.

The results corresponding to the four test combinations are shown in Figure 5.8. The 2080 RTX GPU achieved a top speed of 9,280 Gflops and 8,975 Gflops for 500 and 100 steps, respectively, without DP. Implementing DP, the maximum performance was 8,470 Gflops (500 steps) and 8,170 Gflops (100 steps).

On the 1070 GTX GPU case, the maximum speed achieved was 4,415 Gflops (500 steps) and 4,353 Gflops (100 steps) without DP. Using the DP, decreased to 4,338 Gflops (500 steps) and 4,254 Gflops (100 steps).

If we compared both cases, the normal kernel launch (No DP) throws similar results for a small number of particles. This is rather expected since the computing load of the GPU is not saturated. Nevertheless, for more than 1728 particles, the 2080 RTX overcomes the 1070 GTX, delivering more performance due to more computing CUDA cores inside of this architecture. It is well known that using DP will cause a slight difference in performance because of kernel synchronization. However, the performance of the newer Turing architecture used on the 2080 GPU seems to be worse than that of the 1070 Pascal GPU architecture when the number of particles is less than 1728.

In the case where rCUDA is used, the best Gflops peak is obtained with Ethernet as the communication medium. This combination achieved speeds of 4,330 Gflops (500 steps) and 4,320 Gflops (100 steps) in the case without DP. Implementing DP reduced the maximum performance to 4,132 Gflops (500 steps) and 4,099 Gflops (100 steps). Using WiFi, similar results are reached: 4,290 Gflops and 4,280 Gflops without DP, 4,119 Gflops and 4,075 Gflops with DP. In both cases, DP and No DP, we can denote that for a large number of particles, the client achieved similar performance as the server GPU. For a small number of particles, the latency becomes a factor, especially when DP is not used. Using the subframes in Figure 5.8, we can clarify the difference between using DP or not on A) and B). For the same number of steps, the performance for Ethernet and WiFi is increased since only one kernel call is executed from the client-side. Furthermore, when we increase the number of steps to 500 in subframe D), we can denote that the execution in the client is similar to the server-side. Nonetheless, this has a big impact on the frames per second since the execution time in the GPU is increased, which will be shown in the next Figure.

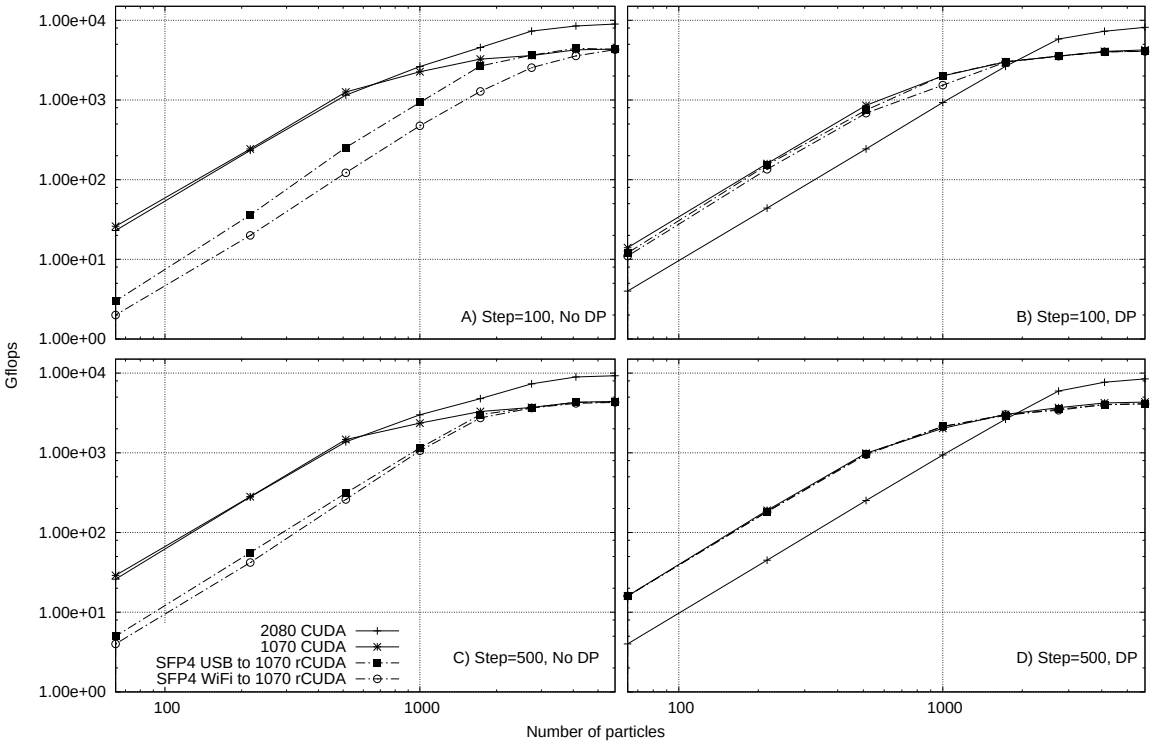

Figure 5.8: MD simulation performance. Results of computing the force between particles is shown every 100 and 500 steps. Configurations include using and excluding DP. Performance is presented in Gflops.

Following the results section on power performance, we show the results that concern frames per second (fps). The main difference between this and the previous one is the inclusion of all the time required to render the MD simulation. In this case memory copy and rendering operations are included.

As we can see in Figure 5.9, the results for various configurations are shown. The 2080 GPU system reached 7 fps and 33 fps for 500 and 100 steps, respectively, without DP for the largest number of particles. Implementing DP, similar results of 7 fps and 31 fps were achieved. The 1070 GPU rendered 4 fps (500 steps) and 17 fps (100 steps) without DP and 4 fps and 16 fps with DP. Although this test also includes copy memory and rendering operations, we can denote similar behavior with previous test. Using native CUDA without DP for a small number of particles we can reach a higher frame rate ∼600 fps for 100 steps. Whereas, using DP the frame rate is decreased to ∼400 fps.

Implementing rCUDA with Ethernet, the visualization speed reached 4 fps (500 steps) and 15 fps (100 steps) without DP, compared with 3 fps and 14 fps when DP was applied. Changing the communication medium to WiFi, we obtained a maximum of 3 fps (500 steps) and 14 fps (100 steps) without DP and 3 fps and 13 fps with DP. With a small number of particles, the communication medium has a direct impact on rendering performance. Nevertheless, in the presence of DP, we obtained better frame rates with both Ethernet and WiFi

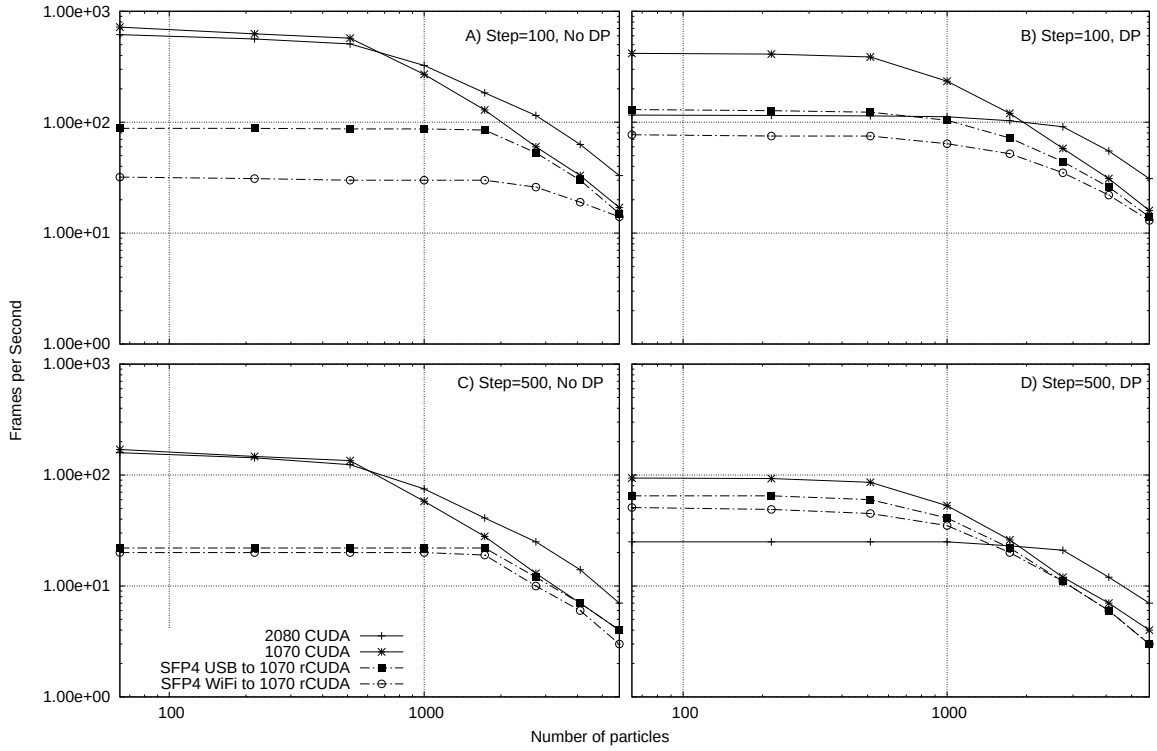

Figure 5.9: MD simulation and visualization performance. The rendering speed of our experiment is shown.

for less than 2744 particles in the system. We can clarify this as follows: in the presence of 1,000 particles in the simulation for 100 steps, using DP and WiFi, a frame rate of 64 fps is reached. Whereas, without DP, a 30 fps are reached. Using Ethernet on the same configuration, 104 fps are reached on DP, and 87 fps without it. The frame rate in the MD simulation is directly related to the number of particles in the system and the number of steps. As we increase n, the time for computing is also higher. When using native CUDA, No DP and 100 steps are the best choice to achieved a high frame rate. However, in the case of using a remote GPU, the client reaches more frame rate when DP is used for the same number of steps.

Computation Performance vs Frame Rate

Here, we show the relation between computational power and the rendering performance. Different from previous configurations, we set the number of particles in the simulation to $n = 2744$. This was selected since this is typically the order at which the computational power becomes a factor. From the point of MD simulation of NaCl, this number of particles has a practical benefit. For example, when we observe the melting phase of a crystal the temperature differs depending on the number of particles. When $n < 2000$ the temperature is from 1,040K to 1,070K. With $n = 2744$ the temperature is 1,080K, which is close to the

melting temperature of 1,081K for NaCl.

We can see this in Figures 5.8 and 5.9. This is observable when the remote execution of the GPU is closer to the native one. As well, we changed the number of simulation steps to 250 and 100. Previous test, setting step to 500 saturates the GPU performing more Gflops but lower frame rate ∼4 fps from the client-side.

Another desktop GPU (1080 GTX) and notebook GPU (970M GTX) are included for comparison. Figure 5.10 shows the complete results of this test. We also included a reference for the client side (similar small point) to the number of Gflops computed excluding communication time. The 2080 GPU achieves better frame rates and computation performance in any of the four cases. The GeForce 1080 achieved 4,736 Gflops when 250 steps and No DP. Using DP 4,400 Gflops are achieved. The amount of Gflops using DP in this architecture is decreased as expected.

On the 1070 case as a server, we can observe from the client side that using No DP with 100 steps provides a high frame rate, 36 fps on WiFi, and 45 fps with Ethernet. However, the Gflops peak from the server-side is not close enough. Contrastingly, using DP always reduces the performance distance between client and server. More precisely, in the 250 step DP configuration, we can observe through our reference points, the Gflops performance is almost similar to the server attaining 19 fps using WiFi, and 20 using Ethernet.

Using the 970 delivers 1,488 and 1,471 Gflops for No DP and DP respectively. Utilizing this GPU as the server provides a closer peak performance from the client-side. This is due to the configuration for $n = 2744$ almost reaches the top computational performance of 970. However, implementing DP with 250 steps, the client using Ethernet is executed faster than the server itself. This effect is rather well documented by the rCUDA authors [109, 114]. The main reason for this behavior is that the algorithm used for synchronization points and finalizing tasks on rCUDA is faster than the one provided for native CUDA.

CPU implementation using OpenMP is included as well. This achieved performance of 3.56 Gflops and 0.060 fps.

Power Efficiency vs Frame Rate

The results in this section present power efficiency using the configurations similar to the previous experiment. The number of particles is set to $n = 2744$ in order to make a direct comparison. As well, the number of steps is selected from 250 and 100. To compute the number of Gflops/W we consider the total amount of computing power delivered by the GPU using both (client and server) electric power consumption. The number of flops per watt is shown in Figure 5.11.

Turing architecture of the 2080 provides the best outcome in terms of performance per watt, with 26.2 Gflops/W with no DP and 100 steps which are rather expected. The GeForce

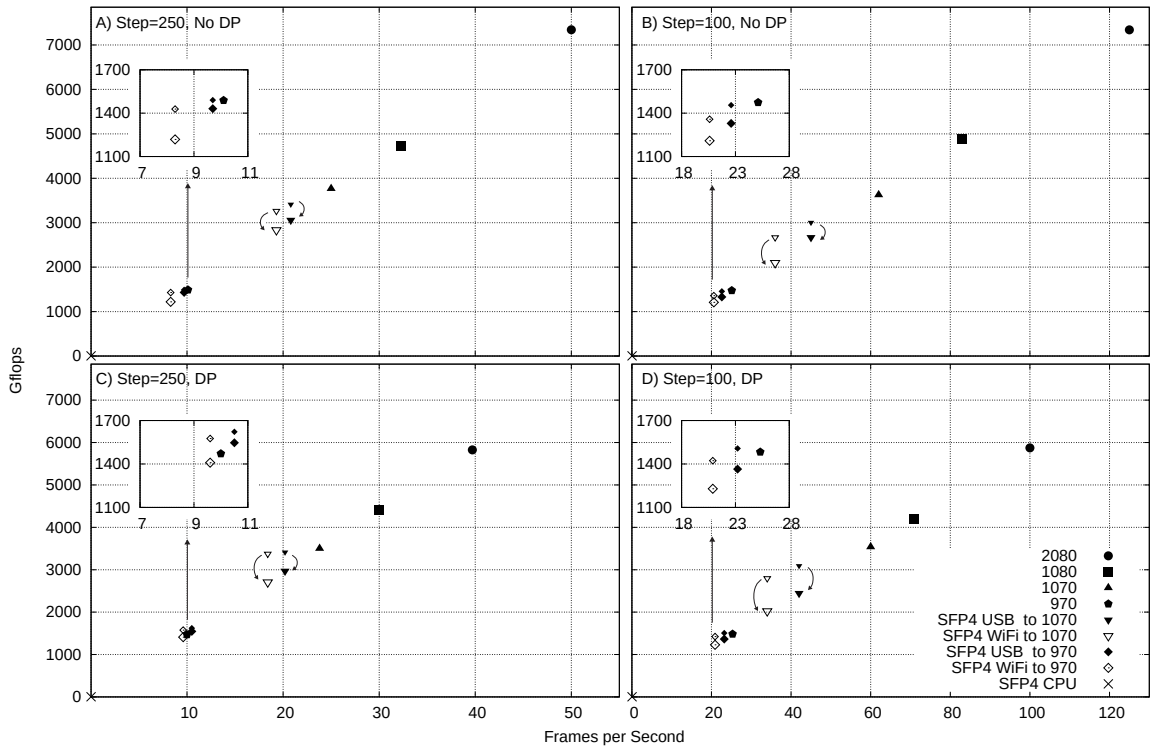

Figure 5.10: Computation performance vs frame rate. The number of particles is set to $n = 2744$. Small similar objects represents the Gflops measured with only GPU time as reference.

1080 reached only 17.9 Gflops/W compared to the 21.7 Gflops/W from the 1070 GPU for the same configuration. Desktop GPUs consume 272 W under maximum computing performance which provides a better frame rate but low power efficiency. Moreover, when the step is set to 250 and DP is used, the 1070 GPU achieves 21.3 Gflops/W compared to the 20.6 Gflops/W delivered from 2080 GPU.

In the case of 1070 as a server, we reached 15.9 and 15.5 Gflops/W using Ethernet and WiFi respectively for 100 steps and No DP. The power efficiency is higher than 15.5 and 14.3 Gflops/W when DP is used. The main reason for this is the variation in the Gflops delivered from DP and No DP from Figure 5.10 are not huge for the same amount $\sim$150 W of electrical power. Nonetheless, when we set to 250 steps, we can get 18.8 and 18.3 Gflops/W when DP is used for Ethernet and WiFi respectively compared to the 18.5 and 18.1 Gflops/W on No DP configuration. Although, using DP on 250 steps impact on fps minimally, achieves better computational performance.

The 970 reached 11.5 Gflops/W when the step is set to 250 and DP is used over WiFi. This is higher than the 11.3 Gflops/W delivered by Ethernet connection due to the faster execution and more power consumption compared to the native one.

Using the CPU implementation reached 3.5 Gflops/W and 3.6 Gflops/W for 100 and 250 steps respectively.

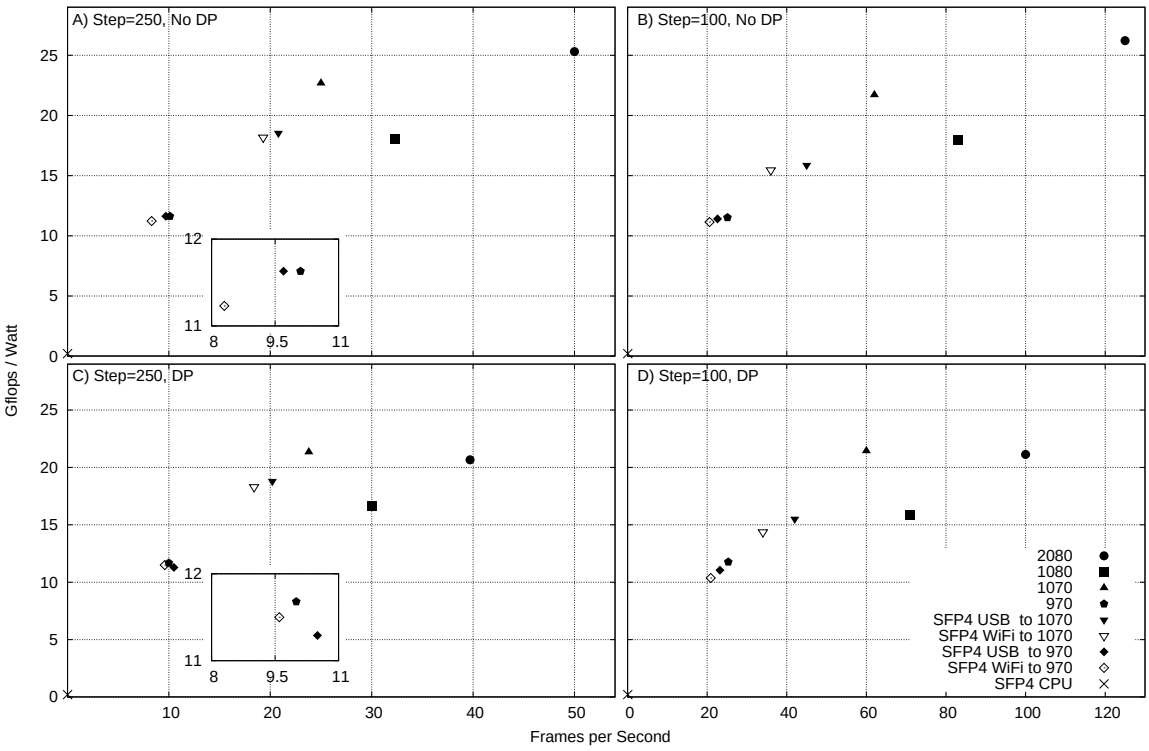

Figure 5.11: Power efficiency vs frame rate. The number of particles is set to $n = 2744$.

Power Efficiency implementing Multiple Clients

Here, we show the results using one server and multiple clients. In our previous results, we have shown that the GeForce 1070 using DP at 250 steps is the optimal server-side configuration for MD simulations and visualization. We compute the amount of Gflops/W and consider the power consumption from both client and server. As well, we varied $n = \{1000, 1728, 2744, 4096\}$ since exploring the saturation area of the GPU is needed. Table 5.9 presents the results on the following client configurations:

- Conf.1A: Only one client using Gigabit USB 3.0 Ethernet.
- Conf.1B: Only one client using WiFi 802.11ac 5 GHz.
- Conf.2A: Two clients using WiFi 802.11ac 5 GHz
- Conf.2B: One client using WiFi 802.11ac 5 GHz, another client using Gigabit USB 3.0 Ethernet.
- Conf.2C: Two clients using Gigabit USB 3.0 Ethernet.
- Conf.3A: Three clients using Gigabit USB 3.0 Ethernet.
- Conf.3B: Two clients using WiFi 802.11ac 5 GHz, another client using Gigabit USB 3.0 Ethernet.

	Steps=250, DP			
n	1000	1728	2744	4096
RTX 2080	6.2	13.6	20.7	26.5
GTX 1070	12.2	18.3	21.3	23.6
Conf.1A	8.7	14.1	18.8	23.1
Conf.1B	7.6	13.5	18.3	22.0
Conf.2A	8.3	14.8	18.5	22.8
Conf.2B	10.2	15.2	21.7	23.3
Conf.2C	9.8	14.9	18.7	21.5
Conf.3A	9.7	14.8	16.5	20.2
Conf.3B	9.6	15.5	16.2	20.0
SHIELD	6.3	7.5	8.8	8.8

Table 5.9: Power efficiency (Gflops/watt) using multiple client combinations.

Various configurations for each number of particles is presented. For more than one client, we include at least one Gigabit USB 3.0 Ethernet, as the latency and bandwidth are higher than those of WiFi. Moreover, we also examined the performance from the server-side using native CUDA and tested the SHIELD tablet from NVIDIA. This tablet is equipped with a Tegra K1 GPU and is able to handle CUDA calls through the Java Native Interface (JNI) [95, 96]. However, the results are from normal kernel calls since Tegra K1 is CUDA 3.2 architecture and is not capable of DP.

The outcome is as follows: the best power efficiency combination was achieved with the two-client configuration using Ethernet and WiFi when $n = 2744$. Table 5.10 shows details of the multiple combinations. As we can follow, this configuration of two clients distributes the resources (Gflops) from the GPU keeping a good balance of electric power usage. Nonetheless, the frame rate is significantly reduced for the WiFi client. Compared to two clients using Ethernet or WiFi, we can see a more stable frame rate from both clients. The combination of both resources can not achieve better performance per watt. A similar scenario of distributed resources on the GPU is observable when we used three client configurations.

5.6 Conclusion

In this Chapter, we were able to accelerate heavy parts of an application from a tablet using a remote low-power GPU from a notebook through the rCUDA middleware. Comparisons using GPGPU techniques such as DP to hide the kernel call latency were conducted, and different GPU architectures were examined. Our system achieved better computational performance, more frames per second, and higher performance per watt than a tablet powered by a CUDA-capable GPU and the server itself.

Only the kernels required for intensive computations are downloaded to the server-side. This technique has an advantage over frameworks such as Desktop as a Service (DaaS),

		FPS	Gflops	Power (Watt)		Gflops/Watt
				Client	Server	
RTX 2080		39.7	5827	282		20.7
GTX 1070		23.8	3499	164		21.3
Conf.1A	USB	20.2	2970	8.0	150	18.8
Conf.1B	WiFi	18.4	2706	8.0	140	18.3
Conf.2A	WiFi	11.0	1618	7.5	161	18.5
	WiFi	11.1	1637	7.5		
Conf.2B	WiFi	6.7	978	7.5	160	21.7
	USB	19.2	2813	7.5		
Conf.2C	USB	11.1	1632	7.5	161	18.7
	USB	11.3	1654	7.5		
Conf.3A	USB	7.1	1037	7.5	167	16.5
	USB	7.0	1027	7.5		
	USB	7.2	1056	7.5		
Conf.3B	WiFi	6.9	1013	7.5	166	16.2
	USB	7.1	1041	7.5		
	WiFi	6.9	1007	7.5		
SHIELD		0.5	78	8.8		8.8

Table 5.10: Detail information for Power efficiency (Gflops/watt) using multiple client combinations. The number of steps are 250, and $n = 2744$.

because the main objective of DaaS is to offload everything to a server or virtual machine, including rendering resources and I/O events. The main problem with this approach is that every user-interface event on the client has to be sent to the server in the cloud. Because of this, the network communication time may significantly affect the usability of the application. Conversely, the kernel offloading approach processes all interactive events on the client-side, so network performance is not seriously affected.

Using DP has significant meaning when offloading is performed. We show that keeping the GPU saturated with more steps helps in the reduction of latency from the client-side. However, as more steps are used, the frame rate is reduced. We found that for 250 steps, not only achieving a good frame rate is feasible for our MD simulation, but also a better power efficiency when multiple clients are used. Our approach can also be applied for many other scenarios where kernels could be wrapped using DP for offloading. Applications such as fluid dynamics, weather forecasting, and video analysis are few examples to mention where the GPU is implemented to overcome computational bottlenecks. Most of them consist of many kernel implementations that could be implemented using our approach. However, we need to assure the consistency of the data access in those different scenarios.

From the MD simulation point of view, we achieved the visualization of the crystallization phase for Na Cl particles. This was possible due to enough computational performance and frames per second delivered from our system. We can explain the outcome of the visualization

as follows: for crystallization phase, we need 3×10^5 MD steps for around $n = 2000$. By gradually decreasing the temperature, liquid Na Cl forms a crystal. This takes one minute for a user to observe when calculation speed is step = 250 and fps = 20, which we can achieve with our system for $N = 2744$. When $n > 2744$ particles the user might find it difficult to interact or observe in real-time the crystallization phase.

From a casual point of view, mobile devices are not expected to perform intensive computations and save energy at the same time. However, cloud computing or similar systems like ours are an interesting approach along the lines of simultaneously achieving more computational power and better performance per watt on mobile devices. There exist some situations in which systems such as ours can deliver positive differences for interactive systems. For instance, when the user is in a remote location and there is no sufficient internet connection to reach the cloud, a notebook powered by a GPU could execute interactive simulations. Examples include oil extraction points in the sea or when diving and the tablet must be used underwater.

6

Future Directions

We have studied GPU techniques in order to accelerate MD simulations and visualization using tablets as a medium for interaction. On Chapter 4 and 5 we proposed to offload intensive computations to a remote GPU using virtualization framework tools. Furthermore, in Chapter 5 we proposed to use Dynamic Parallelism to tackle latency between server and client. DP is a capability inside the GPU that was originally designed to allow GPUs to use recursion inside the kernels. This characteristic allows a child kernel to be invoked from a parent kernel. However, our purpose to use DP inside our MD simulation and visualization is to reduce kernel call latency. It is common for GPU applications to be constituted from more than one kernel. In our approach we needed to wrap all kernel calls inside the MD simulation code. Nevertheless, all the data that kernels use need to be inside of the GPU all the time. This may be a constraint in different applications from ours since other applications may require to sent back data to the CPU.

Even though we applied DP to reduce the communication bottleneck between the host (CPU) and device (GPU), they're still more space for improvement using the newer GPU capabilities in software and hardware. One of them is the usage of Graphics Interoperability which enables common memory space between CUDA and OpenGL/Direct3D. This allows the reduction of memory copies during the visualization process, thus speeding up the execution of the rendering. Another technique is the usage of the hardware decoder/encoder for images inside the GPU. In this Chapter, we present how we can complement our system by applying these features, as well as tackling the rendering problem in a server-client scheme.

6.1 Migrating All to GPU: Avoiding Communication Bottleneck

Since the conception of the GPU, the main bottleneck using this parallel hardware is residing on the data transferring to the CPU. Data movement between GPU and CPU is done through the PCIe bus. This communication process can have a large impact on performance, especially

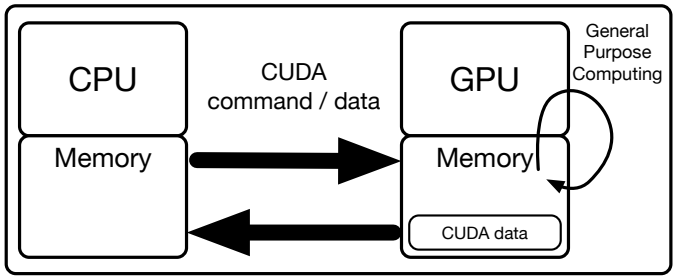

Figure 6.1: GPU scheme to perform general purpose computing using CUDA.

when we consider that the bandwidth of the PCIe is much lower than the GPU device memory bandwidth [118, 119]. It is rather well known that naive implementations in GPU code incur in delaying GPU computation until data transferrer is completed. This is an important reason for overlapping computation and communication. However, this presents a nontrivial optimization which requires a considerable effort from the developer [120, 121]. Furthermore, in our approach not only computations for the simulation are happening inside the GPU but also for rendering the visualization. Thus, communication with the CPU is necessary for setting, manipulating, and drawing.

6.1.1 Implementing Graphics Interoperability

The main concept behind this idea can be express as follows. The GPU can be used to overcome intensive computation through CUDA. Figure 6.1 shows a naive scheme using the GPU for general-purpose computations. Here, we can denote that not only communication inside the GPU memory is needed but between CPU and GPU as well. Moreover, Figure 6.2 shows the access to the GPU in order to render geometry by calling OpenGL API. Similar communication behavior is noted in this scenario as well as using the GPU with CUDA. Furthermore, a naive implementation for simulation and visualization using the GPU is shown in Figure 6.3. This scenario shows the excessive communication between CPU and GPU due to the null awareness in memory resources between CUDA and OpenGL. In our MD simulation, the positions of atoms, which are essential data for visualization, are calculated on the GPU and sent back to the CPU. However, this variable data is sent back again to the GPU. This operation is needed since OpenGL need to bind the data into its context to finalize the rendering process.

In order to reduce the overhead of redundant communications, CUDA provides a software feature called graphics interoperability which allows sharing memory resources between CUDA and the rendering context, OpenGL and Direct3D. Applying this technique, the data back and forth between CPU and GPU is not needed, alleviating the communication and providing better performance on the simulation and visualization. However, using this fea-

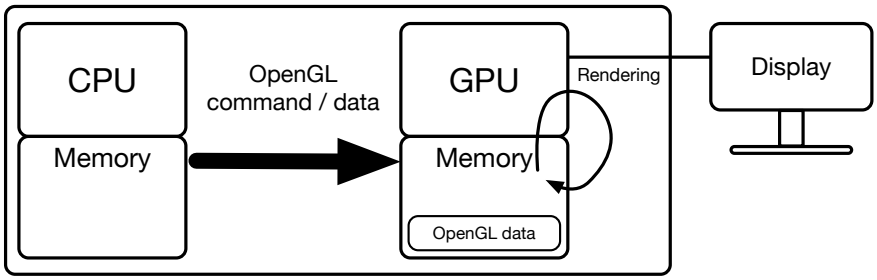

Figure 6.2: GPU scheme to perform rendering using OpenGL.

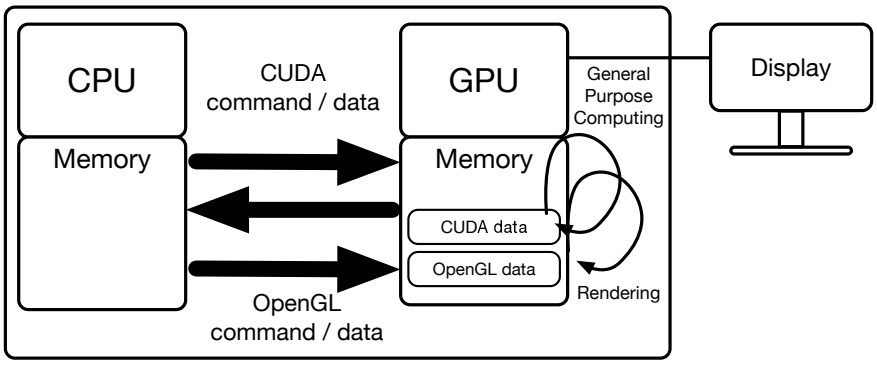

Figure 6.3: GPU scheme to perform rendering and general purpose computing. No optimization is used between OpenGL and CUDA.

ture is not straight forward: the developer needs to keep congruency between CUDA and OpenGL memory space.

On the first steps on this research, we have successfully accelerated our MD simulation and visualization using CUDA and OpenGL graphics interoperability [122]. Other proposals report better performance using this technique as well as [123, 124, 125]. Another motivation using graphics interoperability is that GPU virtualization frameworks such as rCUDA and DS-CUDA are lack of this feature. This is rather expected since graphics interoperability is

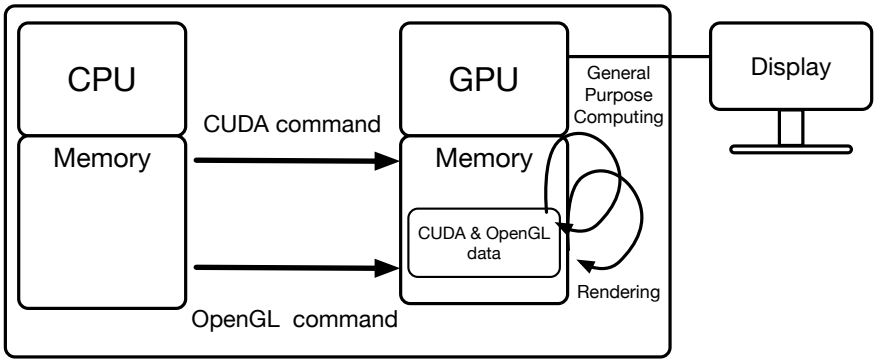

Figure 6.4: GPU scheme to perform rendering and general purpose computing. Graphics interoperability optimization is used between OpenGL and CUDA.

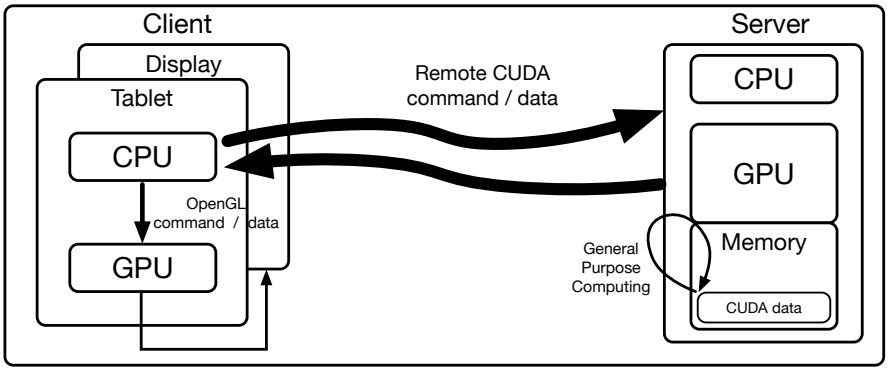

Figure 6.5: GPU virtualization for general purpose computing using CUDA.

a feature to work inside of the local rendering and computation context. More specifically, CUDA and OpenGL could not be easily mapped using a remote GPU. However, in the following section, we tackle the idea of sharing the final frame buffer between the client and the server.

6.1.2 Implementing Encode/Decoder on the GPU for Frame-Buffer Retrieval

As we mentioned in Chapter 2, since the introduction of NVIDIA's GPU Kepler architecture a hardware-based video encoder and decoder acceleration called *NVENC* and *NVDEC* were included on the GPU [126]. This is an independent and fully dedicated hardware which does not use the graphics engine on the GPU. This presents an advantage since the GPU and CPU are free to perform other operations. According to previous studies [127, 128], utilizing this video codec engine on the GPU saves time: both process, rendering and encoding happen in the GPU memory space. It also reduces the size of the transfer in a server-client scheme. For example, a 1920×1080 on 24 bit using RGB format is about 6 MBytes on data size. Using, H.264 on the GPU encoder engine can drastically compact the data to 13 KBytes. Thus, implementing this feature shortens the transferring time in a server-client scheme, thus enabling a better interaction. Using our approach for MD simulation and visualization using remote tablets, we can implement this GPU hardware feature as follows: we propose to bind a remote frame buffer on the server-side. This final outcome will be broadcasted to the client-side. The frame buffer is chosen to be shared since it contains all the final render pixel information processed through the OpenGL pipeline. Bringing this buffer information only from the server-side will lead to the alleviation of the heavy graphics process for non-high-end GPU clients.

A basic scheme for this implementation is as follows. In Figure 6.5 we can denote the usage of a remote GPU for offloading intensive computations from a client-side. Figure 6.6 shows a

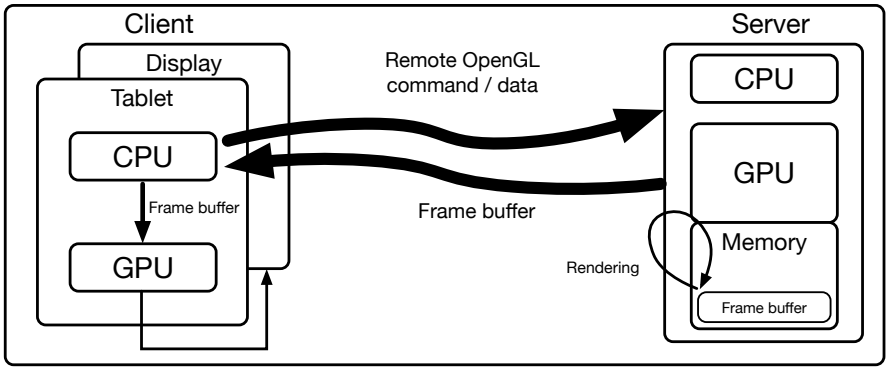

Figure 6.6: GPU virtualization for remote rendering using OpenGL.

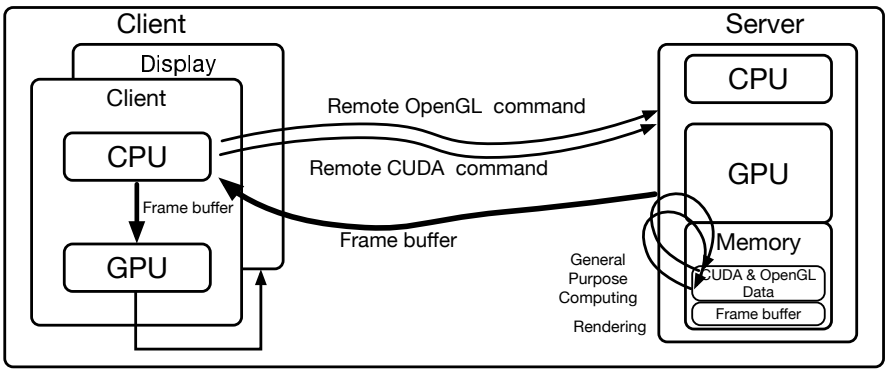

Figure 6.7: Full GPU virtualization for remote rendering and general purpose computing. CUDA and OpenGL are used.

scenario where the OpenGL frame buffer is rendered on a server-side and it is sent back to the client-side. A naive implementation using both schemes will create a huge communication bottleneck between server and client. Consider as well that the connection speed between client and server can be slow e.g. WiFi.

To this aim, CUDA and OpenGL graphics interoperability must be implemented on the server-side. Figure 6.7 highlights the communication reduction, since only the frame buffer is back to the client-side. Furthermore, the frame buffer can be shared using compression techniques such as H.264 which are natively supported by GPUs. This will allow reducing, even more, the transfer size; thus more speed performance is expected on the application running on the client-side.

6.1.3 EdRender: First Approach to Graphics Interoperability on GPU Virtualization Frameworks

In order to proceed the reduction of bottleneck communication between GPU and CPU in the server-client scheme, we propose to implement Graphics Interoperability utilizing the Encoder and Decoder on hardware capabilities in our MD simulation and visualization using

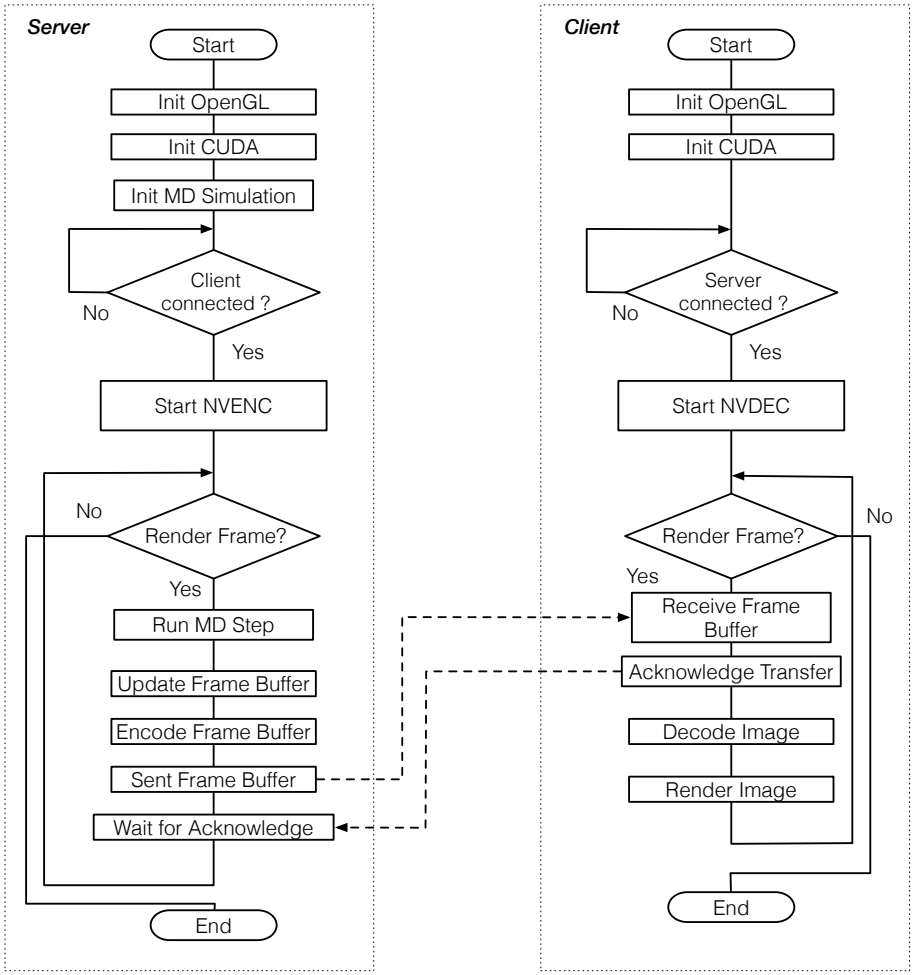

Figure 6.8: EdRender process flow. Server and Client implementations are presented.

tablets. As we know, rCUDA [113] and DS-CUDA [108] GPU virtualization frameworks do not support this feature.

Figure 6.8 shows a simplified schematic chart flow of the main algorithm for our *EdRender* framework. We implemented both, server and client-side, considering that both sides are powered by NVIDIA GPUs. This is due to the Encoder and Decoder capability that was described in the previous section. Despite the fact that there are not so many tablets powered by NVIDIA GPU, we take the first step in this direction by using a Laptop machine powered by NVIDIA GPU as a client. A server machine is using a High-End GPU Quadro model. Our approach later could be applied to a real-case scenario using a mobile device since most of these devices are equipped with a video decoder.

We can expand the explanation of the implementation of the server-side as follows. First, we start the application by setting up the OpenGL context. This process includes the shaders, the auxiliary library for window handling, and buffer registration for rendering to a custom frame buffer. Next, we initialize the CUDA context, setting up the device and providing access to OpenGL memory space through Graphics Interoperability. Following, the MD simulation

is initialized, including variables for atoms, position, velocity and other main variables. Before the next step, we wait for the connection from the client. The communication is performed via socket implementations using the TCP protocol. Once the client has been connected to the server, the encoder engine is set up. After this process the main loop starts. Here, we can set up the amount of number of steps for the MD simulation. Once the steps are completed, we render the atoms using OpenGL. However, we do not render to a normal output: since we want to encode the final output or image, a special frame buffer is prepared at the beginning. Thus, the rendering is re-directed to a texture map which is used to be encoded by the NVENC engine on the GPU. After the encoding is done, the data finally is returned to the CPU and be transmitted using sockets to the client-side. Our naive implementation waits from an acknowledge answer from the client-side before another frame is processed.

On the client-side, we can denote the following. When the application starts, we initialize and set up the OpenGL context. This includes, as well as the server-side, variables for shaders, the auxiliary library for windows handling and the texture which will be used for rendering the decoded frame from the server. The next step is the initialization of CUDA context, as well we set up the device to be able to perform Graphics Interoperability with OpenGL. Here, we used this feature in order to handle the decoded image inside the GPU memory. Without this technique enabled, we would have to send back this data to the CPU. Following is the connection to the server-side by sockets. Once the connection is performed, the decoder engine is set up and we enter the main loop of the client side. Here, the first procedure is to wait for a frame buffer from the server-side. Once the broadcasting is successful, we respond with an acknowledge message. In the next step, the decoder of the frame buffer rendered by the server is processed. It is important to mention that NVDEC provides APIs for handling and parsing the image to decode. However, the implementation of the parser included in this API might be not the optimal [126]. After the decoded image is ready, we can use this inside OpenGL due to the implementation of the Graphics Interoperability feature. The last step is to render the final image to a texture in the proper resolution on the client-side.

6.1.4 EdRender - Preliminary Results

In order to test our idea for implementing Graphics Interoperability for remote devices, we prepared two machines powered by NVIDIA's GPU. Table 6.1 shows the full specifications of the server equipment. Note that here we are using Quadro High-End GPU. The main reason to use this GPU for the experiment is only due to the API for handling the encoder. Compared to the commodity GPU models or GeForce GTX, Quadro GPUs offer a more handy API for managing the encoder/decoder called NvIFRO [129]. Table 6.2 shows the full characteristics of the client machine. We choose a low powered GPU which we used for previous experiments on offloading kernel using DP mechanism. In this case, we use the

Element	Description
CPU	Intel Core i5-2500HQ, 3.30 GHz, 4 Cores
GPU	Quadro K5200 , 2304 CUDA Cores, PCIe Gen3
OS	Ubuntu 16.04 LTS x86-64
CUDA	Driver 390.48, Toolkit 7.0, SDK 7.0

Table 6.1: Server specifications. Desktop powered with NVIDIA's Quadro K5200 GPU.

Element	Description
CPU	Intel Core i7-6700HQ, 2.60 GHz, 8 Cores
GPU	GeForce 1070 GTX , 2048 CUDA Cores, PCIe Gen3
OS	Ubuntu 16.04 LTS x86-64
CUDA	Driver 390.48, Toolkit 7.0, SDK 7.0

Table 6.2: Client specifications. Notebook powered with NVIDIA's 1070 GTX GPU.

normal API for NVDEC. We also used the image parser for the decoder which is provided by the video codec SDK from NVIDIA.

For comparison purposes, we selected another two different proposals to offload OpenGL graphics in a server-client scheme. The first one is VirtualGL [130, 131] Linux-toolkit which is an open-source solution for graphics acceleration in a remote display. This framework allows the OpenGL commands and 3D data to be re-directed to a server machine where the actual rendering is processed. After the final image is rendered, VirtualGL sends back the result over the network to the virtual display. The second proposal for comparison is X11[132, 133] forwarding or indirect rendering on X11 window system machines. Since the core of the implementation for X11 is a server-client scheme, we can use a remote display in order to process all the OpenGL remotely. This can be set up via *ssh* tunneling. We also included test using the GPU virtualization frameworks DS-CUDA and rCUDA using DP mechanism for kernel offloading.

Gigabit Ethernet connection is used between client and server. CUDA-MemCPY and CUDA-Interop are executed in the server machine locally for comparison purposes. CUDA-MemCPY returns data to the CPU in order to execute rendering. CUDA-Interop uses the technique explained in Section 6.1.1 for memory copy avoidance between CPU and GPU. The resolution for the final screen is set to 1280×720 which is standard 720p. The MD simulation step is set to 10.

Figure 6.9 shows the results of our MD simulation and visualization using various graphics offloading techniques. The time taking to render one frame is presented in seconds. The faster execution is performed by using CUDA locally and Graphics Interoperability as expected. For a low amount of particles, we can denote the difference between both executions. Graphics Interoperability alleviates the communication bottleneck, making a faster execu-

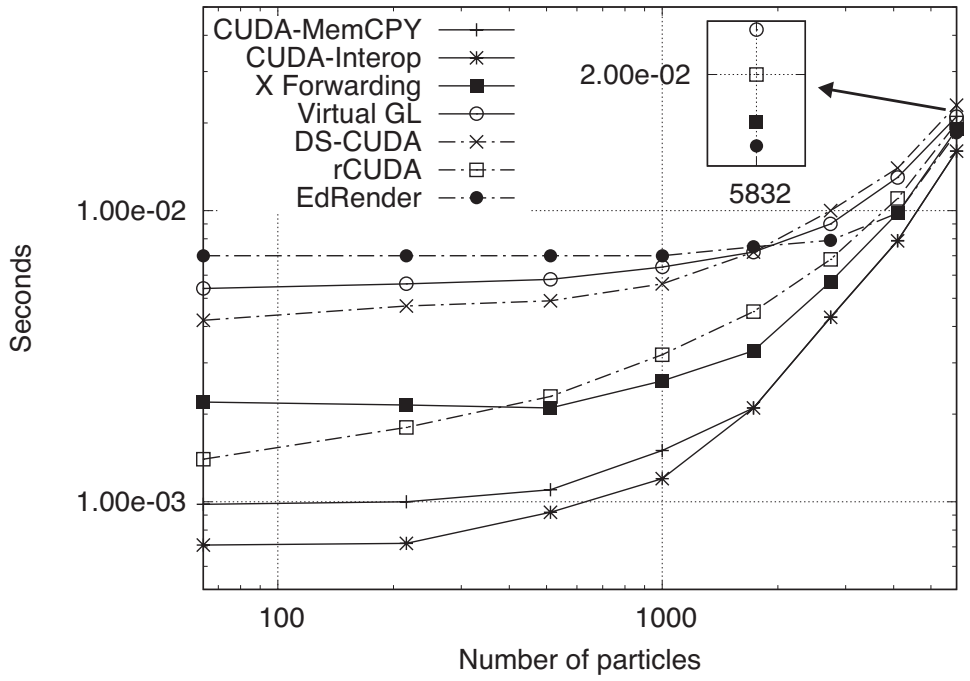

Figure 6.9: MD simulation and visualization using graphics offloading. Rendering speed is presented in seconds. CUDA-MemCPY and CUDA-Interop refers to local execution.

tion. However, for more than 2744 particles the performance of both approaches are similar. In this case, the computation of the simulation takes longer than other processes, saturating the GPU in order to compute force, position, and velocity for the particles. Using rCUDA with DP for kernel wrapping shows close performance for a low amount of particles compared to local execution, as shown in Chapter 5. Implementing X11 forwarding provides us the best range of execution for several hundred particles in the system in a server-client scheme. This may be due to high-tuned latency in X11 implementation. Nevertheless, a more detailed analysis would confirm this assumption. DS-CUDA implementation is the worst for a large number of particles. However, it overperformed our naive implementation and VirtualGL for a low amount of particles. In the case of VirutalGL, we suspected a more efficient execution against X11, since they claim its communication library is better implemented. However, in our test, the performance is rather poor. More detailed analysis is necessary as well to confirm our assumption. Lastly, we have our proposal EdRender which presents a bad performance for a low number of particles. This is due to the naive implementation of our communication socket library. In Figure 6.9, we used knowledge in order to process the data request from both sides, server, and client. This could be alleviated by using buffering techniques. Nevertheless, for a large number of particles, we can denote a faster execution than X11 and rCUDA. Our implementation still has a big room for improvement.

6.2 Conclusion

We have proposed to further alleviate the communication bottleneck between CPU and GPU for MD simulations and visualizations in a server-client scheme. Our results from Chapter 4 and 5 offloading kernel and using DP as a wrapping mechanism show that it is feasible to use a tablet to execute interactive MD simulation in real-time. In this Chapter, we proposed to improve the execution performance of our MD simulation using tablets. The first steps were to apply software capabilities that reduce communication overhead between the rendering and computation process inside the GPU. As well, we took advantage of the hardware capabilities of the encoder/decoder for sharing the frame buffer. This proposal could be seen as implementing graphics capabilities for GPU virtualization frameworks. From the nature of a server and client system, these features are rather known to be difficult to implement, especially sharing render resources. However, such efforts are considered in X11 forwarding and VirtualGL. We implemented a naive framework called EdRender which uses CUDA capabilities for computation and also sharing Graphics Interoperability by sharing the frame buffer through the network. Our results showed a poor performance of our proposal. Nevertheless, by customizing the communication routines further we can expect better results.

Techniques similar to our proposal are becoming real applications for gaming on streaming. They include the cases of Stadia from Google [134], PlayStation Now from Sony [135], and Xbox Game Streaming form Microsoft [136]. This approach applies a very robust GPU on the server-side in order to render the game at maximum detail. After rendering is performed the frame buffer is encoded for streaming. On the other side, there is a medium-power client terminal (tablet or smartphone) which decodes the final image. However, the input for the game comes from the client-side. This means that latency becomes a very important factor in this new approach in order to get a good experience.

7

Concluding Remarks

The co-processor called GPU was originally designed to support the CPU in the acceleration of image rendering. The rapid development of these graphics chips due to the popularity of games and media helped the GPU industry to evolve its ubiquitous parallel architecture. Nowadays, supercomputers are powered by GPUs performing heavy and large computations. This trend of using GPUs for general-purpose computing had become a natural way to accelerate applications in the HPC field such as MD simulations, Deep Learning, Networking topology, etc. Scientific computer simulations of physical phenomena are usually executed without visualization. After the simulation is performed, the results are analyzed and visualized using another special computer entity. To overcome this split scheme, the early stages of this research has been focused on the ability to perform real-time high-performance MD simulation and visualization using GPUs. Furthermore, post PC devices such as tablets have proved to be a path to redefine the way users interact with computers and visualize data, especially when interactive manipulation and simulation have become a new trend for analyzing a large amount of data on the fly. However, the computing power of touch devices is still not enough for such simulations. In this dissertation, we proposed the exploration of using GPU virtualization frameworks with tablets in order to achieve real-time MD simulation and visualization. GPU virtualization frameworks can complement the low computing power, handling GPUs remotely in order to perform heavy simulations on mobile devices.

In Chapter 4 we proposed to offload intensive computations from a tablet performing an MD simulation and visualization through DS-CUDA virtualization framework. We used a low-powered GPU from a notebook in order to keep the power efficiency of the whole system. We used the DS-CUDA framework to enable the development of remote offloading using mobile devices. Only CUDA kernels were offloaded due to the ability of DS-CUDA preprocessor to wrap seamlessly CUDA code without modification. Speed up of Gflops were obtained when the MD was compared between GPU and CPU implementation. However, a trade of less amount on frames/sec were noted when a large amount of Gflops was attained. It

was found that saturating the GPU the communication overhead could be hidden between the tablet and the GPU. However, this is not the optimal way to achieve real-time visualization of MD simulations.

In Chapter 5 we applied Dynamic Parallelism as a novel idea to tackle communication reduction in the execution of real-time MD simulation and visualization using tablets. We used the rCUDA virtualization framework instead of DS-CUDA. The main reason includes that rCUDA is more up to date and presents better kernel latency compared against DS-CUDA. We implemented DP in order to hide kernel call latency in our MD simulation and visualization. This technique allows our system to achieve better computational performance, more frames per second than a tablet powered by a CUDA capable GPU. As well, we found that keeping the GPU saturated with more steps in the MD simulation helped in the reduction of the latency from the client-side. However, using more steps affects the frame rate of the visualization. We found that 250 steps were optimal for our system achieving enough frame rate and better power efficiency when multiple clients were used.

Lastly, in Chapter 6 we made the first steps in order to further alleviate the congestion in the communication between client and server for MD simulations and visualizations. Implementing graphics capabilities for GPU virtualization frameworks are rather known to be difficult, especially sharing rendering resources. This is due to the nature of a server and client scheme. First thoughts to reduce communication overhead between the rendering and computation process inside the GPU were to apply software capabilities such as Graphics Interoperability and take advantage of the hardware capabilities of encoder/decoder. This will allow putting all together inside the GPU, in order to perform both simulation and visualization. We implemented a naive framework that uses such capabilities, sharing the frame buffer through the network. Our preliminary results demonstrated a poor performance from our proposal. However, by customizing the communication routines further, we can expect better results.

Our initial aim was to be able to hook up a tablet from a supercomputer in order to achieve real-time simulation and visualization. Through this dissertation, we discussed the main problems in order to use the main hardware accelerator in the supercomputer which is the GPU. We proposed a system capable of MD simulation and visualization in real-time using a tablet. We realized that the actual frameworks for using remote GPUs are not ready for such a task. Reducing the communication between server and client is a key factor in order to achieve such kind of simulations. We paved the path to complement these GPU remote frameworks, including a technique using DP for better performance and also sharing frame buffers techniques for a complete offload to a GPU. This dissertation walk trough this topic using a small server and client scenario in order to analyze the basic problems and bottlenecks. This will aid to achieve the use of more sophisticated and robust servers in the

future.

Complementing the real-time simulation and visualization, another important topic inside this dissertation is the interactivity that handheld devices provide whit so many sensors. Our system proposal aims the offloading of only kernel parts (computationally intensive) to the GPU. Using this approach allows the developer to maintain control and access to all development ecosystem on the tablet device. As well as to keep the asynchronous execution of the application: on this scenario meanwhile the intensive routines are performed in the remote GPU, we still have computational resources on the tablet to perform other actions. This allows access with minimum latency to other sensors in order to react and provide feedback to the simulation. As we mentioned in Chapter 3, we can take advantage of this feedback in order to interact and alter the simulation and visualization. Our approach in the current MD simulation is rather simple, only modifying certain values and the possibility to visualize different angles of the simulation. However, using other interactive sensors we can provide a new level of interactivity to the simulation. For example, we can use haptic sensors to provide real-time force feedback. The ability to modify the crystal structure using 3D hand recognition is another example. Moreover, utilizing VR glasses in order to provide more depth and realism to the visualization.

A huge room for improvement is expected since the evolution of the GPU will continue to boost by the incoming services for gaming on the cloud. These new coming technologies and services will leverage new features such as real-time ray tracing rendering for photo-realistic images. Furthermore, the server-client scheme will become also more common in the incoming years.

List of contributions

Related to this dissertation

Journals

1. **Martinez-Noriega Edgar Josafat**, Syunji Yazaki, and Tetsu Narumi, "CUDA Offloading for Energy-Efficient and High-Frame-Rate Simulations using Tablets", *Concurrency and Computation: Practice and Experience*, e5488, August 2019. (The contents of Chapter 5)

International conferences

1. **Martinez-Noriega Edgar Josafat**, and Tetsu Narumi, "Remote Graphics Rendering for MD simulation using NVIDIA's Pascal Architecture", *2017 International Summer School on HPC Challenges in Computational Sciences*, USA, Co, Boulder, June 2017. (The contents of Chapter 6)

2. **Martinez-Noriega Edgar Josafat**, and Tetsu Narumi, "High Performance Computing on Mobile Devices through Distributed-Shared CUDA", *GPU Technology Conference (GTC)*, USA, San Jose CA, S5290, March 2015. (The contents of Chapter 4)

Domestic conferences

1. **Martinez-Noriega Edgar Josafat**, and Tetsu Narumi, "MD simulation and visualization for low powered devices offloading CUDA code", *RIKEN AICS HPC Youth Work-Shop*, Kobe, Japan, November 2016. (The contents of Chapter 5)

2. **Martinez-Noriega Edgar Josafat**, and Tetsu Narumi, "CUDA Offloading for Molecular Dynamics Simulation", *21st Computational Engineering Conference*, Niigata, Japan, May 2016. (The contents of Chapter 5)

3. Tetsu Narumi, Minoru Oikawa, **Martinez-Noriega Edgar Josafat**, and Kenji Yasuoka, "DS-CUDA: GPU Virtualization Middleware to Support Migration Functionality", *153th High Performance Computing Research*, Ehime, Japan, February 2016. (The contents of Chapter 4)

Others

International conferences

1. **Martinez-Noriega Edgar Josafat**, Atsushi Kawai, Kazuyuki Yoshikawa, Kenji Yasuoka and Tetsu Narumi, "Running CUDA through GPU virtualization", *GPU Technology Conference (GTC)*, USA, San Jose CA, P4160, March 2014.

2. **Martinez-Noriega Edgar Josafat**, Atsushi Kawai, Kazuyuki Yoshikawa, Kenji Yasuoka and Tetsu Narumi, "CUDA on Android tablets", *Super Computing Conference (SC)*, USA, Denver, November 2013.

3. **Martinez-Noriega Edgar Josafat**, Gualberto Aguilar Torres, and Gabriel Sanchez Perez, "Alto Rendimiento en Simulaciones Moleculares Dinamicas a traves de la Unidad de Procesamiento Grafico", *9th Student Congress on Prototypes and Projects of Computer Engineering*, Mexico, Mexico City, June 2012.

Domestic conferences

1. **Martinez-Noriega Edgar Josafat**, Atsushi Kawai, Kazuyuki Yoshikawa, Kenji Yasuoka and Tetsu Narumi, "CUDA enabled for Android Tablets through DS-CUDA", *Annual Symposium on Advance Computing Systems and Infrastructures (SACSIS 2013)*, Sendai,Japan, May 2013.

2. Kazuyuki Yoshikawa, **Martinez-Noriega Edgar Josafat**, Atsushi Kawai, Kenji Yasuoka and Tetsu Narumi, "Reliability improvement of GPGPU system using DS-CUDA", *Annual Symposium on Advance Computing Systems and Infrastructures (SACSIS 2013)*, Sendai,Japan, May 2013.

3. **Martinez-Noriega Edgar Josafat**, and Tetsu Narumi, "High Performance N-Body Simulation and Visualization through CUDA Architecture", *Bulletin of the University of Electro-communications*, Tokyo, Japan, pp. 59-64, March 2011.

References

[1] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossvry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang., *"Anton, a special-purpose machine for molecular dynamics simulation. "*, In Proceedings of the 34th International Symposium on Computer Architecture, June 2007.

[2] Bakker, A.F., Gilmer,G.H.,Grabow, M.H., Thompson,K. *"A special purpose computer for molecular dynamics calculations "*, J.Comput. Phys. 1990, 90, 313-35.

[3] Fine, R., Dimmler, G., Levinthal, C. *"FASTRUN: A special purpose, hardwired computer for molecular simulation "*, Protein Struc. Funct. Genet. 1991, 11, 242-53.

[4] Yuri N. *"Performance analysis of clearspeed's CSX600 interconnects, in Parallel and Distributed Processing with Applications "*, 2009 IEEE International Symposium, pp. 203-10

[5] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. *"Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations. "*, In Proceedings of the ACM/IEEE SC2003 Conference, November 2003.

[6] England, J.N., *"A system for interactive modeling of physical curved surface objects."*, In Proceedings of SIGGRAPH 78 1978, 336-340. 1978.

[7] Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. and Varshney, *"A. Real-Time Procedural Textures"*, In Proceedings of Symposium on Interactive 3D Graphics 1992, ACM / ACM Press, 95-100. 1992.

[8] Potmesil, M. and Hoffert, E.M., *"The Pixel Machine: A Parallel Image Computer."*, In Proceedings of SIGGRAPH 89 1989, ACM, 69-78. 1989.

[9] Top500 Supercomputer Sites. Top500 and Green500 Supercomputers lists - June 2019, https://www.top500.org/lists/2019/6/ [October 2019].

[10] Scogland TRW, Lin H, Feng WC. A first look at integrated GPUs for green high-performance computing. *Computer Science-Research and Development*, 2010;25:125-134.

[11] Narumi T. DS-CUDA: A Handy Tool to Use GPUs in a Cloud Network. *Tsubame ESJ.: e-Sciencie Journal*, March 2017, 15;12-17.

[12] Y Weng, C Cao, Q Hou, K Zhou, *"Real-time facial animation on mobile devices "*, Computational Visual Media Conference 2013,Volume 76, Issue 3, May 2014, Pages 172:179.

[13] Pei-Jung Lin, Sheng-Chang Chen, Yi-Hsung Li, Meng-Syue Wu, Shih-Yue Chen, *"An Implementation of Augmented Reality and Location Awareness Services in Mobile Devices "*, Lecture Notes in Electrical Engineering Volume 274, 2014, pp 509-514.

[14] M Bedford, T Wheeler, J Bloor, *"Directing specialist care through alerting to mobile devices "*, International Digital Health and Care Congress, The King's Fund, London, September 10-12 2014.

[15] M Miknis, P Plassmann, C Jones, *"Virtual environment stereo image capture using the Unreal Development Kit"*, Computer and Information Technology (GSCIT),14-16 June 2014,1 - 5.

[16] S Burigat, L Chittaro, *"Visualizing the results of interactive queries for geographic data on mobile devices"*, Proceedings of the 13th annual ACM international workshop on Geographic information systems,Pages 277 - 284, New York, NY, USA 2005.

[17] Krone, M., Bidmon, K., Ertl, T. Interactive visualization of molecular surface dynamics. *IEEE transactions on visualization and computer graphics*, 15(6), pp.1391-1398.

[18] Stone, J. E., Messmer, P., Sisneros, R., Schulten, K. High performance molecular visualization: In-situ and parallel rendering with EGL. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1014-1023. IEEE, 2016.

[19] Nonaka, J., Sakamoto, N., Shimizu, T., Fujita, M., Ono, K., Koyamada, K. Distributed Particle-based Rendering Framework for Large Data Visualization on HPC Environments. *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 300-307. IEEE, 2017.

[20] Sabou, A., Gorgan, D. Remote interactive visualization for particle-based simulations on graphics clusters. *40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 253-258. IEEE, 2017.

[21] Lin YD, Chu ETH, Lai YC, Huang TJ. Time-and-Energy-Aware Computation Offloading in Handle Devices to Coprocessors and Clouds. *IEEE Systems Journal*, 2015;9:393-405.

[22] Elgendy IA, El-Kawkagy M, Keshk A. Improving the Performance of Mobile Applications Using Cloud Computing. *The 9th International Conference on Informatics and Systems (INFOS2014)*, December 2014, Cairo, Egypt;109-115.

[23] Kolb J, Chaudhary P, Schillinger A, Chandra A, Weissman J. Cloud-Based, User-Centric Mobile Application Optimization. *Cloud Engineering (IC2E)*, 2015, IEEE International Conference, 2015;26-35.

[24] Acosta A, Almeida F. Parallel Implementations of the Particle Filter Algorithm for Android Mobile Devices. *in Parallel, Distributed and Network-Based Processing (PDP)*, March 2015, 23rd Euromicro International Conference;244-247.

[25] Fatica M, Phillips EH. Synthetic Aperture Radar imaging on a CUDA-enabled mobile platform. *High Performance Extreme Computing Conference*, 2014, HPEC;1-5.

[26] Ju, Q., Chen, S. T., Zhang, Y. Benchmarking renderscript: potential for energy efficient multi-core mobile devices. *12th Annual IEEE International Conference on Sensing, Communication, and Networking-Workshops*, pp. 1-6. IEEE, 2015.

[27] Kemp R, Palmer N, Kielmann T, Bal HE, Aarts B, Ghuloum AM. Using RenderScript and rCUDA for Compute Intensive Tasks on Mobile Devices: a Case Study. *Software Engineering (Workshops)*, 2013;13:305-318.

[28] Eom H, Juste PS, Figueiredo R, Tickoo O, Illikkal R, Iyer R. OpenCL-Based Remote Offloading Framework for Trusted Mobile Cloud Computing. *Parallel and Distributed Systems (ICPADS)*, December 2013, International Conference;240-248.

[29] Montella R, Giunta G, Laccetti G, Lapegna M, Palmieri C, Ferraro C, Pelliccia V, Hong C, Spence I, Nikolopoulos D. On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework. *International Journal of Parallel Programming*, Oct 2017, 45;5:1142-1163.

[30] Reaño C, Prades J, Silla F. Exploring the Use of Remote GPU Virtualization in Low-Power Systems for Bioinformatics Applications. *In Proceedings of the 47th International Conference on Parallel Processing Companion (ICPP)*, International Conference on Parallel Processing Companion, 2018;8:1-8.

[31] Pratapa, S., Krajcevski, P., Manocha, D. MPTC: video rendering for virtual screens using compressed textures. *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, p. 14. ACM, 2017.

[32] McCarthy, D., Schulze, J., Urgen, P. Distributed VR rendering using NVIDIA OptiX. *Electronic Imaging*, 29;2017(3):36-41, 2017.

[33] Stone, J. E., Sherman, W. R., Schulten, K. Immersive molecular visualization with omnidirectional stereoscopic ray tracing and remote rendering. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1048-1057. IEEE, 2016.

[34] Lindholm, E., Kligard, M. J., and Moreton, H. A user-programmable vertex engine. *In Proceedings of SIGGRAPH 2001*, ACM Press/Addison-Wesley Publishing Co., 149:158.

[35] Mark, W. R., Glanville, R. S., Akeley, K., and Kil- gard, M. J. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions*. Graph. 22, 3, 896:907.

[36] Thompson, C. J., Hahn, S., and Oskin, M. Using modern graphics architectures for general-purpose computing: A framework and analysis. *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002, pp. 306-317.

[37] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P. Brook for GPUs: stream computing on graphics hardware. *ACM transactions on graphics (TOG)*, 2004 Aug 1;23(3):777-86.

[38] Kirk, D. B., W. M., W. H.. Programming massively parallel processors: a hands-on approach. *Book*,Morgan kaufmann; 2016 Nov 24.

[39] Patterson, D. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges. *Nvidia Whitepaper,* 2009 Sep 30;47.

[40] ANSI-IEEE 754-1985. *"American National Standard – IEEE Standard for Binary Floating-Point Arithmetic."*, American National Standards Institute, Inc., New York, 1985.

[41] Lee, G., Chun, B. G., Katz, Y. H. Heterogeneity-Aware Resource Allocation and Scheduling. *University of California Workshop,* California, Berkeley 2011.

[42] Exposito, R. R., Taboada, G. L., Ramos, S., Tourino, J., Doallo, R. General-purpose computation on GPUs for high performance cloud computing, *Concurrency and Computation: Practice and Experience,* no. 12 (2013): 1628-1642.

[43] Green, S. Particle simulation using cuda. *NVIDIA whitepaper,* 2010 May;6, pp.121-128.

[44] Glaser, J., Nguyen, T. D., Anderson, J. A., Lui, P., Spiga, F., Millan, J. A., Glotzer, S. C. Strong scaling of general-purpose molecular dynamics simulations on GPUs. *Computer Physics Communications,* 2015 Jul 1;192:97-107.

[45] Goldberg, R. P. Survey of virtual machine research. *Computer,* 1974 Jun;7(6):34-45.

[46] Hong, C. H., Spence, I.,Nikolopoulos, GPU virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 2017 Oct 9;50(3):35.

[47] FreeDesktop.org. Noveau: Accelereted open source driver for nvidia cards, https://nouveau.freedesktop.org/wiki [September 2019].

[48] Menychtas, K., Shen, K., Scott, M. L. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. *In Presented as part of the 2013 USENIX Annual Technical Conference*, 2013 (pp. 291-296).

[49] Herrera, A. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. *White paper - Nvidia Corp*,2014:1-8.

[50] Van Doorn, L. Hardware virtualization trends. *ACM Usenix International Conference On Virtual Execution Environments: Proceedings of the 2nd international conference on Virtual execution environments* vol. 14, no. 16, pp. 45-45. 2006.

[51] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Wiegert, J. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 2006 Aug 1;10(3).

[52] Lagar-Cavilla, H. A., Tolia, N., Satyanarayanan, M., De Lara, E. VMM-independent graphics acceleration. *Proceedings of the 3rd international conference on Virtual execution environments*, pp. 33-43. ACM, 2007.

[53] Hansen, J. G. Blink: Advanced display multiplexing for virtualized applications. *Proceedings of NOSSDAV*, 2007 Jun 4.

[54] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D.,Klosowski, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM transactions on graphics (TOG)*, vol. 21, no. 3, pp. 693-702. ACM, 2002.

[55] Kuzkin, M. A., Tormasov, Method and system for remote device access in virtual environment. *U.S. Patent No. 8,805,947*, Patent 8,805,947, issued August 12, 2014.

[56] Lee, C., Kim, S. W., Yoo, C. VADI: GPU virtualization for an automotive platform. *IEEE Transactions on Industrial Informatics*, no. 1 (2015): 277-290.

[57] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., Ranganathan, P. GViM: GPU-accelerated virtual machines. *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pp. 17-24. ACM, 2009.

[58] Shi, L., Chen, H., Sun, J., Li, K. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, no. 6 (2011): 804-816.

[59] Giunta, G., Montella, R., Agrillo, G., Coviello, G. A GPGPU transparent virtualization component for high performance computing clouds. *European Conference on Parallel Processing*, pp. 379-391. Springer, Berlin, Heidelberg, 2010.

[60] Li, T., Narayana, V. K., El-Araby, E., El-Ghazawi, T. GPU resource sharing and virtualization on high performance computing systems. *International Conference on Parallel Processing* pp. 733-742. IEEE, 2011.

[61] Gupta, V., Schwan, K., Tolia, N., Talwar, V., Ranganathan, P. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. *USENIX Annual Technical Conference (USENIX ATC 11)*, p. 31. 2011.

[62] Merritt, A. M., Gupta, V., Verma, A., Gavrilovska, A., Schwan, K. Shadowfax: scaling in heterogeneous cluster systems via GPGPU assemblies. *Proceedings of the 5th international workshop on Virtualization technologies in distributed computing*, pp. 3-10. ACM, 2011.

[63] Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Feng, W. C. VOCL: An optimized environment for transparent virtualization of graphics processing units. *Innovative Parallel Computing (InPar)*, pp. 1-12. IEEE, 2012.

[64] Duato, J., Pena, A. J., Silla, F., Mayo, R., Quintana-Orti, E. S. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. *International Conference on High Performance Computing and Simulation*, pp. 224-231. IEEE, 2010.

[65] Oikawa, M., Kawai, A., Nomura, K., Yasuoka, K., Yoshikawa, K., Narumi, T. DS-CUDA: a middleware to use many GPUs in the cloud environment. *SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1207-1214. IEEE, 2012.

[66] Defanti, Thomas A., and Maxine D. Brown. Visualization in scientific computing. *Advances in Computers*, 1991 Jan 1, Vol. 33, pp. 247-307.

[67] Harvey, M.J., Giupponi, G., De Fabritiis, G. *"ACEMD: Accelerating biomolecular dynamics in the microsecond time scale"*, J. Chem. Theory Comput. 2009, 5, 1632-9.

[68] Friedrichs, M.S., Eastman, P., Eastman, P., Vaidyanathan, V., Houston, M., Le Grand, S., Beberg, A.L. Ensing, D. L., Bruns, C.M., Pande, *"Accelerating molecular dynamic simulation on graphics processing units."*, J. Comput. Chem. 2009, 30, 864-72.

[69] G. Shi and V. Kindratenko, *"Implementation of NAMD molecular dynamics non-bonded forcefield on the Cell Broadband Engine processor"*, In Proceedings of the 9th International Workshop on Parallel and Distributed Scientific and Engineering Computing, April 2008.

[70] Hailong Yang, Bo Li, Yongjian Wang, Zhongzhi Luan, Depei Qian and Tianshu Chu *"Accelerating Dock6s Amber Scoring with Graphic Processing Unit "*, Department of Computer Scinece and Engineering, Sino-German Joint Software Institute, Beihang University, 2010, China.

[71] Brooks, B. R., Brooks III, C. L., Mackerell Jr, A. D., Nilsson, L., Petrella, R. J., Roux, B., Caflisch, A. CHARMM: the biomolecular simulation program, *Journal of computational chemistry*, 2009 Jul 30;30(10):1545-614.

[72] Ribarsky, William, Yves Jean, Thomas Kindler, Weiming Gu, Gregory Eisenhauer, Karsten Schwan, and Fred Alyea, An integrated approach for steering, visualization, and analysis of atmospheric simulations, *In Proceedings IEEE Visualization*, vol. 95. 1995.

[73] Beazley, David M., and Peter S. Lomdahl, Lightweight computational steering of very large scale molecular dynamics simulations, *In Supercomputing 96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, pp. 50-50. IEEE, 1996.

[74] Vetter, Jeffrey Scott, and Karsten Schwan, Progress: A toolkit for interactive program steering, *Georgia Institute of Technology*, 1995.

[75] Fukushige, T., Taiji, M., Makino, J., Ebisuzaki, T., and Sugimoto, D., *"A Highly-Parallelized Special-Purpose Computer for Many-body Simulations with An Arbitrary Central Force: MD-GRAPE."*, Astrophysical Journal, 468, pp. 51-61, 1996.

[76] Taiji, M., Fukushige, T., Makino, J., Ebisuzaki, T., and Sugimoto, D., *"MD-GRAPE: A Parallel Special-Purpose Computer System for Classical Molecular Dynamics Simulations."*, Physics Computing '94 Lugano, Switzerland, in Proceedings of the 6th Joint EPS-APS international conference on Physics Computing, European Physical Society, Geneva, pp. 200-203, 1994.

[77] University of Fukui, Department of Applied Physics. Real Time Molecular Dynamics Simulation and Visualization - Claret Ver 0.53, http://polymer.apphy.u-fukui.ac.jp/˜koishi/claret/index.php [October 2018].

[78] Freeglut - The Free OpenGL Utility library - Sep 2019, http://freeglut.sourceforge.net [September 2019].

[79] M.P. Tosi,F.G. Fumi, *"J. Phys.Chem. Solids"*, 25, 1964, 45.

[80] M.P. Allen,D.J. Tildesley, *"Computer Simulation Liquids"*, Clarendon,Oxford,1987.

[81] Kiss, G., Khan, N. H., Tegnander, E., Eik-Nes, S. H., Torp, H. Fast ultrasound signal and image processing on a tablet device. *In 2015 IEEE International Ultrasonics Symposium,* 2015 Oct 21 (pp. 1-4). IEEE.

[82] Sabou, A., Gorgan, D.Remote interactive visualization for particle-based simulations on graphics clusters. *40th International Convention on Information and Communication Technology, Electronics and Microelectronics,* 2017 May 22 (pp. 253-258). IEEE.

[83] Stone, J. E., Sherman, W. R., Schulten, K. Immersive molecular visualization with omnidirectional stereoscopic ray tracing and remote rendering. *In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW),* pp. 1048-1057. IEEE, 2016.

[84] Krone, M., Bidmon, K., Ertl,T. Interactive visualization of molecular surface dynamics. *IEEE transactions on visualization and computer graphics 15,* no. 6 (2009): 1391-1398.

[85] Stone, J. E., Gullingsrud, J., Schulten, K. A system for interactive molecular dynamics simulation. *In Proceedings of the 2001 symposium on Interactive 3D graphics,* pp. 191-194. ACM, 2001.

[86] FGLFW multi-platform Utility library library for OpenGL, OpenGL ES and Vulkan, https://www.glfw.org [September 2019].

[87] Matthias Trapp, *"OpenGL-Performance and Bottlenecks"*, Seminar, University of Postdam, Winter semester 2003.

[88] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices.", Workshop on Parallel Programming and Applications on Accelerator Clusters, 2010.

[89] J.Duato, A.J.Pena, F.Silla, R.Mayo, and E.S.Quintana, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters.", 2011 IEEE International Conference on Parallel Processing, 2011, pp. 365:374.

[90] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability.", The Fourth International Conference on Future Computational Technologies and Applications, Nice, France, 2012, pp.8-10.

[91] J-H. Huang, "Opening Keynote at GTC 2015:Leaps in Visual Computing.", GPU Technology Conference, Silicon Valley,Keynote presentation, April 4-7, 2016.

[92] Atsushi Kawai, Kenji Y asuoka, Kazuyuki Yoshikawa, and Tetsu Narumi, *"Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programability and Reliability"*, The Fourth International Conference on Future Computational Technologies and Applications, Nice, France, 2012.

[93] Narumi Laboratory Web Page, "DS-CUDA Software Package", http://narumi.cs.uec.ac.jp/dscuda/

[94] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, *"DS-CUDA:a Middleware to Use Many GPUs in the Cloud Environment"*, SC Companion:High Performance Computing, Networking Storeage and Analysis, pp. 1207-1213, 2013.

[95] Android Developer Sites. Android NDK, http://developer.android.com/intl/es/tools/sdk/ndk/index [October 2019].

[96] Android Developer Sites. JNI Tips, http://developer.android.com/intl/es/training/articles/perf-jni [October 2019].

[97] Khronos group. The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl [October 2018].

[98] CUDA for developers. CUDA Zone, https://developer.nvidia.com/cuda-zone [October 2019].

[99] Huang J-H. Opening Keynote at GTC 2018. *GPU Technology Conference*, March 2018, Silicon Valley.

[100] Stone JE, Gullingsrud J, Schulten K. A system for interactive molecular dynamics simulation. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 2001;I3D'01:191-194.

[101] Luehr N, Jin AG, Martinez TJ. Ab Initio Interactive Molecular Dynamics on Graphical Processing Units (GPUs). *Journal of Chemical Theory and Computation*, 2015;11:4536-4544.

[102] Giunta G, Montella R, Agrillo G, Coviello G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. *European Conference on Parallel Processing - (Euro-Par 2010)*, 2010, 6271;379-391.

[103] Merrit A, Gupta V, Verma A, Gavrilovska A, Schwan K. Shadowfax: Scaling in Heterogeneous Cluster Systems via GPGPU Assemblies. *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*, June 2011, 11;3-10.

[104] Suzuki Y, Kato S, Yamada H, Kono K. Gpuvm: Gpu virtualization at the hypervisor. *IEEE Transactions on Computers*, September 2016, 1;65:2752-2766.

[105] Barak A, Ben-Nun T, Levy E, Shiloh A. A package for OpenCL-based heterogeneous computing on clusters with many GPU devices. *Workshop on Parallel Programming and Applications on Accelerator Clusters*, IEEE International Conference, 2010;1-7.

[106] Shi L, Chen H, Sun J. vCUDA: GPU accelerated high performance computing in virtual machines. *In Proc. of the IEEE Parallel and Distributed Processing Symposium*, IPDPS, 2019;1-11.

[107] Liang TY, Chang YW. GridCuda: A Grid-Enabled CUDA Programming Toolkit. *In Proc. of the IEEE Advanced Information Networking and Applications Workshops*, 2011, WAINA;141-146.

[108] Kawai A, Yasuoka K, Yoshikawa K, Narumi T. Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability. *The Fourth International Conference on Future Computational Technologies and Applications*, 2012, Nice, France;8-10.

[109] Reaño C, Silla F, Shainer G, Schultz S. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. *In Proceedings of the International Middleware Conference*, 2015, Middleware Industry 15;4:1-7.

[110] Reaño C, Silla F. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. *In 2015 IEEE International Conference on Cluster Computing*, 2015, CLUSTER 15, IEEE Computer Society;488-489.

[111] Duato J, Pena AJ, Silla F, Mayo R, Quintana-Orti ES. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. *High Performance Computing and Simulation (HPCS)*, 2010, International Conference on IEEE;224-231.

[112] Duato J, Pena AJ, Silla F, Mayo R, Quintana-Orti ES. Performance of CUDA Virtualized Remote GPUs in High Performance Clusters. *IEEE International Conference on Parallel Processing*, 2011;365-374.

[113] Silla F, Iserte S, Reano C, Prades J. On the benefits of the remote GPU virtualization mechanism: The rCUDA case. *Concurrency and Computation: Practice and Experience*, 2017;29:e4072.

[114] Reaño C, Silla F, Castello A, Peña A, Mayo R, Quintana-Orti E, Duato J. Improving the user experience of the rCUDA remote GPU virtualization framework. *Concurrency and Computation: Practice and Experience*, 2015, 27;14:3746-3770.

[115] Huang S, Xiao S, Feng WC. On the energy efficiency of graphics processing units for scientific computing. *Parallel & Distributed Processing, IPDPS*, 2009, IEEE International Symposium;1-8.

[116] Azmat S, Wills L, Wills S. Parallelizing Multimodal Background Modeling on a Low-Power Integrated GPU. *Journal of Signal Processing Systems*, 2016;1-11.

[117] Tosi MP, Fumi FG. Ionic sizes and born repulsive parameters in the NaCl-type alkali halides—II: The generalized Huggins-Mayer form. *Journal of Physics and Chemistry of Solids*, 1964;25:45-52.

[118] Fujii, Y., Azumi, T., Nishio, N., Kato, S., Edahiro, M. Data transfer matters for GPU computing. *International Conference on Parallel and Distributed Systems*, pp. 275-282. IEEE, 2013.

[119] van Werkhoven, B., Maassen, J., Seinstra, F. J., Bal, H. E. Performance Models for CPU-GPU Data Transfers. *ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 11-20, 2014.

[120] White III, J. B., Dongarra, J. J, Overlapping computation and communication for advection on hybrid parallel computers. *International Parallel and Distributed Processing Symposium*, pp. 59-67. IEEE, 2011.

[121] GoMez-Luna, J., GonzaLez-Linares, J. M., Benavides, J. I., Guil, N. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing*, no. 9 (2012): 1117-1126.

[122] Martinez-Noriega Edgar Josafat, Narumi Tetsu, High Performance N-Body Simulation and Visualization through CUDA Architecture *The 25th UEC International Mini-Conference for International Students*, Tokyo-Japan, March, 2011, pp 59,64.

[123] Demir, V., Elsherbeni, A. Z. Utilization of CUDA-OpenGL interoperability to display electromagnetic fields calculated by FDTD. *CEM'11 Computational Electromagnetics International Workshop*, pp. 95-98. IEEE, 2011.

[124] Abdellah, M., Eldeib, A., Owis, M. I. GPU acceleration for digitally reconstructed radiographs using bindless texture objects and CUDA/OpenGL interoperability. *37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 4242-4245. IEEE, 2015.

[125] Camillo, M. S., Shin-Ting, W. Accessing CUDA features in the OpenGL rendering pipeline: A case study using N-Body simulation. *30th SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 315-322. IEEE, 2017.

[126] NVIDIA Developer Sites - NVIDIA Video Codec SDK, https://developer.nvidia.com/nvidia-video-codec-sdk [October 2019].

[127] Wilhelmsen, M. A., Stensland, H. K., Gaddam, V. R., Mortensen, A., Langseth, R., Griwodz, C., Halvorsen, P. Using a commodity hardware video encoder for interactive video streaming. *IEEE International Symposium on Multimedia*, pp. 251-254. IEEE, 2014.

[128] de Souza, D. F., Ilic, A., Roma, N., Sousa, L. GHEVC: An efficient HEVC decoder for graphics processing units. *IEEE Transactions on Multimedia*, 19(3), pp.459-474, 2016.

[129] NVIDIA Developer Sites - NVIDIA Capture SDK, https://developer.nvidia.com/capture-sdk [October 2019].

[130] VirtualGL 3D without Boundaries - The VirtualGL project, https://virtualgl.org/ [October 2019].

[131] Nagella, S., Sastry, L., Fowler, R. Remote rendering on visualization cluster using VirtualGL and Chromium. *Proc. VizNET Conf.*, October 2008.

[132] Nye, A. X Protocol Reference Manual: For X11, *Book*, Release 6. "O'Reilly Media, Inc.", 1995.

[133] Rosmanith, H., Volkert, J. Traffic forwarding with GSH-GLOGIN. *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 213-219. IEEE, 2005.

[134] Google sites - Stadia, games without a console or downloads, https://store.google.com/srp=/product/stadia [October 2019].

[135] PlayStation sites - PlayStation Now, gaming on demand, https://www.playstation.com/en-us/explore/playstation-now [October 2019].

[136] Xbox sites - Xbox Game Streaming Preview, https://www.xbox.com/en-US/xbox-game-streaming [October 2019].