平成26年度　修士論文

High Performance Computing on Mobile Devices

電気通信大学大学院　情報理工学研究科
情報・通信工学専攻　成見研究室
学籍番号　1331098
氏名　MARTINEZ NORIEGA Edgar Josafat
指導教員　成見　哲教授
副指導教員　寺田実　准教授

平成27年1月30日

概要

モバイルデバイス上の高性能コンピューティング

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

近年GPGPU (General Purpose Computing on Graphics Processors Units)が盛んになっており、GPUを使って特定のアプリケーションを高速化することが行われている。GPUはスーパーコンピューターを構築する有効な選択肢の一つとなっており、現在のトップ10のスーパーコンピューター中の2台がPFlops(10^{15}の浮動小数点演算/秒)を達成するためにGPUが使われている。GPUはコアを沢山備えた並列処理装置であり、階層的なメモリ構造になっている。このため、良いコストパフォーマンスが期待できる。2006年にNVIDIAがCUDA(Compute Unified Device Architecture)をリリースして以来、GPUを簡単にプログラムすることが出来るようになった。

一方、携帯性・低消費電力・タッチスクリーン・接続性など独特の機能を持つモバイル機器は、コンピュータとやり取りするための新しい潮流として興味深い進化を遂げている。アプリケーションプログラムから見ると、人々がどうやってデータを扱うか、どうやってデータを可視化すべきか、などが変わってきている。モバイル機器にも何らかのGPUが内蔵されていることが多いが、ARM CPUを主流とするモバイル機器においてはそれ程性能を期待することは出来ない。

クラウドやグリッドコンピューティングは、サーバー側の高性能な計算能力をモバイル機器から活用するための有望なアプローチの一つである。GPU仮想化ツールであるDS-CUDA (Distributed-Shared CUDA)も、そのような技術の一つである。DS-CUDAはソフトウェアレベルでGPUを仮想化するもので、クライアントマシンからローカルネットワーク経由でリモートのGPUを透過的に使うことが出来る。ソースコードを変更しなくても、リコンパイルするだけでGPUを搭載しないデバイスで見かけ上GPUを使ってアプリケーションを加速することができる。しかし、クライアントとサーバ間の通信がボトルネックになる可能性がある。

本研究では、Androidデバイス上でNaClの分子動力学シミュレーションを加速し

た。DS-CUDAを使うことでGPUを使って原子間の力及び速度の計算を高速化することが出来た。もともとLinuxしかサポートされていなかったDS-CUDAをAndroid上で動作させるために、NDK(ネイティブ開発キット)を用いてCコードを動作させ、OpenGL ES 1.1を用いて描画した。クライアントマシンとしてNVIDIAのSHIELDタブレットを使用し、サーバとしてGeForce GTX 680Mが搭載されているノートパソコンを使用した。サーバーはKnoppix 7.1 とCUDA 6.0上で動作させた。当研究ではKnoppix内にGPUやCUDA,DS-CUDAを動作させる環境を組み込んだは「Knoppix for CUDA」と呼ぶディストリビューションを作っており、そのDVDからブートするだけでDS-CUDAサーバーが構築できる。

5,832原子のシミュレーションを行った場合、SHIELDタブレットのCPU上では0.073 GFLOPSの速度しか出なかったものが、本研究により5700倍高速の420 GFLOPSの計算速度を達成した。システムのボトルネックは主にネットワークの帯域幅であるが、原子数が多い場合（例えば5,382原子）には球を描画するルーチンがネックであることが分かった。原子数が多い場合は計算量が大きくなるため相対的にネットワークはボトルネックとはならない。

Abstract

High Performance Computing on Mobile Devices

by

MARTINEZ NORIEGA Edgar Josafat

The University of Electro-Communications

Professor Narumi Tetsu

GPGPU (General Purpose Computing on Graphics Processors Units) has become one of the most common ways to accelerate scientific applications and built supercomputers. At the present time, 2/10 top supercomputers are equipped with GPUs to achieve PFlops (10^{15} floating operation per second) of performance. The GPU is designed with massively programmable parallel processors, different memory hierarchy and many core chips, thus are really attractive due to its performance/cost benefit. NVIDIA is one of the pioneers to offer an easy way to program and develop for GPUs through CUDA (Compute Unified Device Architecture), released in 2006.

On the other hand, mobile devices are becoming another interesting way to interact with computers due to its specific and particular capabilities:mobility, portability, low power consumption, touch-screen, connectivity, and others. In this scenario, applications are changing the direction in which people interact and visualize data. Mobile devices are dotted with GPUs as well, however these ones cannot be easily used to achieve more performance than only using the mobile processor e.g ARM.

Cloud or grid computing is one of the promising approaches to exploit the computing power on the server side for mobile devices using high performance computing frameworks such as DS-CUDA (Distributed-Shared CUDA). DS-CUDA is a GPU virtualization tool at software level that allow us to borrow an NVIDIA GPU remotely from our local network to accelerate an app inside of a device/computer which does not contain a physical GPU. DS-CUDA enables us to use our own CUDA source code without any major modifications to accelerate the application, however communication between the client and server might become bottleneck.

We implemented an NaCl MD (Molecular Dynamics) simulation with CUDA on an Android

Device. On the simulator the behaviour between several ions of Na+ and Cl- at vacuum level is shown. We accelerate the computation of force, velocity and coordinate using CUDA via DS-CUDA. Other process are running on C code through NDK (Native Development Kit) and OpenGL ES 1.1 is used for rendering the visuals. We used an NVIDIA's SHIELD tablet as a client, and a laptop as a server equipped with GeForce GTX 680M running with Knoppix 7.1 linux and CUDA 6.0. The server can be built just by booting from "KNOPPIX for CUDA" DVD, which we distribute on the web. The server is connected via wired Gigabit-ethernet to our access point and the client is connected via Wifi 802.11n. We reached up to 420 Gflops in force computation on a simulation with 5,832 ions, 5,700 times faster than the 0.073 Gflops delivered from CPU implementation on SHIELD. The bottleneck in our simulation is the bandwidth delivered from our local LAN, although we tried to minimize the communication between server and client, updating every 100 MD steps. Rendering the spheres is also a bottleneck in the application, for large amount of particles (5,382 ions), however the amount of operations per second becomes larger as the number of bodies increases.

Acknowledgements

Special thanks goes to Professor Narumi Tetsu for all its support since the beginning of this journey. For all its advices, knowledge and encourage to continue. For all opportunities he had gave me and believed in me. For all support during my permanency in Japan. Also to Professor Terada for leading this project. As well as for Dr. Oikawa and Dr. Takai for their valuable advices over DS-CUDA and molecular dynamics. I want to express gratitude as well to Dr. Rio which helps a lot in our inter ship in Saudi. To professor Dr. Yasuoka and all CREST members for giving me the chance to form part of it. For profesor Atsushi Kawai who has tough me everything of DS-CUDA and also its special support during hard times. To professor Suwako who has driven me for a lot of new opportunities and various points of view.

Special thanks to my friends Kadri and Waya, for their big support during this 3 years in Japan. To all my lab members for their support in my Japanese language, specially Shiotani Kun.

To all my family in Mexico for their unconditional support. To all my cousins for all their experiences and support. To my mother Edith and father Raul who have been the pillars of my education and support in all my professional career and life. To my brother Raul who has been an inspiration to follow and my best friend in life.

Lastly, special thanks to MengMeng Wang for her special support.

Contents

List of Figures

List of Tables

1

Introduction

High Performance Computing is a field that has been evolved through the years in the way it is applied (Software), how it is implemented (Hardware) and how we used it. In the beginning with first big computers (ancestor of the super computers) such as the Electronic Numerical Integrator Computer (ENIAC) and the Universal Automatic Computer (UNIVAC) a whole room of a building was need it to fit this only 1K Floating Operation per Second (FLOPS). The applications fot his kind of machines where mainly for military purposes. One big advancement on the reduction of this big machines was the introduction of the TTL technology at the end of the 70's. Companies like Intel, ARM, Zilog, IBM and Motorola between the most famous at that time, started the development of micro processors for what will become the personal computing. Clusters and mainframes where build with these technology processors but there was a major difficulty in the programming, communication between nodes and the algorithm implementation. There has been the development as well of accelerators which are dedicated for special purposes such as molecular dynamics. These specialized hardware presents a highly parallel architecture and multi-core implementation such as the MD-GRAPE [1], Anton [2], ATOMS [3], FASTRUN [4] and C SX600 [5]. However, the development of these specialized hardware, as shown in Table 1.1, is really high considering the small scope range of their applications.

The Graphics Processor Unit (GPU) is another specialized hardware which also presents a parallel architecture design. With the advent of the graphical operating systems, the need for rendering pixels and presents into a display has been increased, and the device which encourage this task has been the GPU. Due to its primitive operations, this device is presents next characteristics:

- Low cost (Compared to other specialized hardware).
- High parallelism.

Accelerator	Manufacture	Estimate cost per node (USD)
CX600	ClearSpeed	~ $10,00
MDGRAPE-3	Riken	~ $9,000,000
ATOMS	AT&T Bell	~ $186,000 (1990)
FASTRUN	Colombia University	~ $17,000 (1989)
GPU	NVIDIA / ATI	~ $200-800

Table 1.1: Cost development for specialized hardware accelerators.

- Optimized for Floating point calculation.
- Massively programmable processors.

It was in the 70's decade when machines like Ikonas [6], the Pixel Machine [7] and Pixel Planes 5 [8] used the GPU to compute something more than pixels. Here, the General Purpose on Graphics Processing Unit (GPGPU) was born. In the same way as with the first parallel computers, managing and programing these devices for another purpose wasn't an easy task due to the buffer manipulation and other shaders which requires enough knowledge of the programming language pipeline such as OpenGL or Direct3D. Nonetheless, NVIDIA a graphic card company decided to invest in this field, creating the Compute Unified Device Architecture (CUDA) in 2006. CUDA is an architecture and programming framework that enables dramatic increases in computing performance by and easily development on GPUs. Since then, CUDA has successfully accelerated applications in many fields:

- Bio-informatics.
- Computational chemistry.
- Computational fluid dynamics.
- Computational structural mechanics.
- Data science.
- Defence.
- Computational finance.
- Imaging and Computer vision.
- Weather and climate.
- Medical imaging.

Also, some of the top of super computers, in the list of TOP500 [21] are equipped with GPUs as accelerators. In order to handle this tumultuous conglomerate of GPUs in cloud

environment some HPC frameworks has been developed such as Distributed-Shared CUDA. This framework helps to solve the major difficulties in programming multi-node heterogeneous computers virtualizing GPUs on a distributed network as if they where attached to a single node. In this sense, using a remote GPU from another device which is not equipped with one accelerator of this kind is a feasible capability of DS-CUDA.

On another scenario, mobile devices are becoming primary devices for various of our main activities such as reading email, taking pictures, playing games, using social networks and also creating our own content. Since the appearance of the "smart-phone" and tablet devices, the effort to integrate many sensors into a mobile device has lead new kind of applications and new services development. New paths to interact with data and also to represent this information to the user. Efforts to create new contents has lead a numerous variety of research topics such as visualization data, virtual reality, health based applications, between others [9] [10] [11] [12] & [13]. However, due to its mobile nature, these devices are equipped with low computation power processors such as ARM and not yet programmable GPUs. Undoubtedly, next generation of mobile device applications will will require a lot of computational power.

We proposed to merge the High Performance Computing applications with the mobile device in this research. Through DS-CUDA framework, utilize remotely CUDA capable GPU on the tablet to run an N-body simulation. An study about performance will be presented as well.

1.1 Research Purpose - Objective

The main purpose of this research is to approach high performance computing applications for mobile devices. These touching screen devices hold a new way to dive into the information presented to the user. First applications prototyped by the author were galaxy simulation or molecular dynamics, Figures 1.1 & 1.2 respectively, on Tablet device. However, to reach this goal an accelerator is needed due to the low power precessing of the mobile devices. Utilizing DS-CUDA framework seems to be a feasible solution since its open source nature and the ability to use remotely a GPU from our local network.

Thus, 3 main objectives are summarized in this work:

First

DS-CUDA compatible with Android. In order to use remotely a GPU from our local network inside of Android tablet device we need to be able to run client libraries and generate proper DS-CUDA stubs for Android. Two approaches are presented: Using a terminal emulator and through the Java Native Development kit.

Second

Create an HPC application for Android. To test the performance of DS-CUDA a

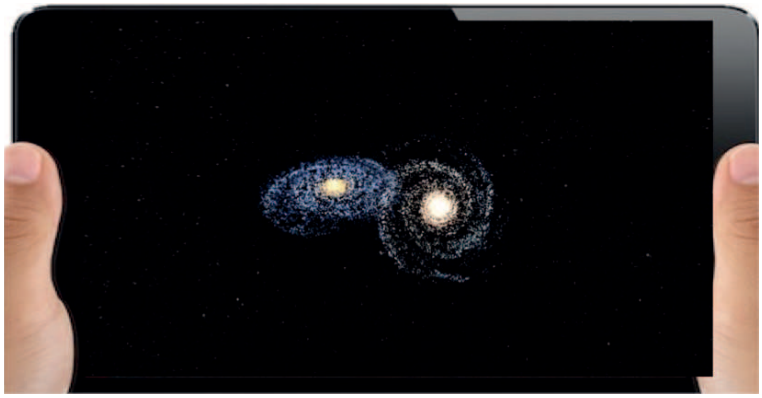

Figure 1.1: Prototype of Mobile-HPC app 1. Galaxy simulation.

Figure 1.2: Prototype of Mobile-HPC app 2. Molecular Dynamics.

molecular dynamics simulation is implemented using OpenGL ES.

Third

Analyse the performance of the application. Modelling all the main process and communication through the simulation and identifying the bottlenecks.

1.2 Thesis Organization

The present work is divided into 7 Chapters. Chapter 1 introduces all the background about the General Purpose computing using GPUs, as well as the importance of the mobile devices and its unique characteristics. Chapter 2 is about CUDA, its programming model and capabilities. Also some introduction for mobile architectures is mentioned. Chapter 3 talks about the mobile devices and its evolution, the capabilities and applications. A description of Android operating system is also included. In Chapter 4 we go through DS-CUDA framework, its overview and usage. Also, here we mentioned a contribution that we made as consequence of this research. Chapter 5 is about "claret" molecular dynamics simulation software, its main capabilities and versions. The description of the port for Android is detailed here. The evaluation of the performance of claret is remarked in Chapter 6, as well as the system architecture prepared for the test evaluation. Last Chapter 7 some remarks and conclusion are listed.

2

CUDA

The graphics cards are designed to render high-quality 3D textures in real time. Due to the large competitive market in this range of devices, the Graphic Processing Unit (GPU) has become a powerful hardware for computing comparatively for low Cost. Interestingly, this hardware become able to compute any kind of data rather than only pixels inside of shaders. This new paradigm has its origins based on the General Purpose Computing on Graphics Processing Units (GPGPU), particularly known for their great programming difficulty due to the high level of knowledge in the graphics pipeline.

In recent years, scientist and developers have started using GPU as an accelerator for their algorithms where they can get benefit of thousands of threads of this device.

2.1 Compute Unified Device Architecture - CUDA

CUDA is a computing architecture and framework developed by NVIDIA, a graphic card company, with the main purpose of facilitate the parallel programming of GPUs. CUDA allow developers to get immerse into the great power that the GPU brings with its many cores and memory hierarchy. Before, graphics cards were only conceived in the high end multimedia, media design and games sector, but nowadays CUDA allows GPU to work in another kind of fields:

- Science
- Engineering
- Economy

High level of knowledge of the graphics pipeline[1] is not needed,and only the knowledge of C language is required.

[1]Is defined as the process or consecutive subroutines that are required to deploy graphics generated by an output device.

Model	Computing Capability	Multiprocessors	CUDA cores
GeForce GTX 560 Ti	2.1	8	384
GeForce GTX 460	2.1	7	336
GeForce GTX 470M	2.1	6	288
GeForce GTS 450, GTX 460M	2.1	4	192
GeForce GT 445M	2.1	3	144
GeForce GT 430, GT 440, GT 435M, GT 425M, GT 420M	2.1	2	96
GeForce GT 520, GT 415M	2.1	1	48
GeForce GTX 580	2.0	16	512
GeForce GTX 570, GTX 480	2.0	15	480
GeForce GT 470	2.0	14	448
GeForce GTX 465, GTX 480M	2.0	11	352
GeForce 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280, GTX 275	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT	1.1	14	112

Table 2.1: First CUDA capable GPUs and its specifications.

It wasn't until five years later of the presentation of GeForce[2] 3 that the general purpose computing was ready for its first steps. In November of 2006, NVIDIA introduced to the world the first CUDA capable GPU GeForce 8800 GTX. This DirectX 10 capable GPU bring the first speed up on science and start the revolution of GPGPU. Since then, NVIDIA continues to develop and release new CUDA capable graphic cards. Table 2.1 shows some of the first CUDA capable devices.

Since NVIDIA wanted to create a whole new family of graphics processing unit for general purpose computing the Arithmetic Logic Unit (ALU) were designed under the standard IEEE 754-1985 [18] for single floating point precision, as well as including many functions not oriented to graphics rendering. Also they implemented a whole new memory hierarchy inside

[2]One of the most famous NVIDIA's GPU branch, dedicated to gamers and high end multimedia PC.

of the device composed up to 5 levels. All of this and other great capabilities are included in CUDA allowing GPU's to achieve general purpose computing and also graphics rendering.

2.2 Programming Model

A structure of a CUDA program is grouped in one or more phases that are executed in the *host* (CPU) or inside of the *device* (GPU). The parts inside of the code which shows a lot of parallelism are executed inside of the device. In the other hand, the serial parts are running in the host side. Thus, a CUDA program is a combination of code execution inside of host and device. NVIDIA provides an special based C compiler called *nvcc* which separates and process the different code for each device, as the Figure[3] 2.1 shows. The code that belongs to the host is ANSI C standard, process by normal C language compilers and runs in an "commercial" CPU. The code executed in the device is processed in some different ANSI C standard that extend key-words for parallel functions called *kernels* and its associated data structures.

2.2.1 Kernels

The called *kernels* are subroutines executed inside of the GPU which in turn call a massive number of *threads*. Each GPU is composed for many Multiprocessors (MP) which also contains tons of threads. Depending of the compute capability, the developer may launch up to 1024 threads per MP or more. In this way, one thread does not process the same information at the same time since each one have a different "Index". This special identifier tells the thread what to do with different data and accessing different memory region. Next, we provided a very easy sample of a kernel.

```
1
2  // Strucutre of a Kernel
3  __global__ void MiKernel(float* x, float* v, float cons)
4       {
5      int i = threadIdx.x;
6      x[i] = x[i] + v [i] * cons;
7      }
8      ......
9      ......
10     ......
11     int main()
12     {
```

[3]Image courtesy of http://developer.nvidia.com//nvidia-gpu-computing-documentation//

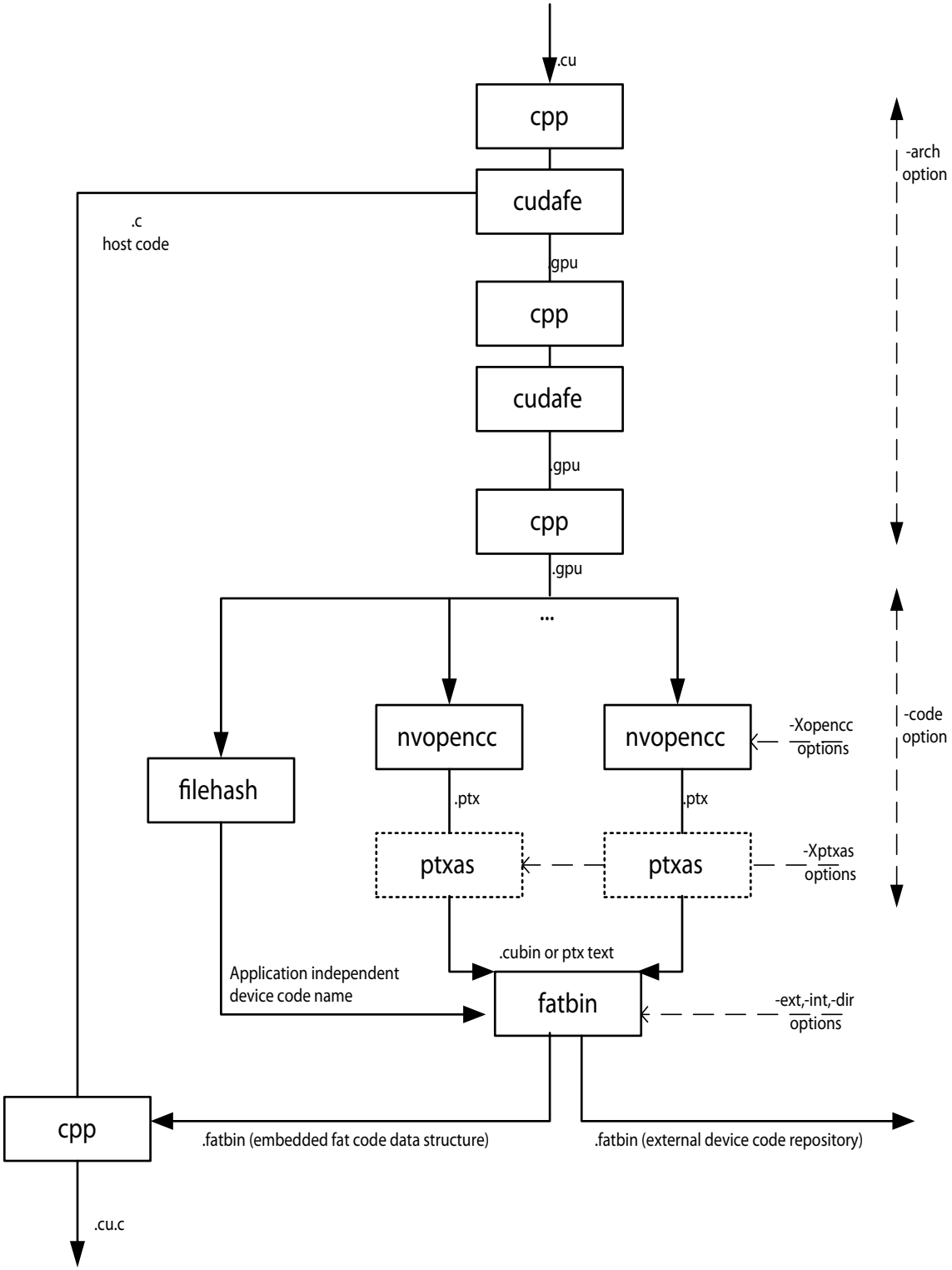

Figure 2.1: Nvcc complete compilation trajectory.

```
13          //Kernel call from the Host
14          MiKernel<<<1,N>>>(X, V, Cons);
15      }
```

As we can see in the kernel sample, an especial identifier of it is the reserve word **__global__** in which the developer can specify the number of threads to be launched in side of the GPU. This is done through the identifier <<<....>>>. The special index for each thread is reachable by one *built-in* variable called **threadIdx**. This patern operates in the paradigm *Single Instruction, Multiple Data* (SIMD) which is used typically on the GPU, contrary to the CPU which uses *Single Instruction, Single Data* (SISD). Also, the kernel allows sentences like "if-then-else". NVIDIA has developed the concept of *Single Instruction, Multiple Thread* (SIMT). One example of this new concept is executing code depending if the index of a thread is odd or even. However, to get the best inside of a GPU we have to take care of details such as the called *warp*. This means that if each MP contains 8 processors only 32 threads will be executed at the same time, then the developer should be able to consider conflict of memory between the access of the indexes.

2.2.2 Thread Management

Conveniently, the variable **threadIdx** is a 3 component vector that can identify threads by an Unidimensional (1D), Bi-dimensional (2D) or Tree-dimensional (3D) arrangement.

- `threadIdx.x, threadIdx.y y threadIdx.z`

This bunch of threads get grouped into blocks, which can be collapsed by 1D, 2D and 3D index variable **blockIdx**.

- `blockIdx.x , blockIdx.y y blockIdx.z`

This naturally offers a way to invoke elements in cross connection such as vector, matrix or volume. Blocks are organized into a grid, as is shown in Figure 2.2.

There is a limit of threads that the coder can call per block. In actual GPUs the limit is over 1024 threads, this is due to special memory segment shared for all threads inside of the same MP and the architecture of the GPU itself. Nevertheless, a kernel is able to execute a multiple amount of blocks per time. Thus, the total number of threads is equal to the number of threads per block times the number of blocks.

2.2.3 Memory

NVIDIA's GPUs are equipped with 5 different memory, each of one with different characteristics and functionality. Its crucial its understanding in the path of seraching the best

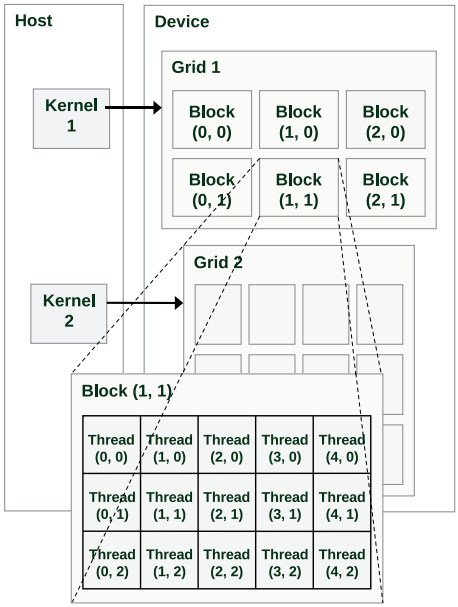

Figure 2.2: Threads organization inside of CUDA architecture.

Memory	Global	Constant	Texture	Shared	Local
Access	W/R	R	R	W/R	W/R
Size	More than 100 MB	64 KB	More than 100 MB	16 KB	More than 100 MB
Scope	Application	Application	Application	Per Block	Per Thread

Table 2.2: Main characteristics of CUDA memory. W/R = Reading and Writing. R = Read only.

performance on the GPGPU. Table 2.2 refers to main characteristics of this memory spaces. Next we will a brief description and usage of this 5 different memory spaces.

Global Memory

This is the main memory zone that a kernel is able to write and read data. Dynamic memory allocation on the fly is not allowed, it must be allocated before the application starts. This memory is variable according to the GPU model and nowadays goes beyond the 1GB. During the kernel call, this memory space is persistent.

Constant Memory

The constant memory is very small, reaching only 16KB and "read" only. This space is persistent along the kernel calls. The host can load any kind of data inside of this space of memory. Read only refers that inside of a kernel this space can not be modified by the device.

Texture Memory

This memory space is quick and read only. Specialized to load mapping and modelling elements in 2D and 3D. CUDA offers to the ability to communicate with graphics pipelines

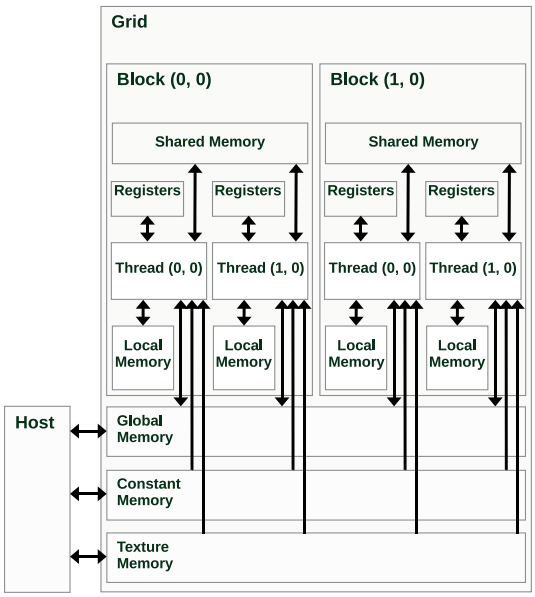

Figure 2.3: Organization of CUDA memory.

such as OpenGL and Direct X in order to save time reaching objects in space memory, thereby, the rendering time is faster.

Shared Memory

The shared memory is small space, about 16KB which does not resides inside of the main memory and it is not persistent along the kernel's call. The host can not load any data, however, when the host call the kernel, this can specify up to 16KB read and write zone for all the threads within a block. In this way, all the threads inside of a block share this memory space. After the last execution of the last thread, this space is unallocated. Using this memory space is faster than the global memory for the same threads within a block.

Local Memory

Local memory has similar functions to global memory, only the life time and the variable scope are limited to one single thread. The main reason is as follows: If every MP can run up to 1024 threads concurrently and only 16384[4] register, each thread can only use 16 of them with full load. If more different variables are need at the same time, these will be allocated in the local memory. Unfortunately, this choice is left for the compiler in order to save registers.

As the Figure 2.3 shows, CUDA architecture has various different types of memory, which can be not so easy so manage but that the developer can used according to his needs. This may impact directly to the performance of the final CUDA application.

[4]The exact number of register of GPU may vary with a different model version.

2.3 CUDA for mobile architectures

With the increase usage of smart phones and tablets, new processors architectures were developed such as ARM architecture, in order to follow the special computing and power demand that these new devices requires for daily task. NVIDIA's company, as graphics accelerator hardware pioneer enter to this market with the introduction of a new branch of mobile processors called Tegra. This system on chip (SoC) is aimed for mobile devices such as smartphones, digital cameras, personal digital assistants and internet mobile devices. Trough many iterations of this new SoC, Tegra APX, Tegra 2, 3 and 4 this new branch of NVIDIA's processor took a place in mobile market. However, all of these chips are not CUDA capable.

It was on April 2014 when NVIDIA finally released one mobile chip capable of CUDA architecture, Tegra K1. This new ARM cortex general purpose 32 bit processor includes a general-purpose GPU. Also is able to run OpenGL ES 3.1, CUDA 6.5 and OpenGL 4.4. The company claims that it outperforms both Xbox 360 and PS3 whilst consuming significantly less power.

Some of the motivations to use this new chip are solutions for compute-intensive embedded projects like autonomous robotic systems, advance driver assistance systems, mobile medical imaging and intelligent video analytics.

3

Mobile Devices

The human has the basic need of communication, and since the technology appears, we haven't stopped searching for better, new, crazy and innovative ways of transmit information between each other. Commodity is also an strong point when we think about technology development, thus mobility. Maybe one of the most important events on the development of the human communication history is the invention of the telephone which allow the globe to become a vast network of electricity-based speaking people. With the arrived of transistor and micro electronic components, the compute devices started to become smaller and smaller, and with the integration of the radio wave technology, the first clues of mobile devices start to appear. Changing the way we communicate, where and how, the mobile phone is maybe one of the most important devices nowadays, which changes and creates industries and businesses. Not only the usage of mobile phones but also the Personal Digital Assistant (PDA), tablets, music players, watches, video game portable devices, and GPS are now part of our daily life, and sometimes essentials.

3.1 First mobile devices an its capabilities

The radio telephony used in the second world war is the predecessor of the existing mobile phones. At the same time, telephones on auto-mobiles were possible however not so famous and common to use. AT&T company was the first in commercialized a Mobile Service Telephone in 1949. In 1973 Martin Cooper from Motorola introduce the first mobile called "the brick" [28] as is shown[1] in Figure 3.1. It was capable of 60 minutes of talking however taking 10 hours to get full charge.

Another mobile device, which impact over the society and the way we live was the Sony's Walkman. In 1979 the first portable cassette tape was on sale for $150 dollars [29]. Power consumption was low requiring only either one AA battery or one gumstick-type rechargeable.

[1] Images courtesy of "2007Computex e21-MartinCooper" by 2007Computex-e21Forum-MartinCooper.jpg

Figure 3.1: Martin Cooper photographed in 2007 with his 1973 hand-held mobile phone prototype.

Figure 3.2: Palm model TX.

This small device, change a way in which used to enjyo usic in a travel, on the public transport and also when we do fitness activities.

The introduction of PDAs or personal data assistant were the first steps of what we know common today as a tablet or handheld PC. the calle dPDA has an electronic visual display, including a web browser, capable of playing audio as a portable media player. Most of PDAs have access to internet, intra-nets or extra-nets via Wi-Fi and most importantly utilizes touch technology. One example is the Palm TX which is shown[2] in Figure 3.3.

Perhaps another big mobile which made a huge impact in the way of its application is the Nintendo GameBoy. Released in Japan in 1989, this is one of the first mobile consoles to has a tremendous success. Holding the purpose of playing video games only, this 15-30 hours portable game console was powered by 4 AA batteries. It was included in the system a CPU custom 8 bit processor at 4.19 MHz.

As we can denote from the devices mentioned before, they were the first on its category, moving forward the technology and evolving the way of communication. Strongly, we can observe that the functionality was limited to one main function, the battery life was mention

[2]Images courtesy of Palm T—X Photographer: Stefano Palazzo Used Camera: Kodak Z740

Figure 3.3: Various Post PC devices.

to have a period of hours and the interactivity with the user was tied.

3.2 Post PC devices

During the end of the 90's and the first half of new millennium, the cellular phone was consider the gadget of the moment. Accompanied with the first PDAs and the popularity of Internet, this mobile devices were doted all ready with the first mobile OS, such as Palm OS, Black Berry OS, Windows CE/Pcoket PC or Simbian. These small apparatus, as the ones shown[3] in the Figure 3.3, were developed to deliver more than one main function to the user, making them more intuitively and dynamic, thus more complex in design and powerful. But it wasn't until 2007 when Apple introduce the iPhone that the real-smart phone, one of the most important Post PC devices, came along to the scenario to define a new way to interact with mobile computers. Since then, multiple devices such as Tablets, E-Readers, smart-watches and others, make usage of new technologies to increase the capabilities, functionality, computation performance, interactivity and services which provides a unique experience to the users.

3.2.1 Main Capabilities

Possibly, the greatest impact of the Post PC devices over the first mobile devices are the bunch of capabilities inside of them. In the beginning, mobility was only focused on one main task due to the technology frontier and scarce resources. However, nowadays the experience is much more complete within a tablet or smart phone. Some of these new features are:

- Mobility.
- Connectivity.

[3]Images Courtesy of Courtesy of www.isanweb.es

- Limited Memory.
- Portability.
- Huge Ecosystem.
- Low power consumption.
- Touch screen capabilities
- Low processing (ARM processors)

With these various capabilities, the applications for a mobile devices are more wide rich in experience and contents. The search for crating new contents has lead a numerous variety of research topics such as virtual reality, visualization data, health based applications, games and others [9] [10] [11] [12] & [13]. As well as this dissertation will approach the high performance computing on mobile devices through remote GPGPU.

3.3 Android Ecosystem

Although, recently there are more than one operating system for mobile devices such as iOS for apple products, Windows mobile, WebOS and Ubuntu touch, the author decided to use Android due to its widely adoption through the mobile device community, Linux based architecture, more compatibility through wide range of devices and its ease of application implementation. Android is was developed by Android Inc. which was absorbed by Google in 2005. Since then Android has become one the most emblematic products of this company. Android since its first appearance has many iterations versions which are named with "desserts", such as 1.6 Donut, 2.0 Eclair, 2.2 Froyo, 2.3 Gingerbread, 3.0 Honeycomb, 4.0 Ice cream sandwich, 4.1 Jelly bean, 4.4 Kitkat and finally the most recent 5.0 Lollipop.

3.3.1 Programming model

Android has a Linux kernel on its core, and on the top of this there are the middleware, libraries and APIs most based on C language. Also, this mobile OS contain an application framework which includes Java-Compatible bytecode. The Dalvik virtual machine is in charge of generate dex-code which is translated directly from Java bytecode. This virtual machine is optimized to use few memory which is one of the "commons" inside of mobile devices. It is designed to execute various instances. In the Figure 3.4 a general view of the programming model[4] is displayed.

[4]Image courtesy of "Android-System-Architecture" by Smieh - Anatomy Physiology of an Android.

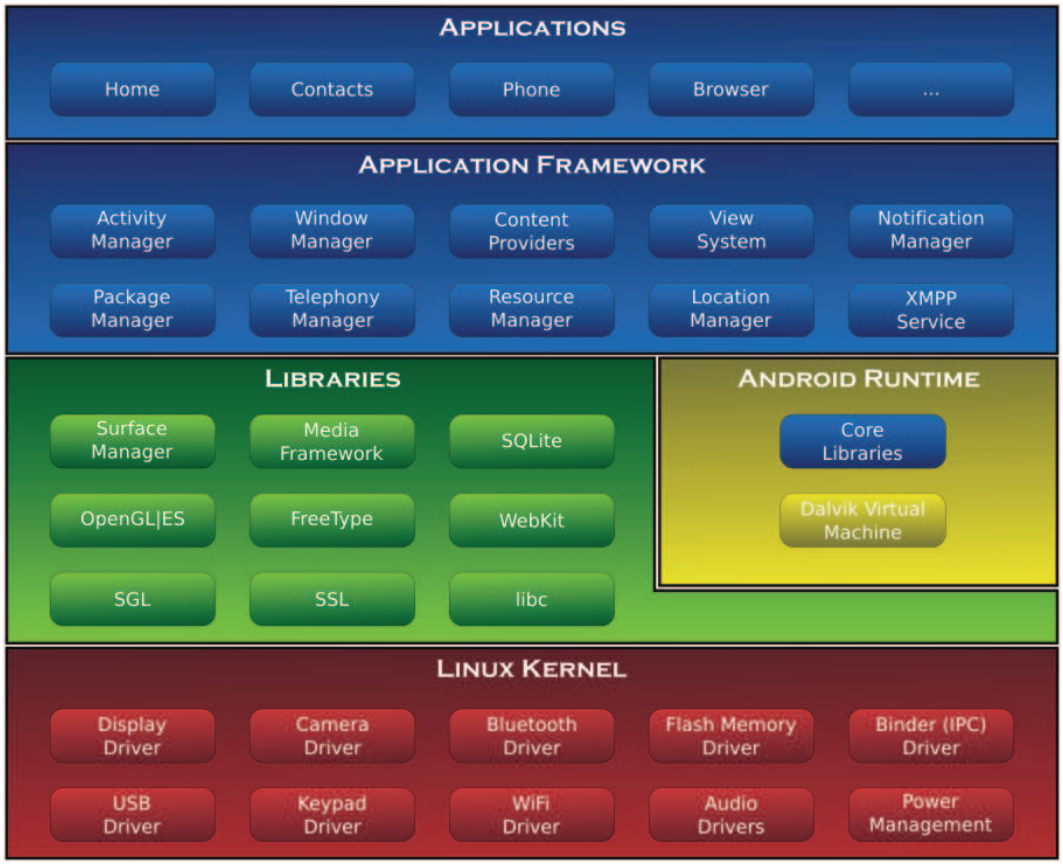

Figure 3.4: Android programming model architecture.

3.3.2 OpenGL ES

OpenGL is the one the most widely known computer graphics rendering plication programming interface developed by Krhonos Group. OpenGL ES for embedded systems supports the generation of 2D and 3D graphics for smart-phones, tablets, video-game consoles and PDAs. Ti has been released various major versions such as 1.0, 1.1, 2.0 and 3.0. The similarities between the normal OpenGL API are perceptible but not identical. One of the major disadvantages for porting between these two API is that GLUT [5] is not included. Also OpenGL ES comes with its own shading language. Android supports OpenGL ES since the Android 1.6 version on its different major versions.

3.3.3 Native Development Kit

The called NDK toolkit allows running C and C++ code inside of Android device. There are 2 good reasons why is practical to use C instead of Java language. First, the execution of C code is done natively inside of the ARM processor, intended for such applications with CPU-intensive workload such as game engines, signal processing, physics simulation and so on. Second, to use libraries which are already exist in C language code and also the usage of

[5] OpenGL Utility Toolkit facilitate the window managing, keyboard and mouse input functions and others.

third party code. NDK set of tool-chains that can generate native ARM binaries on Linux, OS X and windows platforms. It also provides a set of system headers for stable APIs:

- libc.
- limm.
- JNI interfaces.
- libz for compression.
- liblog for Android logging.
- OpenGL ES 1.1 and 2.0.
- libjnigraphics.
- Minimal set of C++.
- OpenSL ES for native audio.
- APIS for Android native applications.

Basically NDK uses a set of ARM compilers to create static and shared libraries which are loaded by the Dalvik virtual machine. This libraries can be created by specifying correctly the *Android.mk* and *Application.mk* files. Under the folder called "jni" inside of a Android project, these two files are similar to "make" based type archives. Inside of them we can specify paths, compiler flags, directives, libraries and other C like options. Finally, a dynamic library is created which is loaded in Java code through *System.Load()* function. NDK also provides a set of sample programs (SDK) which describes how to create a shared library and included inside of your Java Code.

4

DS-CUDA

Super computers and clusters inside of the High Performance Computing (HPC) field are nowadays equipped with several hundreds of thousand cores. Due to this large topology of CPU connections, communication among the cores tends to be the bottle neck rather than the computation itself. Moreover, as we explained in chapter 2, with the arising of GPGPU and its good welcoming from CUDA's hand, more and more scientific community integrated this devices to its HPC cluster as accelerators. Some of the top machines in the TOP500 [21] list are also equipped with GPUs. However, a limiting point is the ability to allocate a few GPUs inside of each PC. Therefore, the scenario to implement an algorithm inside of this massively parallel system may be that the developer applies at least three common HPC frameworks: MPI,OpenMP and CUDA. Even a complicated program using all three frameworks may fail to take full advantage of the system performance. In addition, in some cases we have a machine which is not equipped with a GPU, and using them from our local network may benefit educational and demonstrative purposes.

To solve this kind of difficulties, HPC frameworks have arisen such as Distributed-Shared CUDA (DS-CUDA).

4.1 Overview

DS-CUDA is a middleware that allows to manage NVIDIA's GPUs on a distributed network. A single client node and various server nodes compose one DS-CUDA system, as is 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 these parallel devices to process data without having a physical GPU. The program sees all GPUs contained into a cluster as if they were actually attached to the client node. Therefore, DS-CUDA is a kind of GPU-virtualization tool at source code level.

When the client calls native CUDA API, the DS-CUDA preprocessor handles the correct

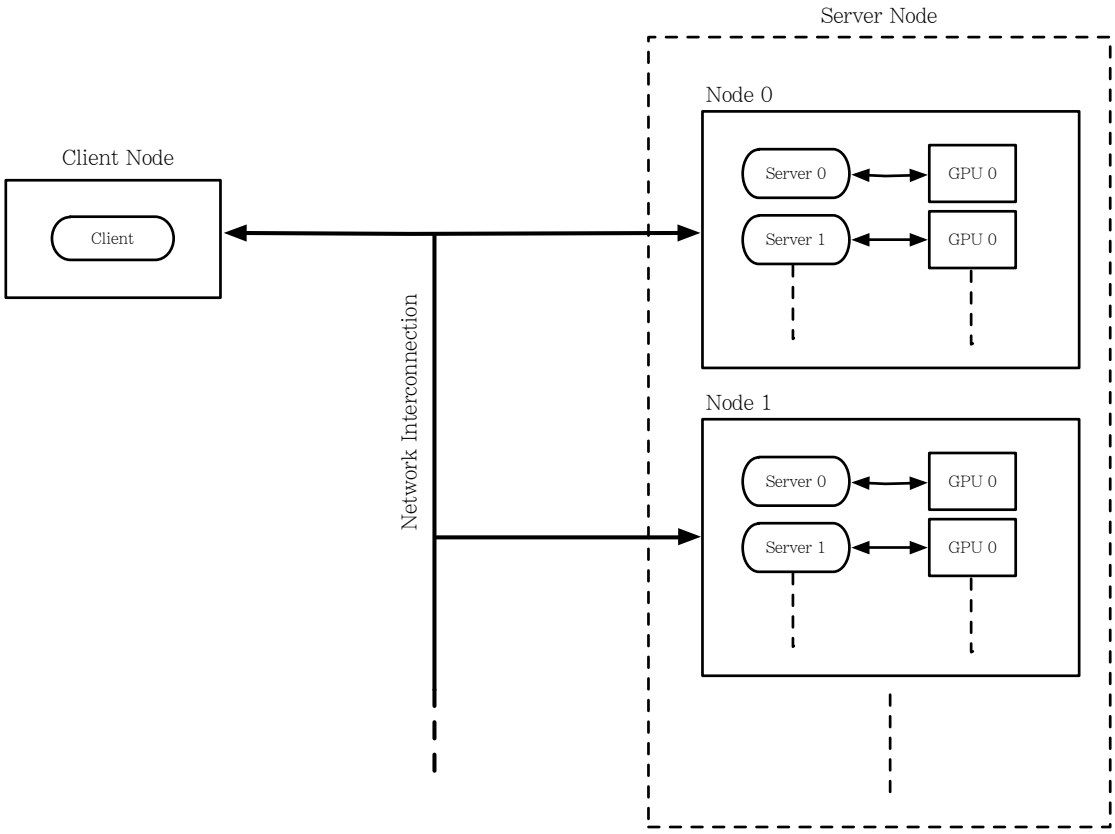

Figure 4.1: Prospect of a typical DS-CUDA system.

wrapper function, which communicates with the server nodes through an InfiniBand (IB-Verb) or TCP socket. The wrapper function sends the proper arguments and data to the server nodes, each of which process this data calling the actual native CUDA API. A detailed description of its implementation is in another paper [22] and also application [23].

4.2 Package Description

DS-CUDA is presented as an Open Source package that can be obtained from http://narumi.cs.uec.ac.jp/dscuda/ as a .tar package. Contents of this package are listed in Figure 4.2. As the image shows, the contents are all binaries including the *dscudacpp* which is a preprocessor who handles the kernels calls inside of the client side. A brief documentation is also included, such as a quick start.However, more detailed explanation is a viable in Japanese at the moment. Some examples are also part of DS-CUDA package, including bandwidth program which measure performance of *cudaMemcpy* function[1]. Claret sample which shows a molecular dynamics simulation between Na Cl particles. Direct sample which copy memory between GPU's using peer to peer function. Matrix multiplication program and the last one which shows a basic computation of adding two numbers such as $a + b = c$. Nevertheless,

[1]This function copy memory between the host (CPU) and the device (GPU).

```
1. Default (ssh)
edgar@Edgar-PC:/usr/local/DSCUDA/DSCUDAlegacy/DSCUDA/dscudapkg1.3.2$ tree -d
.
|-- bin
|-- doc
|-- include
|-- lib
|-- sample
|   |-- bandwidth
|   |   `-- dscudatmp
|   |-- claret
|   |   |-- dscudatmp
|   |   `-- table
|   |-- direct
|   |-- matrixMul
|   |   |-- dscudatmp
|   |   `-- tmp
|   `-- vecadd
|       `-- dscudatmp
|-- script
`-- src
```

Figure 4.2: Contents of DS-CUDA package.

DS-CUDA framework continues in development and it's keeping adding more contents and capabilities which may be different as the ones mentioned here.

4.3 Usage

As we were mentioned before, DS-CUDA was born as a high performance computing framework for NVIDIA's GPUs. This means, that clusters and supercomputers were the main target of this new technology. One more real example is the one shown in Figure 4.3. In this system configuration we can appreciate one client machine without any physical GPU. Also, various server nodes with up to 2 GPU per node.

DS-CUDA runs mainly over Linux based operating systems. Main developers and testers have used distributions such as Knoppix 7.2, Ubuntu 12.04, Ubuntu 14.04 and Fedora 12 to test the package. DS-CUDA works properly with x86 32-bit and 64 bit architectures. One important thing to consider is the communication medium between client and server, such as Inifni-Band verbs or TCP socket. For the first one, you may need the proper libraries and also the device and infrastructure to interconnect your machines. To use TCP socket, Ethernet cable or Wi-fi is used, which is the most common option to test DS-CUDA.

4.3.1 Installation

In order to run DS-CUDA, make sure that your machine includes the following packages and tools:

- CUDA Toolkit & SDK (Tested with 4.1 5.5 and 6.0)

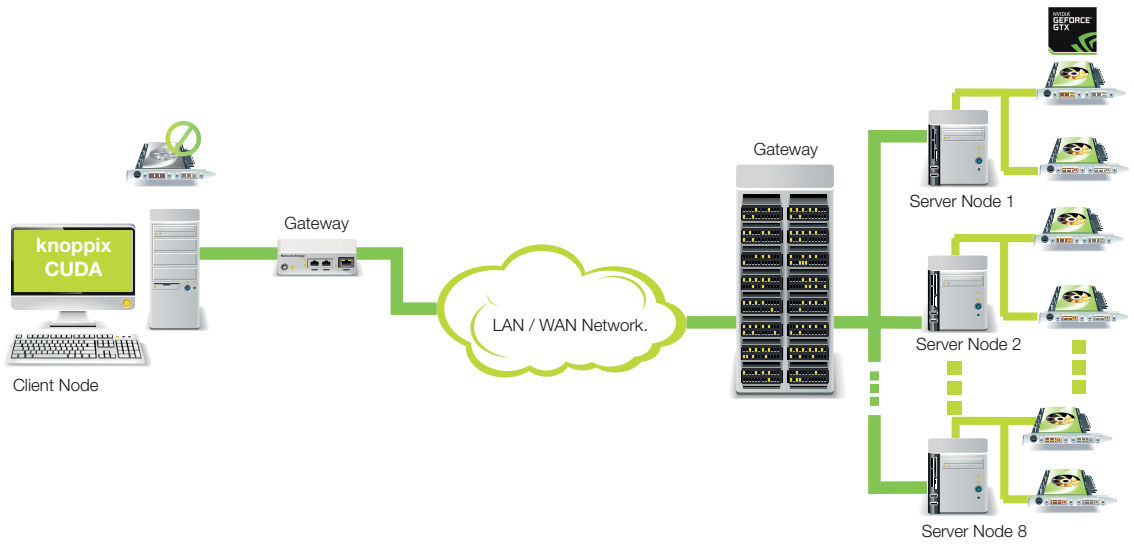

Figure 4.3: Example of a DS-CUDA system inside of Narumi's lab.

- GNU C++ compiler (Tested with version 4.5 and above)
- Ruby (Tested with version 1.8 and above)
- OFED (Tested with version 1.5) – for Infini-Band support.

Once the package is downloaded and extracted, go directly to **$dscudapath/src**. Inside of this directory all the source code is included and through the *make* command you can generate binaries and libraries for client and server, such as :

- $dscudapath/bin/dscudasvr →DS-CUDA server executable.
- $dscudapath/lib/libdscuda_rpc.a →DS-CUDA client library (TCP socket).
- $dscudapath/lib/libdscuda_ibv.a →DS-CUDA client library (IB verb).
- $dscudapath/lib/libcudart.so →Dummy CUDA runtime library.

4.3.2 Configuration

The program running inside of a DS-CUDA client sees virtual devices that are real devices allocated in the server. The mapping of the real virtual devices is given by an environment variable[2] `DSCUDA_SERVER`. DS-CUDA framework manages and configures through environmental variables. The ones of primary usage are listed bellow:

- `DSCUDA_PATH` :Indicates the full path of the package location

[2]Environmental variables provide a way to influence software behaviour on Linux system. They are widely used for customization and configuration.

```
dscudasvr[0] : WarnLevel: 2
dscudasvr[0] : method of remote procedure call: SOCK
dscudasvr[0] : TCP port : 65433 (base + 0)
dscudasvr[0] : ndevice : 1
dscudasvr[0] : real device       : 0
dscudasvr[0] : virtual device    : 0
dscudasvr[0] : listening on port 65433.
```

Figure 4.4: Correct execution output from a DS-CUDA server.

- `DSCUDA_WARNLEVEL` :Configures the warning level. 2 is the default value, but can be increased to 5 for debugging purposes.
- `DSCUDA_REMOTECALL` :Indicates which kind of protocol is used between client and server.
- `DSCUDA_USEDAEMON` :Selects the option to run a daemon which stay listening for client request
- `DSCUDA_SERVER` :Configures the server host-name or "ip" address.

A complete list of DS-CUDA environmental variables refer to Japanese documentation.

4.3.3 Sample Test

In this section we describe a real example of DS-CUDA system test as the one depicted on Figure 4.3, using the *vecadd* sample listed in the samples of the package.

Server Side

Once inside of the main path **$dscudapath/src**, we can create the binary for the server. In this scenario using socket protocol connection we compile *dscudasvr*. Once generated, we need to set some minimal environmental variables such as: `DSCUDA_WARNLEVEL = 2` and `DSCUDA_REMOTECALL = tcp`. Done the mentioned above, we can lunch the program, obtaining an output similar to the Figure 4.4.

On the output of the DS-CUDA server we can observe some useful details: warning level execution, the communication protocol between server and client, number of real devices and the listening port.

Client side

For client side, we have to perform two actions: 1. Generate a proper DS-CUDA library supporting our connection protocol. 2. Compile our CUDA code using the DS-CUDA preprocessor and generate the executable.

For the first stage, we need to generate *libdscuda_rpc.a* which is inside of the main path **$dscudapath/src**. Then we need to compile our CUDA source code with the preproces-

```
WarnLevel: 2
searching DSCUDA servers...
recvfrom found server: "192.168.0.205"
method of remote procedure call: TCP Socket
automatic data recovery: off
waiting for the server to be set up...
Established a socket connection to 192.168.0.205:0 (port 65433) ...
Client IP address : 192.168.0.205
try 0
/usr/local/DSCUDA/dscudapkg1.7.5.1/bin/ptx2symbol:25: warning: assigned but unused variable - args
/usr/local/DSCUDA/dscudapkg1.7.5.1/bin/ptx2symbol:28: warning: assigned but unused variable - ts
/usr/local/DSCUDA/dscudapkg1.7.5.1/lib/utility.rb:17: warning: assigned but unused variable - verbose_default
/usr/local/DSCUDA/dscudapkg1.7.5.1/lib/utility.rb:141: warning: assigned but unused variable - elsep
/usr/local/DSCUDA/dscudapkg1.7.5.1/lib/utility.rb:142: warning: assigned but unused variable - ersep
 39.00 +   6.00 =   45.00
 41.00 +  51.00 =   92.00
 17.00 +  63.00 =   80.00
 10.00 +  44.00 =   54.00
 41.00 +  13.00 =   54.00
 58.00 +  43.00 =  101.00
 50.00 +  59.00 =  109.00
 35.00 +   6.00 =   41.00

try 1
 60.00 +   2.00 =   62.00
 20.00 +  56.00 =   76.00
 27.00 +  40.00 =   67.00
 39.00 +  13.00 =   52.00
 54.00 +  26.00 =   80.00
 46.00 +  35.00 =   81.00
 51.00 +  31.00 =   82.00
  9.00 +  26.00 =   35.00

try 2
 38.00 +  50.00 =   88.00
 13.00 +  55.00 =   68.00
 49.00 +  24.00 =   73.00
 35.00 +  26.00 =   61.00
 37.00 +  29.00 =   66.00
  5.00 +  23.00 =   28.00
 24.00 +  41.00 =   65.00
 30.00 +  20.00 =   50.00
```

Figure 4.5: Correct execution output from a DS-CUDA client.

sor *dscudacpp*, inside of the **$dscudapath/bin** path. This will generate our executable which GPU code will run over the server side. Before the executable may be launched, we need to specify some DS-CUDA environment variables as well:`DSCUDA_WARNLEVEL = 2`, `DSCUDA_REMOTECALL = tcp` and `DSCUDA_SERVER = 192.168.0.205`.

As we can observe in Figure 4.5, the GPU code is executed in one server machine addressed by the ip number 192.168.0.205, which is equipped with one GPU.

4.4 Development Contributions for DS-CUDA

In order to bring the power of GPGPU to mobile devices, specifically over Android platform, DS-CUDA was used as a medium to execute use remote GPU. We were able to adapt DS-CUDA framework to run in Android through 2 methods: 1. Using a terminal emulator and external ARM compiler. 2. Using NDK. As well, we develop a make file system based to maintain the SDK provided in DS-CUDA package. Also we provide a GitHub[3] support. Next sections we explain in detail these contributions.

[3]Web based git repository hosting service.

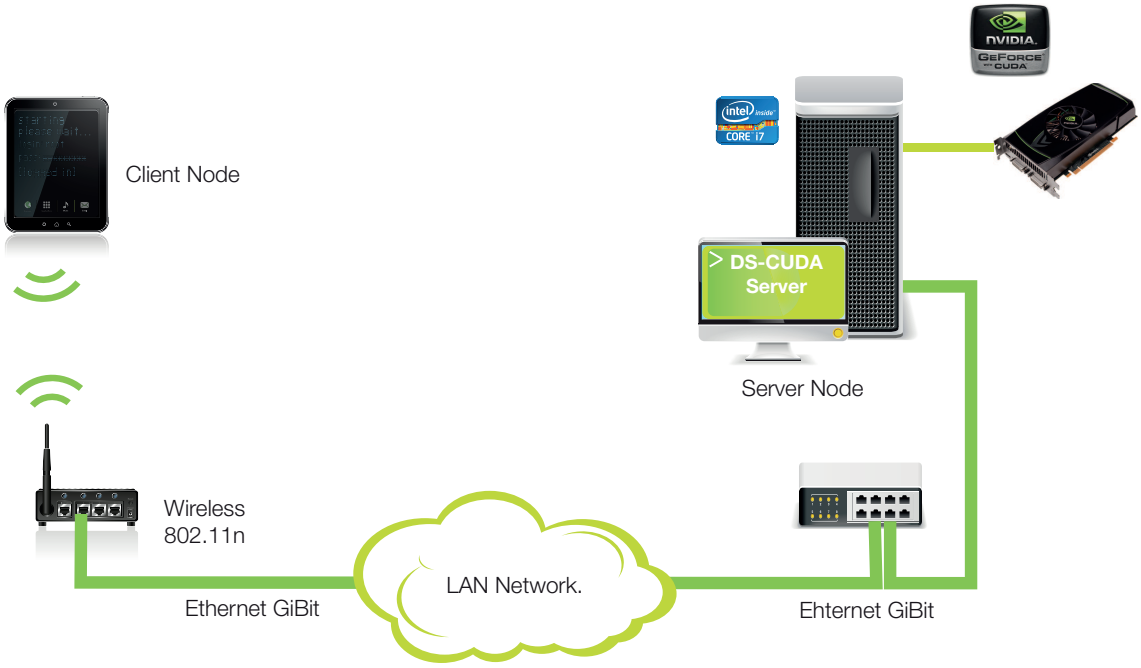

Figure 4.6: System prototype to use DS-CUDA on Tablets.

4.4.1 Enabling Android Tablets

One of the motivations of this research was the idea of merging the world of mobility with high performance computing. In order to create new applications which the author depicted as the first idea when the whole process started, such as in Figures 1.1 & 1.2.

The purpose was connect the interactiveness and mobility capabilities of the tablet with applications that require exhaustive computation. A example system is shown in Figure 4.6. One tablet device connected to a local network through Wi-Fi and a DS-CUDA server equipped with one CUDA capable GPU interconnected to the same local network via Ethernet cable.

Through terminal emulator

The first approach was using the terminal emulator. Inside of the huge Android app ecosystem, there is a terminal emulator shell or bash like. Inside of this emulator you can execute common Linux commands such as *ifconfig*, *./* and *cd* etc. Then, the basic idea was to compile a basic example from DS-CUDA sample package using an external ARM compiler. Copy the binary to the device and execute in terminal as a normal Linux based program. The full steps are detailed next:

First

The scenario is depicted in figure 4.7. The *sample.cu* file contain the proper CUDA code to add some numbers. This file is inserted to *dscudaccp* preprocessor. The output is composed by several files: The *sample.ptx* correspond to low level like code inside of

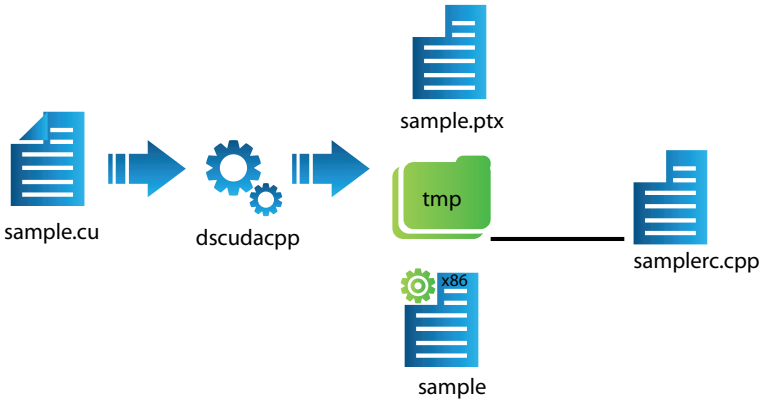

Figure 4.7: Output of dscudacpp preprocessor.

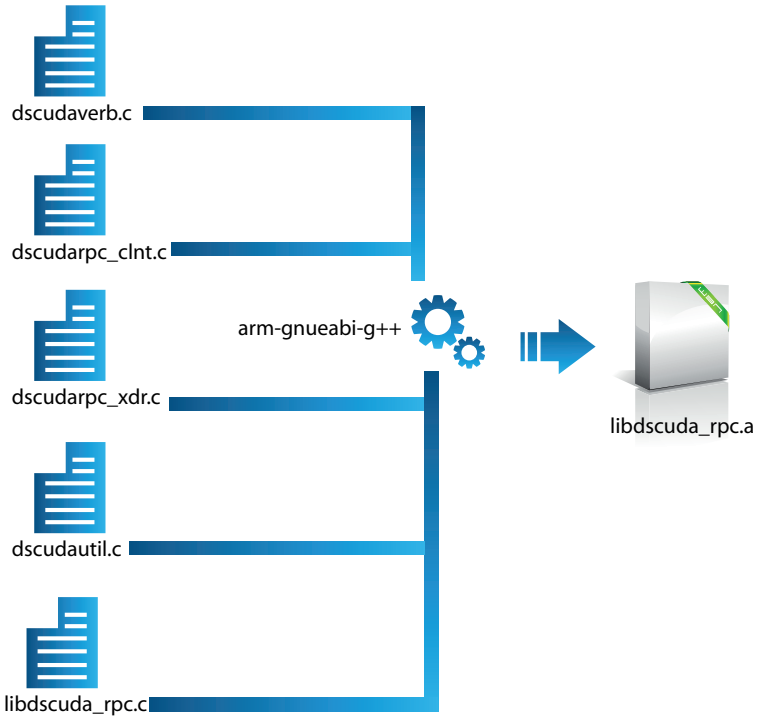

Figure 4.8: Creation of client static library for ARM architecture.

the kernel wrriten in sample.cu. This is generated by nvcc compiler. The *samplerc.cpp* which is a similar version of the original code but wrapping all the native CUDA functions with the DS-CUDA proper ones. Last, the sample binary which is the final executable for desktop machines.

Second

In order to be able to use the APIs from DS-CUDA on client side on the tablet, we need to generate *libdscuda_rpc.a*. This library includes all the wrapper functions to communicate with the server and execute the CUDA code. We used an ARM compiler provided by ARM GNU/Linux tool chain to generate object files and then collapse these into the static library, as Figure 4.8

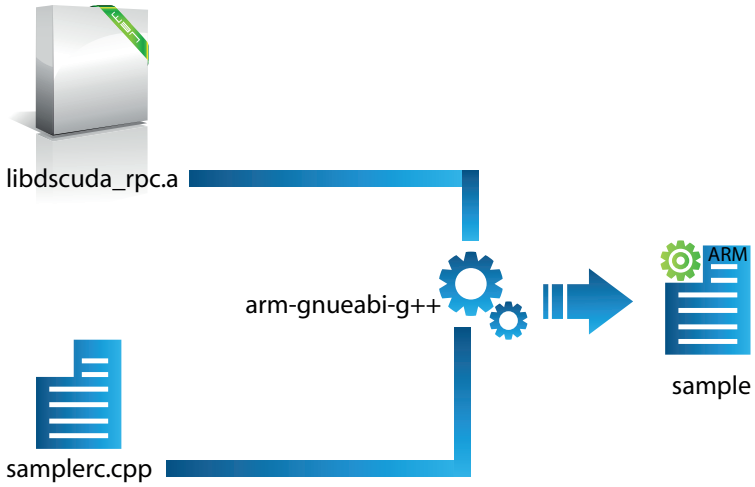

Figure 4.9: Creation DS-CUDA executable for ARM architecture.

Figure 4.10: Copy DS-CUDA executable to tablet.

Third

The samplerc.cpp code is compiled with the ARM compiler and it is integrated with *libdscuda_rcp.a* to create a final executable, Figure 4.9.

Fourth

Finally, two files are copied manually (Via SD card or USB) into the tablet, Figure 4.10. We access this path under the terminal emulator and execute the binary.

The executable will send a copy of the .ptx to the server, which includes the CUDA code for the GPU. A final output from the tablet is pictured in Figure 4.11.

DS-CUDA was enabled by this method, and also the first trials of CUDA over Tablet devices. However, there are some week points about it: Due to usage of terminal emulator, displaying data is restricted only to text.If we require to implement an OpenGL app another way should be found. The process for compiling and testing is complicated, thus become impractically to develop. Copying and pasting the executable and .pxt code generate may damage the SD card and USB bay due to connect and disconnect action. At this point, DS-CUDA version 1.3.2 was using the Remote Procedure Call (RPC) library to interconnect the

Figure 4.11: DS-CUDA executed on Android terminal emulator.

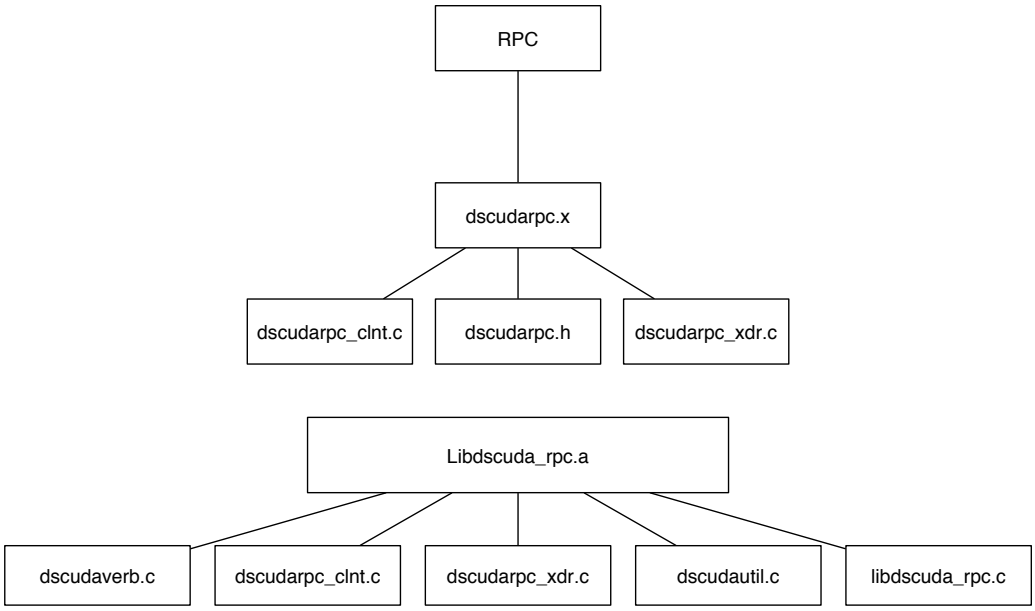

Figure 4.12: Constitution of DS-CUDA client library version 1.3.2. RPC based.

client and server over Ethernet protocol using sockets. With this method, none modifications were made to compile libdscuda.a library. Nevertheless, NDK tool-kit is not capable of using such libraries.

4.4.2 Using Native Development Kit for Android

As we saw in the section above, running DS-CUDA using the terminal emulator allow us to use RPC libraries but limiting to develop shell console based application. One alternative to solve this issue is using NDK which was detailed in 3. The Native Development kit allow to use C code inside of the Java main based program on Android devices. The first trial was to compile the *libdscuda_rpc.a* static library within NDK, however as the contents inside of the package shows, there is no support for RPC libraries inside of NDK. The client library for DS-CUDA version 1.3.2 uses RPC to implement the wrapper functions corresponding to the native CUDA API. A diagram of the contest are shown in Figure 4.12. The *dscudarpc.x* prototypes all the wrapper functions on RPC language, then using *rpcgen* generate the proper stubs for client, server, external data representation and headers in C language. The problem to compile these files corresponding to RPC, such as *dscudarpc_clnt.c* is that NDK do not has some prototype values.

A new version of DS-CUDA is proposed, in which POSIX sockets were used instead of RPC library. The POSIX socket API is a viable inside of NDK enabling the communication with the external apps directly without calling into the Java layer. A socket is a connection end-point that can be named and addressed in order to transmit data between applications that are running either on the same machine or another machine on the network. Thus, the

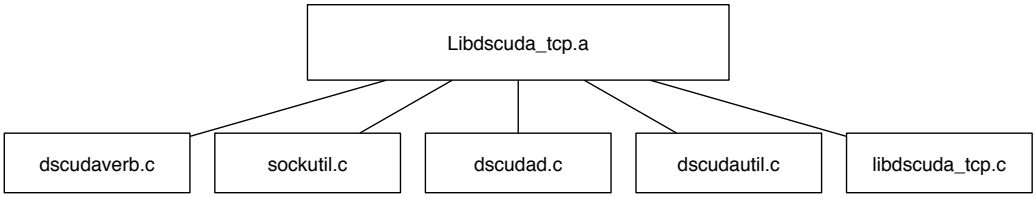

Figure 4.13: Constitution of DS-CUDA client library version 1.5.2. TCP socket based.

new version of client library for DS-CUDA 1.5.2 is shown in Figure 4.13.

The *sockutil.c* file implements the set up connection between client and server, the received and sent message functions only using POSIX sockets. The prototype functions are shown bellow:

```
1 struct sockaddr_in setupSockaddr(unsigned int ipaddr, int ipport);
2 void sendMsgBySocket(int sock, char *msg);
3 void recvMsgBySocket(int sock, char *msg, int msgbufsize);
```

The source code named *libdscuda_tcp.c* implements the remote call which carries the wrapper function to call native CUDA calls and send the proper data to the server side. Basically the implementation of the wrapper functions are the same, only the medium which are carried by is changed from RPC to TCP socket. The next step is to include the DS-CUDA client static library inside of the Java code. Initially, the first test included all the source code of the *libdscuda_tcp.a* inside of the main project, lately we realized that every time we want to use DS-CUDA on the tablet we have to copy all the code. Then, we created a separated project in which only *libdscuda_tcp.a* is generated. The sample of the Android.mk and Application.mk are showed next:

```
1 ## Android.mk
2 ##################Static Library libdscuda_tcp.a
3 LOCAL_PATH := $(call my-dir)
4 include $(CLEAR_VARS)
5
6 LOCAL_MODULE    := dscuda_tcp1.5.2
7
8 LOCAL_CFLAGS := -O0 -g -ffast-math -funroll-loops -I. \
9 -I/usr/local/cuda/include \
10 -I/usr/local/cuda-4.1/NVIDIA_GPU_Computing_SDK/C/common/inc \
11 -I/usr/local/cuda/samples/common/inc -DTCP_ONLY=1
12 LOCAL_SRC_FILES :=  dscudaverb.cpp dscudautil.cpp \
13 sockutil.c libdscuda_tcp.cpp \
```

```
14 LOCAL_LDLIBS := -ldl -llog
15 include $(BUILD_STATIC_LIBRARY)
16 ##################Static Library DS-CUDA Routine
```

As we can observe, all the source code inside of Figure 4.13 is included inside of the Android.mk. Some flags are also included, such as debugging, fast mathematics and unroll. Some local libraries from the NDK are also called such as the log library.

```
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
```

The Application.mk file configures the platform, type of C library to load, architecture and some exceptions for the compiler to use at the time to generate the code. To include the *libdscuda_tcp.a* in another project just use the *include* command inside of the Android.mk file, writing the full path where the static library is located. Include the project as *BUILD_STATIC_LIBRARY* inside just above the compilation line where the main preprocessed CUDA code is located. As a sample, we show a bandwidth sample program which measures the memory transfer between the tablet and the remote GPU. Figure 4.14 shows the output of this app.

The entire Android.mk code for the bandwidth app is showed below:

```
1 ##################Static Library libdscuda_tcp.a
2 LIB_LOCAL_PATH := $(call my-dir)
3 include /usr/local/DSCUDA/dscudapkg1.5.2/src/Android/jni/Android.mk
4 LOCAL_PATH := $(LIB_LOCAL_PATH)
5 ##################Static Library DS-CUDA Routine
6 include $(CLEAR_VARS)
7
8 LOCAL_MODULE     := bandwidth
9
10 LOCAL_CFLAGS := -O0 -g -P -ffast-math -funroll-loops -fpermissive -I. \
11 -I/usr/local/cuda/include \
12 -I/usr/local/DSCUDA/dscudapkg1.5.2/include/common/inc \
13 -I/usr/local/cuda/samples/common/inc -DTCP_ONLY=1
14
```

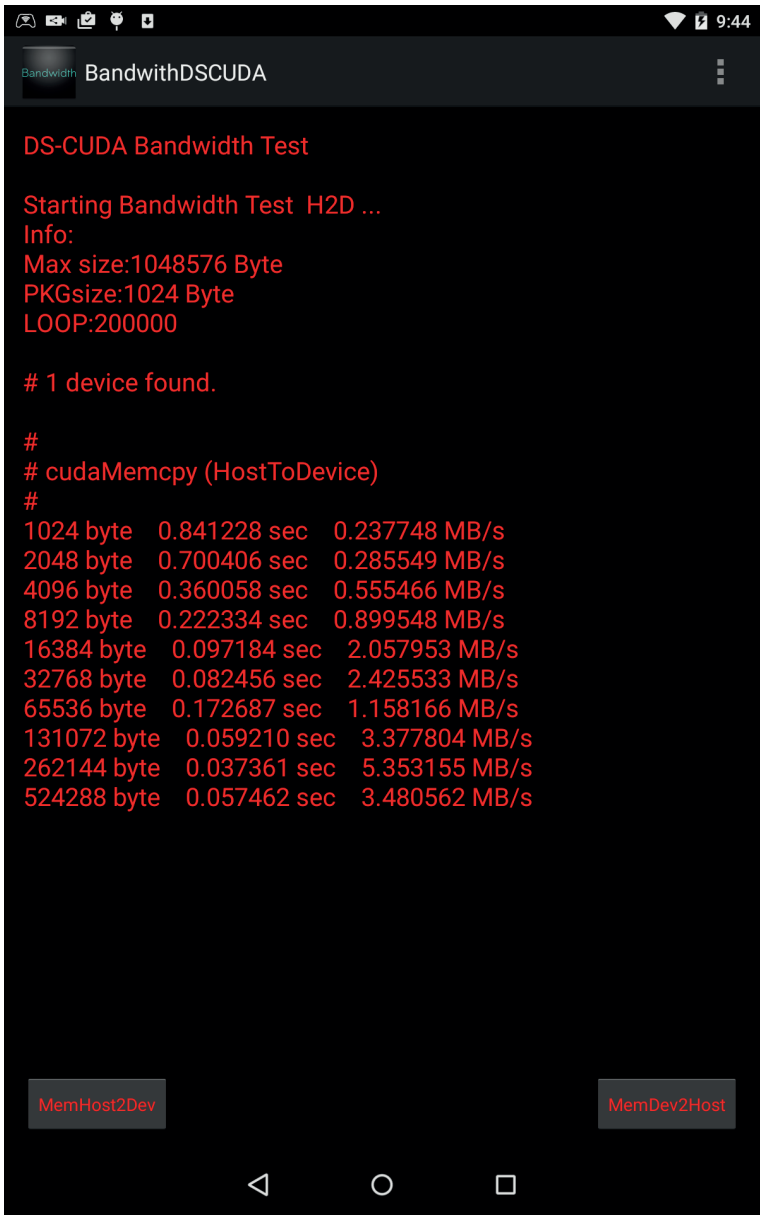

Figure 4.14: Bandwidth sample output. Tablet performing memory transfer to the remote GPU using DS-CUDA.

```
15 LOCAL_SRC_FILES :=  bandwidth.cpp
16 LOCAL_STATIC_LIBRARIES := dscuda_tcp1.5.2
17 LOCAL_LDLIBS := -ldl -llog
18 include $(BUILD_STATIC_LIBRARY)
19 ################Compile Claret main program
20 include $(CLEAR_VARS)
21
22 LOCAL_MODULE := bandwidthtest
23
24 LOCAL_CFLAGS := -g -W -DANDROID_NDK -DDISABLE_IMPORTGL
25 LOCAL_SRC_FILES := app-android.cpp
26
27 LOCAL_STATIC_LIBRARIES := bandwidth
28 LOCAL_LDLIBS := -ldl -llog
29
30 include $(BUILD_SHARED_LIBRARY)
```

4.4.3 Makefile script for DS-CUDA SDK

DS-CUDA package provides some samples for the user to test and learn how to use this framework. Various examples, as in Figure 4.2 is shown, such as memory transfer performance, broadcast memory through peer to peer, matrix multiplication and add simple numbers helps the developer to realize how to compile and apply their own CUDA code over DS-CUDA. However, some times the paths, includes and other issues such architecture of the system e.g. 32 bit or 64 bit make a major difficulty to test right away. The main idea is to create a makefile script which can help to automate the whole generation of DS-CUDA SDK process, configuring according to the user scenario. The basic capabilities of this script are:

- Search over all directories and generate the proper sample.
- Select between TCP socket library or IB verbs (if a viable in the system).
- Generate all binaries with normalized names.
- Distinguish between Linux OS 64/32 bits.
- Set the paths for DS-CUDA.
- Locate CUDA distribution and paths.

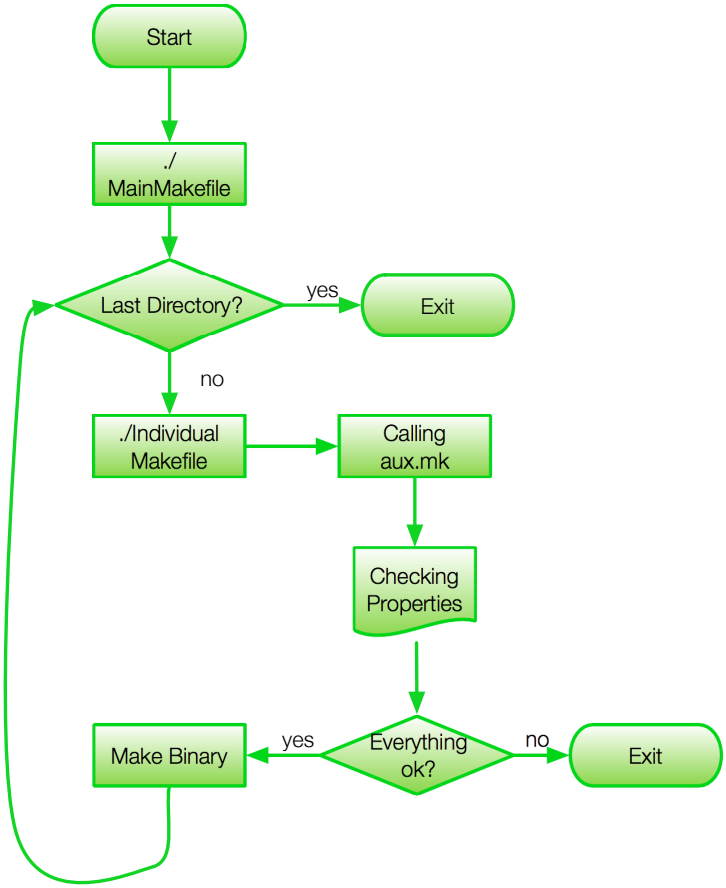

Figure 4.15: Makefile script algorithm for DS-CUDA SDK.

The main algorithm is depicted in Figure 4.15. A main makefile is located inside of the top path of DS-CUDA. One auxiliary file is located inside of the script folder, which is called by all the individual make files inside of the SDK samples.

The Figure 4.16 prints out all the binaries generated with our script. We can observe that all the names are normalized with the name *_rpc* because they were generated under this constraint. If we choose IB verb protocol, a suffix *_ibv* will concatenate the binary name.

4.4.4 Github repository for DS-CUDA

DS-CUDA package can be found in http://narumi.cs.uec.ac.jp/dscuda/ as a tar package. Inside of the package the last version of DS-CUDA and samples can be found. Of course this is common way to distribute our technology with other people, in order to make them know the latest enhancements of our development. Nevertheless, the usage of hosting web services have become widely popular among developers. Using such kind of tools in the cloud bring us some useful capabilities to track and keep the development of DS-CUDA. We propose to use GitHub hosting web provider since *git*[4] is very popular among the Linux OS,

[4]Distributed revision control created by Linus Torvals for Linux kernel.

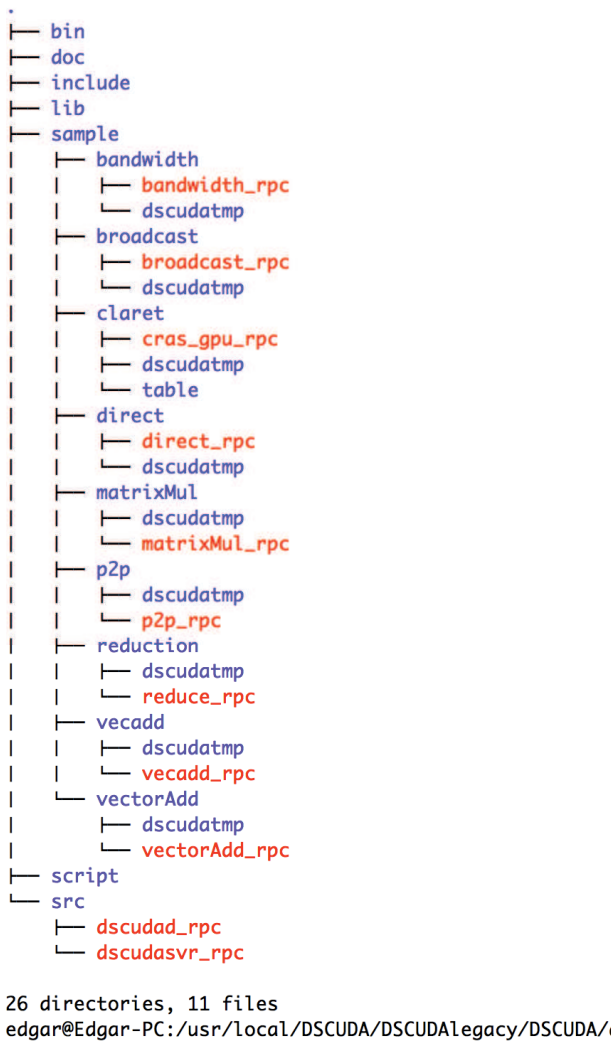

Figure 4.16: DS-CUDA SDK generated with our makefile script.

which are the main target for DS-CUDA framework. Also, GitHub provides free account with indefinite storage but each repository may do not overpass 1GB and no more than 100 MB per file. You can have as many repositories as you desire. Another important issue for free accounts is that the repository must be on the public domain, which is not a restriction for DS-CUDA due to its open source nature. We hosted all versions of DS-CUDA inside of the https://github.com/Daweek/Original_SC. All of them are a viable for downloading and its free. You can get it from the address above or you can try inside of your terminal with the next instructions:

First

Install git on your Linux system: **apt-get install git**

Second

Add the remote repository: **git remote add origin https://github.com/Daweek/Original_SC.git**

Third

Pull the code from the repository: **git fetch –all**

Fourth

Reset to the last version: **git reset –hard origin/master**

Using GitHub help us to keep control of DS-CUDA versions as well as distributing this to major audience.

5

Claret, Molecular Dynamics visualization software

The molecular dynamics simulation (MD) is a natural phenomena descriptor of the matter structure and composition which runs over a computer. This especial software has helped to the better understanding and interpretation of certain material structures. The MD simulation has arisen as a part of Physics Theory, Chemistry, Mathematics and Computer Science. This discipline is dedicated to apply simulation techniques in which atoms and molecules interact whit each other by a certain and quantified amount of time, thus be able to visualize though the computer the evolution and behaviour of the system. Sure enough, this simulations pushes to the limit the power inside of the machine's hardware due to heavy and many computations per body in the system.

5.1 General Process of MD simulation

The molecular dynamic simulation are the answer of the integration of Newton's motion laws, as well as the description of approximate force field generated based on the particle interactions. It is true that there is not only one definitive simulator of MD, there are many which offers many different capabilities and implements different algorithms. Some of them are ACEMD [16], OpenMM [17], NAMD [15] and Amber[14], which uses GPU or another special hardware accelerator. Even with all differences in the variety of simulators, all of them may follow a similar process which is described in Figure 5.1.

The MD simulation itself is a numeric solution of the motion equations, solving present forces acting on the atoms derivated from the potential energy of its 3 spacial components $(x, y \ \& \ z)$. The time step is particularly small, from the order of $t \sim 10^3 - 10^6$ steps which corresponds to some nano seconds in the real life.

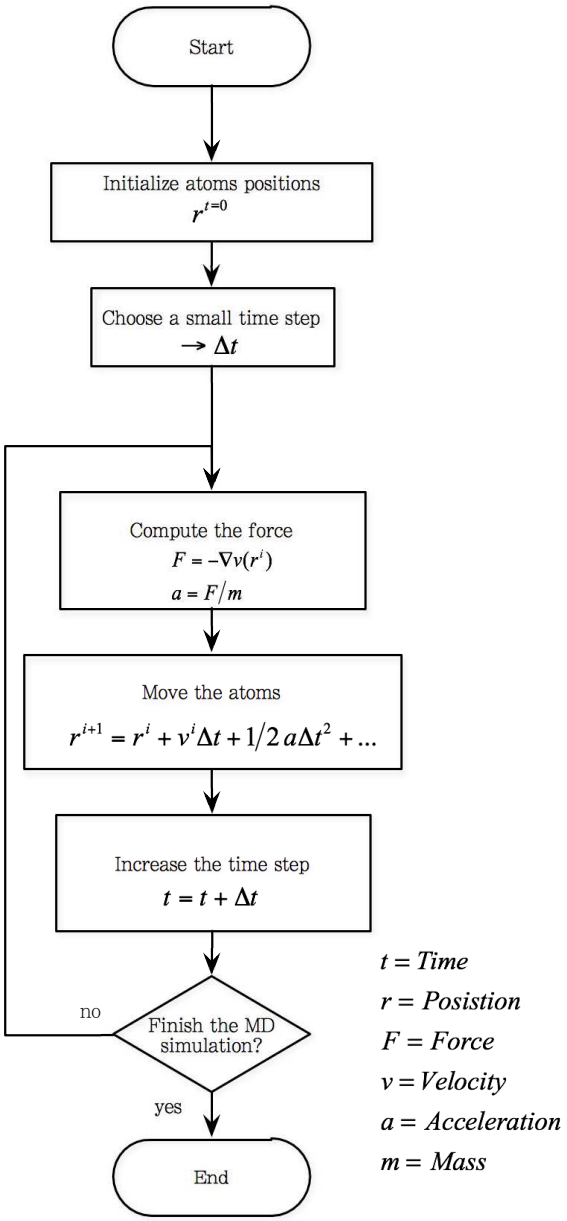

Figure 5.1: General algorithm flow of a molecular dynamic simulation.

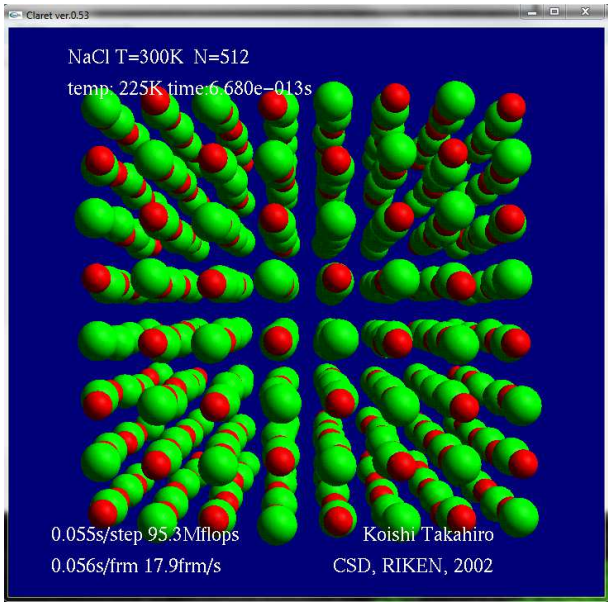

Figure 5.2: Claret simulator, screen-shot.

5.2 Claret overview

Claret MD simulator was born as a educational software purposes developed by Dr. Takahiro Koishi. This software was mainly developed to show the massive computational power of the *Molecular Dynamics Gravity Pipe* (MD-GRAPE 2). This special purpose hardware allows the acceleration of the MD by using several process in parallel. This devices was developed in The University of Tokyo [24] [25] and lately taken by the *Natural Sciences Research Institute* (RIKEN). This special purpose hardware has been used to achieve the Gordon Bell price [1] of best performance on 1995, 1996 and 1999 editions. Claret is written in C language and uses OpenGL for rendering the particles. On its first stages of development, MDGRAPE libraries were included. Nowadays, claret is an educational purpose software used to understand basic MD between particles and also to learn parallel computing techniques. The source code is open and can be downloaded from this address http://atlas.riken.go.jp/~koishi/claret_e.html.

Inside of the simulator we can appreciate sodium (Na+) and chloride (Cl-) particles, a salt crystal which can be viewed on the Figure 5.2. All the particles reside at vacuum level, delimited by a cubic sub space in the visualization. Also, includes variation of the temperature and pressure. If the crystal reaches its boiling or fusion steps, the particles are not able to scape from the wall.

Inside of the application we can visualize the particles behaviour in real time through different capabilities that the software offers, such as:

- Real time visualization of the particle conglomerate.

[1] The Gordon Bell Prize is awarded each year to recognize outstanding achievement in high-performance computing.

Key	Action
q	Exit
v	Visible information on/off
t	Temperature +100K
g	Temperature -100K
y	Temperature +10K
h	TTemperature -10K
!	Restart
z	Pause or Continue
s	Change the time step $+10 \times 10^{-15}$ sec.
c	Change background color
M	27 ion for collision
N	4 ion for collision
m	1 negative ion for collision
n	1 positive ion for collision
1-9	Velocity for collision or number of particles in the system
space	Shot ion for collision

Table 5.1: List of keyboard actions inside of Claret

- Different vision/camera angle.
- Increase - Decrease the temperature.
- Extra ion collision.
- Particle rendering using textures and polygons.
- Useful information of the system: Force performance computation and frames/sec.
- Stereoscopic vision[2].

Some of the capabilities are enabled during compilation time using the `# define` C directive such as the ability of rendering with polygons or textures, stereoscopic vision and the usage of an external accelerator. Other options inside of claret can be enabled by pressing some specific keys, as is shown in Table 5.1.

For the keyboard numbers 2 options are provided: 1. Select the velocity collision for the new generated ions. To shoot space key is required to press. 2. Select the number of ions present in the whole simulation. This particular number is a multiple of 8. If $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ then the total amount of particles, is $n = X \times X \times X \times 8$. In this sense, 8 is the minimum and 5832 the maximum.

[2]For 3D vision a special high frequency display and special glasses are need it

5.3 Technical Specifications

Claret MD simulator follows a general process as the one depicted in Figure 5.1. Next we enlisted all the process that are involved in the main body of the code.

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

As we can observe, the visualization is using OpenGL code and the auxiliary library *glut*. The first lines of code are for the allocation of memory space for large arrays and other variables. Also, the OpenGL state is initialized, creating a the appropriate window. The initial state of the variables, such as, position, pressure, temperature are defined. The *Display* function 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* function enable the mouse input which makes the camera to rotate. The function *Motion* computes a new frame according the new angle provided by

	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 5.2: Parameters of Tosi-Fumi potential. $B = 3.15\mathring{A}^-1$

the mouse motion. *Keyboard* enables the actions provided in Table 5.1. The *Md_run* section is the core of the MD simulation, where the computation of the force, velocity and other constants is done. Finally, the *Mainloop* keeps the simulation alive until the user press "q" or kill the application.

5.3.1 Force calculation

Inter-ionic potential of a rigid-ion model proposed by Tosi and Fumi [26] is used as a force field between ions.

$$\phi_{ij}(r) = \frac{q_i q_j}{r} + A_{ij} B \exp\left[\frac{(\sigma_i + \sigma_j - r)}{\rho}\right] - \frac{C_{ij}}{r^6} - \frac{D_{ij}}{r^8} \tag{5.1}$$

This potential consist of the Coulomb term, a repulsion term, a dipole-dipole term and a dipole-quadruple term, where q_i and q_j are electric charge and r its distance between them. It use the parameters of Equation 5.1 given by Tosi and Fumi. This parameters are showed on Table 5.2. Time integration is performed with the fifth-order predictor-corrector method [27]. The wall boundary condition is adopted. The system at vacuum level is initially equilibrated at $T = 300K$. The number of floating point operations per time-step for calculate force is $n \times n \times 78/t$, where n is the number of particles, 78 the amount of operations inside Equation 5.1 and t the time measured between each step.

5.4 Versions

As an educational software package, Claret has suffered alterations on its code, providing it with some new features. Between these new changes are included: New keyboard actions for the real time simulation, new visual information, different algorithm for force implementation on different accelerators e.g GPU and different methods to render the system particles, to name a few. There is no actual official record of the branching, but in this dissertation we consider to include 3 major versions.

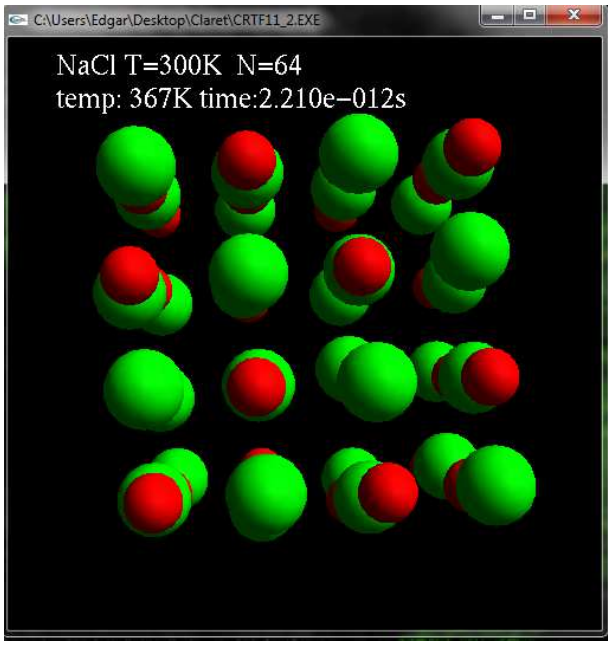

Figure 5.3: Claret simulator V 0.11

5.4.1 Claret V 0.11

This was the first version created by Dr. Takahiro Koishi and it can be found on its web page. The Figure 5.3 shows a screen-shoot of the simulation. Some of the capabilities of this simulator are listed below:

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

It is important to mention that this version do not has the cubic sub space, in other words the wall is not present. The actions provided by the keyboard remain the same as the showed in Table 5.1. The particle rendering is done through polygons and the detail level can be changed by pressing "R" key. All calculation process is done through CPU.

5.4.2 Claret V 0.53

Next major version was 0.53. In this new iteration of claret MD simulator the cubic sub space wall is present. Also, more information is display on the screen. This version was also developed by Dr. Takahiro Koishi and the software can be found on http://polymer.apphy.u-fukui.ac.jp/~koishi/claret/index.php. The Figure 5.4 shows a picture of this version. Some of the main capabilities of this version are showed below:

- Temperature on K scale.

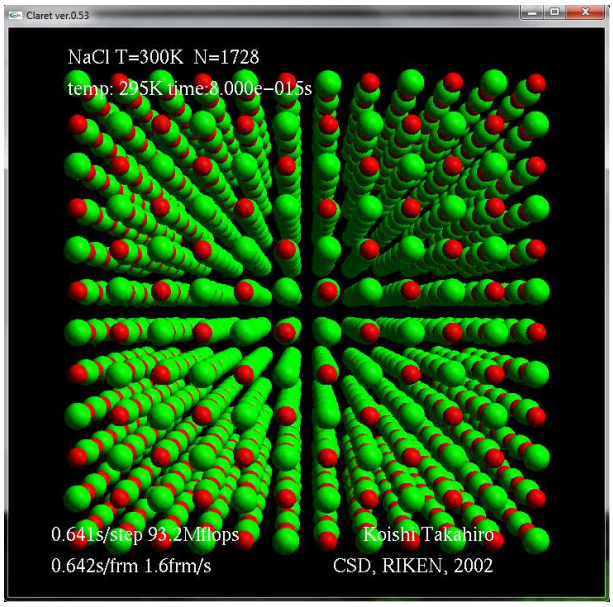

Figure 5.4: Claret simulator V 0.53

- Number of particles present in the simulation.
- Time step.
- Flops measurement.
- Frames per second.

For rendering process polygons and textures are a viable. It has stereoscopic vision enabled. Also the collision of new ions is possible. The force field between atoms can be computed by MD-GRAPE or CPU hardware.

5.4.3 Claret V 1.0

The next version is developed in Narumi Tetsu lab, which is mainly modified to accelerate the force computation using GPU hardware accelerator. Figure 5.5 shows and output of the bodies system in this version. Some of new capabilities are:

- Temperature on K scale.
- Number of particles present in the simulation.
- Time step.
- Flops measurement.
- Frames per second.
- Ion type and charge.

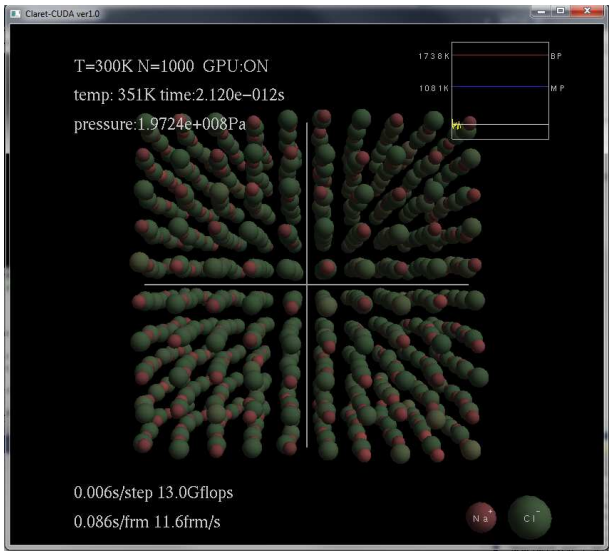

Figure 5.5: Claret simulator V 1.0

- Pressure and temperature meter.
- Accelerator type.

This version was created also as a education purpose. Learning the techniques of GPGPU through CUDA architecture, you can see the impact of performance on Claret. Also, to show how is the acceleration between using GPU and CPU, Claret software is a interesting and intuitive way of learning parallel computing.

5.5 Android Port

One of the main purposes of this dissertation is to create applications which merge high performance computing and mobile devices. The author has a previous experience with claret using CUDA to accelerate the simulation and the behaviour [30], thus, a port of this application was one of the first ideas to merge DS-CUDA over Android devices. The molecular dynamics simulation is an interesting application which has a different impact of the user experience on the tablet due to its touching capabilities and many sensors. Creating a more dynamic and immerse interface to interact with atoms its not easy task due to its ARM processor.

To complete this task first lets look some of the characteristics of claret for PC. As Table 5.3 mention, is a C/C++ based software which uses OpenGL 4.1 for rendering the particles in the system. Indeed, most of the functions used are based on the specification OpenGL 1.1. Also uses freeglut library toolkit to make easier the window handling and others. The latest version of claret for PC includes 3 options to accelerate the force computation: CPU, GPU with CUDA, and remote GPU with DS-CUDA. Of course, the DS-CUDA version is from PC

Claret	
Code	C/C++
Code	OpenGL 4.1 freeglut
Accelerator	CPU CUDA DS-CUDA

Table 5.3: Technical specifications of Claret version for PC.

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

Table 5.4: Differences between the usage of OpenGL / OpenGL ES over claret.

(client) to PC (server).

Android ecosystem allows the usage of C/C++ code trough NDK, as well as the usage of OpenGL ES 1.1 and 2.0. However, at the time we started the porting neither *glut* or *freeglut* were a viable for Android. We tried to compile and use freeglut in order to save as much as possible and we succeed due to poor documentation and no any guided sample we couldn't use. This and other issues are detailed at the time of porting claret to Android are discussed.

5.5.1 Visualization using OpenGL ES 1.1

The first thing to take into account is the life cycle of the app. It is different the development of Android app than from the usual PC. The Figure 5.6 shows the life cycle of the app in Android OS. We chose to use OpenGL ES 1.1 due to the similarity of the code included on claret for PC. Then, we can re-use partial of the code without modification. However, between OpenGL and OpenGL ES are some differences denoted in Table 5.4

Due to the absence of the glut on C/C++ code for Android, we decided to configure this over Java. We found out that OpenGL instructions are independent of the code you are using them. For example, we can initiate the EGL context over Java and continue using API functions inside of C code. Android provide a class *opengl.GLSurfaceView* that helps

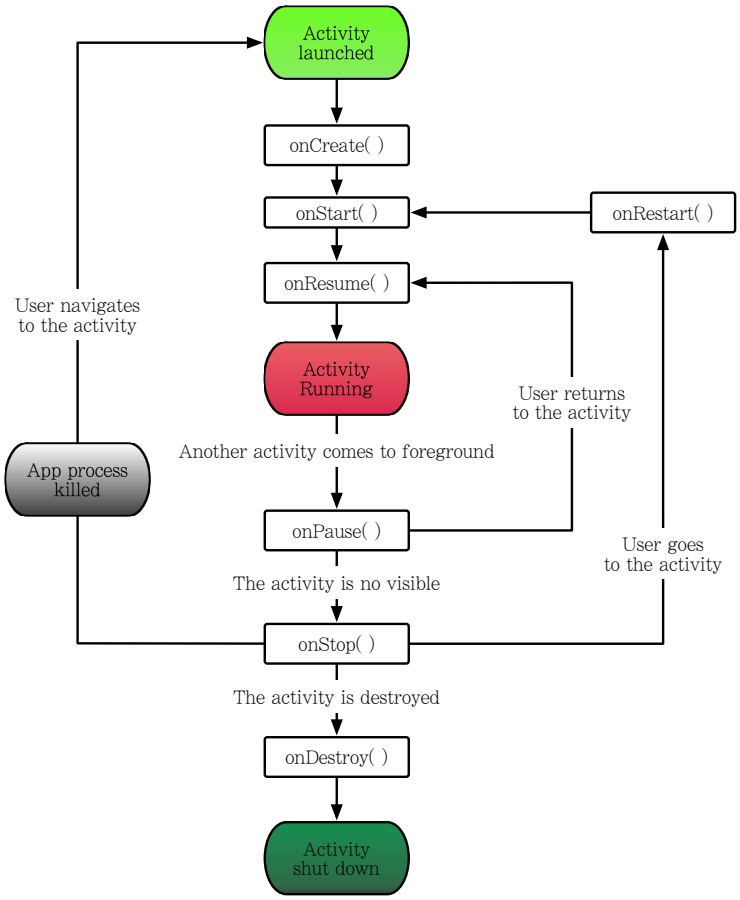

Figure 5.6: Android application life cycle.

to handle the content view for the app. This auxiliary library merge onCreate(), onStart() and onResume() app states, from the Figure 5.6, for new ones which connects to OpenGL ES pipeline such as: *onSurfaceCreated()*, *onSurfaceChanged()* and *onDrawFrame()*. All this three new functions are implemented on C connected trough its proper interface using NDK.

The first function, onSurfaceCreated() some variables and constant are initialized such as initial temperature, time step, velocity, force and position of the particles. The matrix model for the OpenGL and colors are initialized as well. Memory space for big arrays is also allocated. This process is only repeated if the application is restarted or created for the first time.

The second function, onSurfaceChanged() correspond to the resizing of the gl canvas for the actual size of the Android tablet. On tablets you may use it as portrait and landscape mode, which change the total size for the main window buffer in OpenGL. Nonetheless, in claret for Android we restricted the usage as landscape. The matrix model is defined here as well as the initial perspective. The depth buffer for color and depth are cleared in this instance in order to generate new frame.

The third function, onDrawFrame() includes all the rendering part. Basically implements

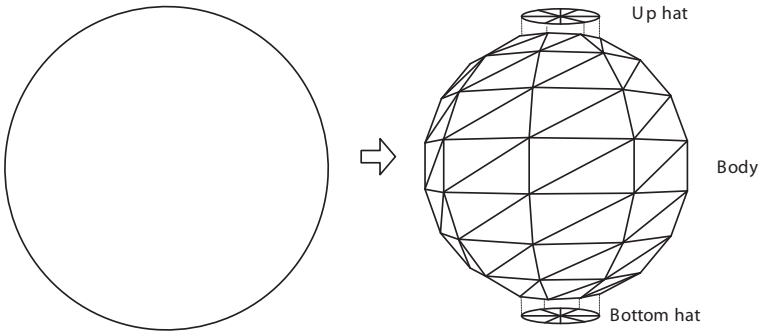

Figure 5.7: Sphere structure mapped by triangle primitives.

two functions: One which is the core for the MD simulation, including the force of the particles and other one which renders all the position of the particles. The computation of some visual information such as amount floating operation per second are performed inside of this routine.

In the original claret code, textures and polygons are able to draw all the atoms presents in the simulation. However, for the Android version we decided to use first polygons to render. Initially a function called *glutSolidSphere ()* from the GLUT library was used to depict all theses bodies. If we want to render massive number of bodies in the best performance way in OpenGL [31], the functions of *glBegin-glEnd* should be avoided. Instead, *glDrawArrays* functions should be called.

A detailed explanation of the sphere mapping is given next: The sphere is divided in 3 parts: the up-hat, the body and the bottom-hat. Each part is the result of drawing consecutive triangles. This is show in Figure 5.7. The amount triangles is due to the graphical detail given by the number $5 \leq d \leq 20$ that is the times division on the up hat and $s = \lfloor d/2 \rfloor$ that belongs to the number of the stacks. We can reach every vertex point with the next equations

$$x = r\mathbf{sin}\theta\mathbf{cos}\varphi \tag{5.2}$$

$$y = r\mathbf{sin}\theta\mathbf{sin}\varphi \tag{5.3}$$

$$z = r\mathbf{cos}\theta \tag{5.4}$$

Where x, y and z are the correspond coordinates, $\theta = 2 \times \pi/d$ and $\varphi = 2 \times \pi/(s \times 2)$. We decide $r = 1$ in order to calculate the general structure for unitary sphere coordinates. These unitary vertex and normal values are computed on CPU again if d changes. These values are store in one array G on constant memory on GPU.

We used 2 buffers to allocate vertex and normal vectors. Finally using glDrawArray() function, the particles are displayed on the screen. One first trial version is presented in

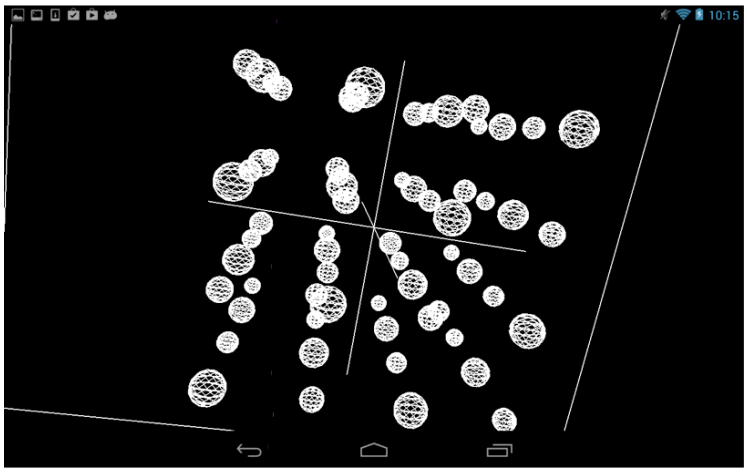

Figure 5.8: First claret version on Android tablet.

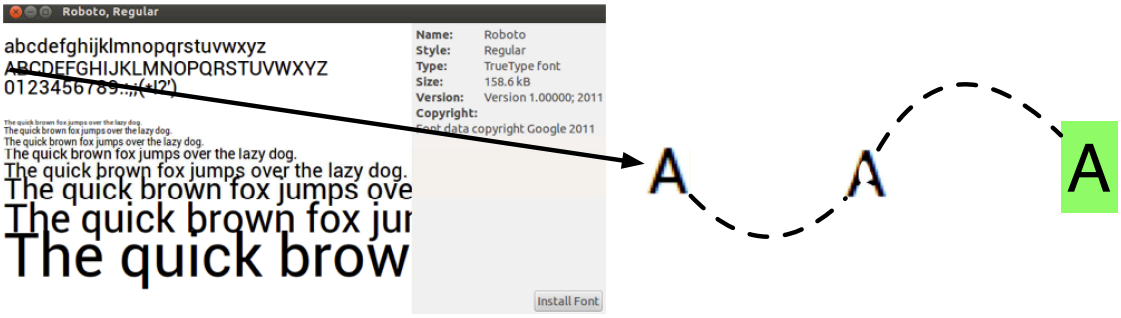

Figure 5.9: Font rendering using textures and .ttf file.

Figure 5.8.

Another issue in the porting process for claret on Android was the ability to render some information using text inside of OpenGL ES. Unfortunately this is not possible. For OpenGL 4.1 the is some utility for freeglut library which helps to render some basic text based on most commons fonts. Nevertheless, we were force to implement some function to display information on claret. A basic idea is to take some .tff file, which contain the actual font information, generate the font map inside of textures and allow to draw in OpenGL ES, as depicted in Figure 5.9.

5.5.2 Enabling DS-CUDA for force computation

As we mentioned in the Chapter 4, DS-CUDA client library is compiled using NDK and could be used in another projects. In claret for Android, we used as well. We put a CUDA code for claret through dscudacpp preprocessor and use it as a core for the force computation between the Na Cl ions. As we can observe in the code above, the *libdscuda_tcp.a* is complementing the claret.c code. Then, this is compiled as a shared library which is part of the onDrawFrame () function described in the last section.

```
1  ## Android.mk for Claret on Android
2  ##################Static Library libdscuda_tcp.a
3  LIB_LOCAL_PATH := $(call my-dir)
4  include /usr/local/DSCUDA/dscudapkg1.5.2/src/Android/jni/Android.mk
5  LOCAL_PATH := $(LIB_LOCAL_PATH)
6
7  ##################Static Library DS-CUDA Routine
8  include $(CLEAR_VARS)
9
10 LOCAL_MODULE     := MR3call
11
12 LOCAL_CFLAGS := -O0 -g -P -ffast-math -funroll-loops -fpermissive -I.\
13 -I/usr/local/cuda/include \
14 -I/usr/local/DSCUDA/dscudapkg1.5.2/include/common/inc \
15 -I/usr/local/cuda/samples/common/inc -DTCP_ONLY=1
16
17 LOCAL_SRC_FILES :=  mr3.cpp
18 LOCAL_STATIC_LIBRARIES := dscuda_tcp1.7.5.1
19 LOCAL_LDLIBS := -ldl -llog
20 include $(BUILD_STATIC_LIBRARY)
21 ##################Compile Claret main program
22 include $(CLEAR_VARS)
23
24 LOCAL_MODULE := claret
25
26 LOCAL_CFLAGS := -g -W -DANDROID_NDK -DDISABLE_IMPORTGL
27
28 LOCAL_SRC_FILES := \
29      claret.c \
30      app-android.cpp \
31
32 LOCAL_STATIC_LIBRARIES := MR3call
33 LOCAL_LDLIBS := -lGLESv1_CM -ldl -llog
34
35 include $(BUILD_SHARED_LIBRARY)
```

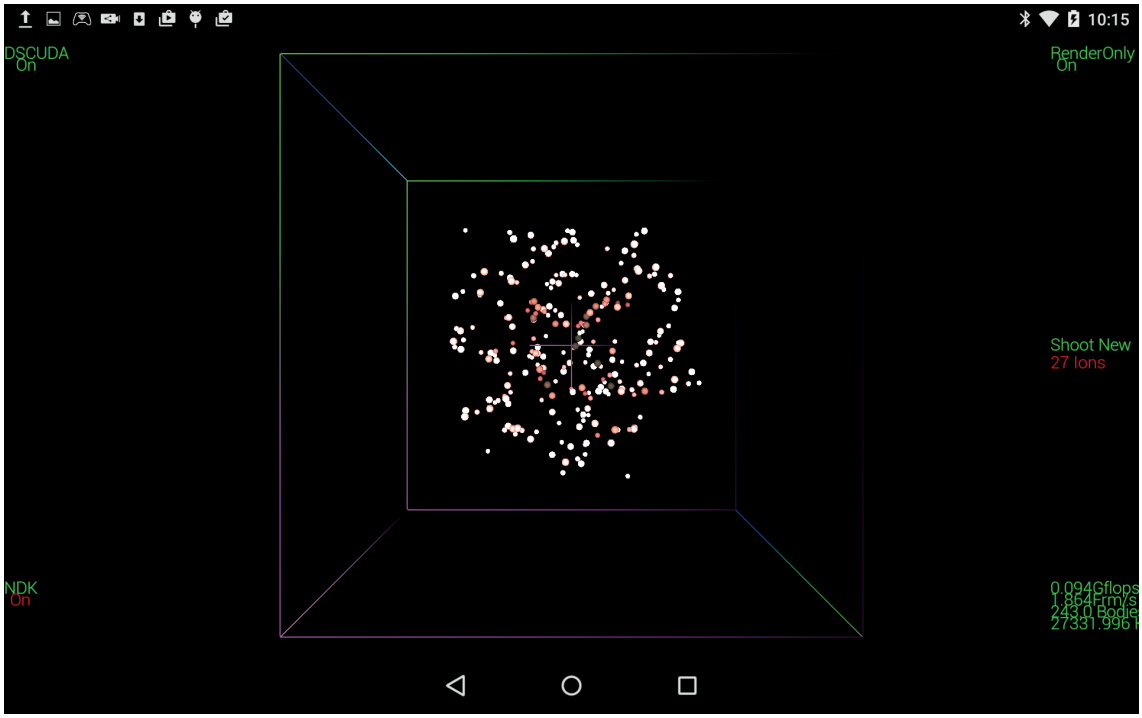

Figure 5.10: Output of claret for Android. Last version.

Finally, we show a picture of the final version of claret on Android tablet. This show less features than the original one or version 1.0. Due to the time and also proting issues, only few information is visualized. The functionality that is included in the app are:

- Changing the camera angle through touch screen capabilities.
- Shot 27 new ions.
- Render only (Take the force routine off).
- DS-CUDA as accelerator.

The functionality is limited in this version of claret for Android, however the capability to use DS-CUDA as an accelerator for force computation has been very attractive to users. We also distribute this software through Google Play store. The software is under educational purposes licence and under the name of *Edgar Josafat Martinez Noriega*. More functionality for claret on mobile devices, as the previous desktop versions, are part of future plans.

6

Evaluation of Claret over different systems

We successfully integrated DS-CUDA framework for mobile devices, specifically for Android tablets. We also achieved the port of claret for Android, which is a adequate n-body simulation to test DS-CUDA due to its heavy computations involve. In this section we specify the components of our system that were used during the experiment. We conform a typical DS-CUDA system, client (Tablet) and server (Laptop). Also, we tested the first mobile CUDA capable GPU inside of Jetson K1 development kit.

We performed the measure of the bandwidth performance transporting data through client and server using different protocols and physical layers. We draw a general model to describe the performance of claret. Also, a general view of DS-CUDA using tablet versus the native CUDA for embedded systems is shown. Last, we show a CPU performance of claret on all different devices we used in the experiment.

6.1 System architecture

The DS-CUDA system is composed as in Figure 6.1. The client is the NVIDIA SHIELD; a platform gaming device which runs Android. Basically is a tablet with a built-in controller. The server is a Laptop Alienware equipped with a CUDA capable GPU running Knoppix OS. We chose a laptop due to its portable purposes in case of demonstration such as conferences and others. The SHIELD device was chosen due to its rendering power coming from its gaming platform nature. The SHIELD device is connected to our local network using Wi-Fi 802.11n. The server is connected through Gigabit Ethernet. Detailed explanation of each component is provided on Table 6.1.

We performed molecular dynamics simulation and visualization using claret over different mediums. The Jetson K1 development kit is tested as well. Next sections a detailed

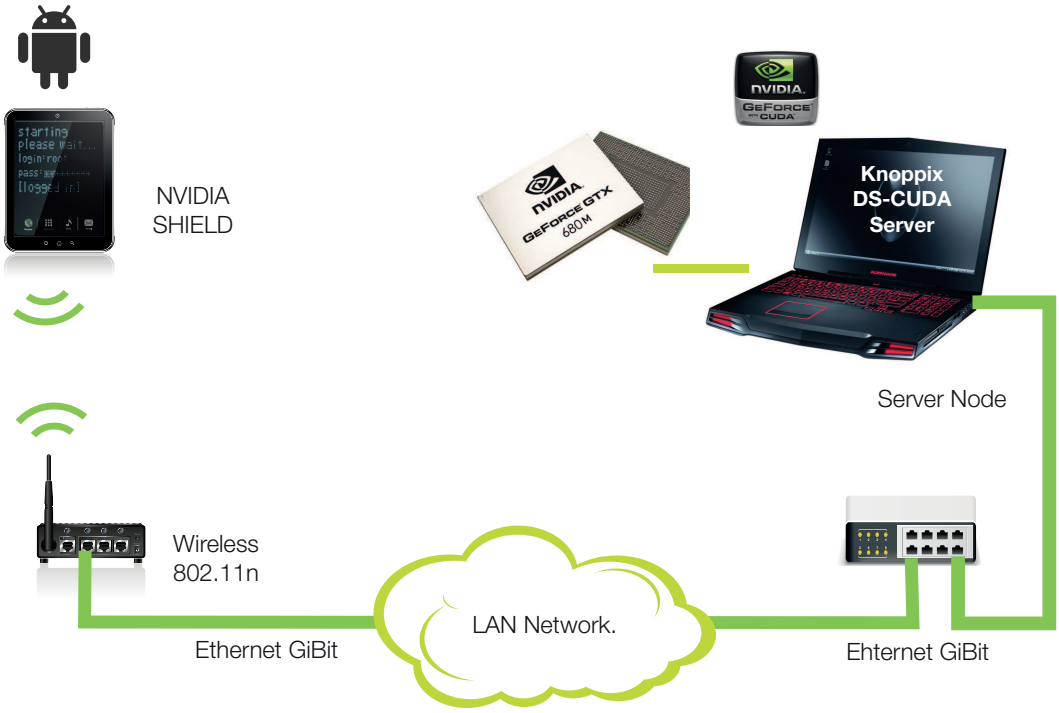

Figure 6.1: System components overview.

Device	CPU	GPU	Memory	OS	CUDA
Alienware Knoppix 7.02 32	Intel Core i7, 2.30 GHz, 8 Cores	GeForce GT 680M, 7 Multi-Processors, 1344 CUDA Cores, Global Memory 2047Mbytes.	16 Gbytes, DDR3, 1600 MHz	Knoppix7.0.2 x86 Linux	Driver 331.62, Toolkit 6.0, SDK 6.0
Tegra K1	Intel Core i7, 2.40 GHz, 8 Cores	Tegra K1 (GK20A), 1 Multiprocessors, 192 CUDA Cores, Global Memory 1746 Mbytes.	2 Gbytes, DDR3L, 933 MHz	Linux for Tegra (Ubuntu 14.04 for ARM)	Driver "Custom for Jet-son K1", Toolkit 6.0, SDK 6.0
NVIDIA "SHIELD"	NVIDIA Tegra 4, ARMv7, 1.912 GHz, 4 Cores	NVIDIA AP, 72 Custom Cores,	2 Gbytes, DDR3L & LPDDR3	Android 4.4.2	- - -

Table 6.1: Specifications of each component of the system.

Figure 6.2: Jetson K1 development kit.

explanation about the experiment is mentioned.

6.2 Bandwidth performance over different mediums

We conducted an experiment in which the performance of the memory copy function is measured. As we mentioned in Chapter 2, to compute with the GPU, first we have to migrate the data from the host (CPU) to the device (GPU). This is done through a CUDA function called *cudaMemcpy ()*. Here you can specify if you want to send data from the CPU to the GPU *Host to Device (H2D)* or from the GPU to CPU *Device to Host (D2H)*. DS-CUDA also implements this function, however the speed of transferring data between the client and server is not the same due to different buses e.g. Ethernet or PCI express. The aim of this test is to show the data transfer speed between various mediums and also to calculate the latency generated in order to complete the analysis for claret in the next section.

We iterate over a loop which send a package and we measured the time to perform the action. The size of the package initially is set to bytes and it doubles until reach 300 Mbytes. We choose this limit due to memory management of Android device. The memory transfer was done trough Jetson K1, PCI express gen 3 X8 and X16 lanes, Gigabit Ethernet, 100M Based Ethernet and Wi-Fi. For the PCI express and Jestson K1 measurement we used native CUDA API and nvcc to compile the code. DS-CUDA preprocessor was used in other cases. 100M Based Ethernet was done through the Android tablet using a USB-Ethernet cable adaptor. Results for this experiment are shown in Tables 6.2 and 6.3.

A full graph of this measurement is shown in Figure 6.3. Next, we performed the measurement of the latency generated over the different mediums. The latency is defined in

Test - Size in bytes	1,024	2,048	4,096	8,192	16,384	32,768	65,536	131,072	262,124
Jetson K1 H2D	16.5436	38.6104	68.2804	115.3441	170.7249	226.5096	288.2094	601.1550	801.9278
Jetson K1 D2H	15.2345	30.3196	56.4063	102.9649	168.6475	265.0135	370.0867	473.1207	554.7091
PCI Ex. gen3 x16 H2D	123.4799	246.6512	493.8694	986.9060	1707.3868	2520.6735	3230.0812	4498.5478	5417.3793
PCI Ex. gen3 x16 D2H	83.4372	164.8430	323.0456	598.6791	1048.0041	1712.2448	2578.8914	3242.7331	3824.5164
PCI Ex. gen3 x8 H2D	108.4942	220.9411	412.1474	652.3812	921.4805	1152.7135	1288.6951	2334.3304	2464.5357
PCI Ex. gen3 x8 D2H	88.8891	140.5244	309.6343	498.7664	698.8008	893.7161	1719.9390	2328.0414	2407.9779
Gigabit H2D	2.4224	1.9678	3.5859	5.6537	7.3602	16.2941	23.3966	43.0115	78.1852
Gigabit D2H	2.4075	3.4797	5.7331	10.7801	19.8839	30.6060	40.8946	46.6500	69.7917
Gigabit H2D	2.3829	4.5106	8.9092	5.9953	8.0703	10.6709	17.8936	44.7310	55.6621
Gigabit D2H	1.9052	2.8307	5.2878	9.9469	17.8264	31.9544	42.9538	61.6025	78.3224
100Base H2D	0.7498	1.3238	1.9393	3.4554	5.6161	6.9986	8.4003	9.1866	10.6754
100Base D2H	0.7297	0.0210	0.3014	0.7566	0.1288	1.0402	1.2159	1.4161	2.0187
Wireless 802.11n H2D	0.3047	0.5141	0.9846	1.5067	2.3050	3.7204	5.1476	6.3466	7.1067
Wireless 802.11n D2H	0.2993	0.3239	0.6289	1.4893	2.0764	3.6316	5.9964	7.7838	9.5621

Table 6.2: Bandwidth performance between different mediums in MBytes. Package size from 1,024 bytes to 262,124 bytes.

Test - Size in bytes	524,288	1,048,576	2,097,152	4,194,304	8,388,608	16,777,216	33,554,432	67,108,864	134,217,728	268,435,456
Jetson K1 H2D	1085.1180	1563.1240	2507.2161	3087.3464	3224.7961	3164.1626	3224.5621	3233.9383	3264.0898	3256.7215
Jetson K1 D2H	601.0634	638.8535	945.0413	1321.9882	1411.7925	1415.7340	1448.3326	1411.6470	1442.6021	1421.3941
PCI Ex. gen3 x16 H2D	5955.9116	6323.6388	7988.1712	8990.2682	9593.7894	9926.9244	10100.1976	10188.1405	10219.4747	10217.2752
PCI Ex. gen3 x16 D2H	4261.4334	4521.1063	5402.7942	5958.3451	6243.9639	6400.9457	6477.7050	6521.2451	6527.7012	6522.2228
PCI Ex. gen3 x8 H2D	2571.3771	4389.3479	5189.7889	5666.9241	5946.2786	6102.0282	6189.6638	6227.7194	6212.3782	6050.4029
PCI Ex. gen3 x8 D2H	2486.0908	2481.5759	4474.1948	5012.6136	5335.0922	5486.3340	5563.9491	5621.8923	5633.9119	5612.8873
Gigabit H2D	99.5929	104.8582	107.9304	109.7768	110.7024	111.1474	111.2350	112.6164	84.1994	84.3711
Gigabit D2H	88.5481	87.9968	92.5851	97.6200	98.4138	99.1563	99.3324	98.9072	74.0344	74.3269
Gigabit H2D	63.5589	66.9117	68.5698	75.3382	62.1478	66.8261	78.9695	72.8281	60.5933	49.9229
Gigabit D2H	83.0734	89.7843	89.0131	90.3493	73.7667	56.8534	89.0131	90.3493	56.8534	55.1797
100Base H2D	11.1035	11.1015	11.2329	11.3453	11.3447	11.3642	11.3731	11.3649	8.5134	8.5052
100Base D2H	4.1338	3.9361	1.8556	4.0876	2.2060	2.2989	2.2222	1.0390	0.5391	0.2462
Wireless 802.11n H2D	7.5368	7.8303	7.4656	7.5245	7.9160	7.5183	7.8949	7.9943	6.09252	5.7483
Wireless 802.11n D2H	10.5434	11.0278	11.2821	10.4856	11.0581	10.6549	11.011	11.0089	8.3021	8.2572

Table 6.3: Bandwidth performance between different mediums in MBytes. Package size from 524,288 bytes to 268,435,456 bytes.

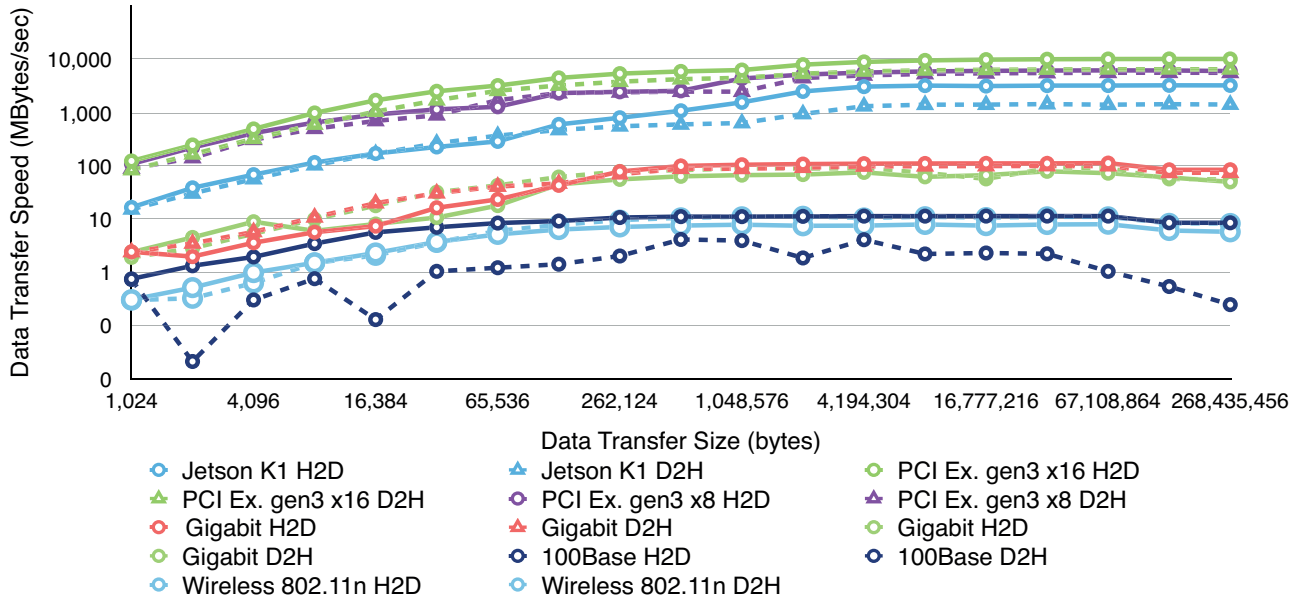

Figure 6.3: Total performance of cudaMemcpy over different mediums.

Equation 6.1. The minimum and maximum correspond to the data in Tables 6.2 and 6.3. The minimum package size is defined as 1024 bytes and maximum to 268,435,456 bytes.

$$L = \frac{MinimumPackageSize}{MinimumBandwidth} - \frac{MinimumPackageSize}{MaximunBandwidth} \tag{6.1}$$

Latency data over different mediums is concentrated on Table 6.4. As we expected, Wi-Fi has the biggest latency which may impact over the application performance sending the data to the GPU located in DS-CUDA server. Jetson K1 has a latency very similar to PCI express due to its construction, built in chip. For some unknow reason the 100M USB-Ethernet cable had some issue when it received data, probably due to the driver version.

Interconnection	**H2D Latency (sec)**	**D2H Latency (sec)**
Jetson K1	6.158E-05	6.650E-05
PCI Express gen3 X16	8.193E-06	1.212E-05
PCI Express gen3 X8	9.269E-06	1.134E-05
Gigabit Ethernet	4.106E-04	4.116E-04
Gigabit Ethernet K1	4.092E-04	5.189E-04
100 Base Ethernet	1.245E-03	-2.756E-03
Wireless 802.11n	3.183E-03	3.297E-03

Table 6.4: Memory latency.

6.3 Claret performance model

In this section we present a study of claret performance on different systems using GPU as an accelerator to compute the force between the particles. Our main target is to measure important process during the molecular dynamics visualization on claret such as GPU computation part, CPU computation part, communication between GPU and CPU and rendering process. In this way, we can observe and detect possible bottlenecks inside of the application.

The whole claret process is proposed in Equation 6.2, where T is the time per frame generated on claret, T_{GPU} is the lapse for computation over the GPU, T_{CPU} correspond to all other process inside of the CPU, T_{COMM} which is the time for CPU and GPU communication and T_{DISP} which correspond to the time for rendering the spheres by OpenGL.

$$T = T_{GPU} + T_{CPU} + T_{COMM} + T_{DISP} \tag{6.2}$$

In order to define each member of the Equation 6.2, we need to quantize this small region of time per particle.

$$T_{GPU} = md\_step * t\_gpu * n^2 \tag{6.3}$$

$$T_{CPU} = t\_cpu * n \tag{6.4}$$

$$T_{COMM} = t\_comm * Data * n * t\_L \tag{6.5}$$

$$T_{DISP} = t\_disp * n \tag{6.6}$$

Thereby, in Equation 6.3 *md_step* is the molecular dynamics step inside of the simulation. *t_gpu* is the force computation time between a pair of particles. n is the number of bodies inside of the simulation. On Equation 6.4, *t_cpu* defines the time per particle inside of the CPU. Inside of Equation 6.5, *t_comm* is the lapse of time to send data between GPU and GPU per particle. *t_L* is the latency of the medium used. Equation 6.6 defines the time per particle *t_disp* to be rendered by OpenGL. We isolated each process of Equation 6.2 on the code a measured directly, taking the time to perform the subroutine. The information generated per particle on different systems is concentrated on Table 6.5. We can observe that for SHIELD *t_gpu* is not present on the table results because it belongs to a DS-CUDA system which uses the GPU remotely from Alienware laptop.

We set up *md_step* to be 100, in order to reduce the communication between CPU and GPU. The *Data* parameter is fixed to 106 which refers to all data in bytes involve to solve

Jetson K1	Time per Particle
t_gpu	5.55E-10
t_cpu	3.69E-07
t_comm	1.89E-10
t_L	9.28E-04
t_disp	2.68E-05
AlienWare	Time per Particle
t_gpu	1.07E-10
t_cpu	7.72E-08
t_comm	2.32E-08
t_L	2.03E-05
t_disp	4.84E-06
SHIELD	Time per Particle
t_cpu	5.76E-06
t_comm	8.18E-07
t_L	6.48E-03
t_disp	1.91E-04

Table 6.5: Time per particle for each process on claret on different systems.

the force between the particles inside of the GPU. Once all the information is gathered, we proceed to solve Equation 6.2 for all n possible arrangement of particles in claret. In Table 6.6 we show these results. We included the simulation time as well.

Figures 6.4, 6.5 and 6.6 shows individual graphs for claret performance on Alienware laptop, Jetson K1 and SHIELD device respectively. Using the model we proposed and also the data taken from the real simulation. As we can observe, all tthree graphs have gaps at the beginning which are caused by the refresh of the screen. The simulation can not go faster than the refresh rate of the screen due to OpenGL rendering. However, the dotted line matches the model after this barrier is broken.

Figures 6.7, 6.8 and 6.9 shows each phase of whole process in claret software. The results using CUDA native (Alienware and Jetson K1) are very similar, the bottleneck for small amount of particles is the communication between CPU and GPU. Altough, increasing the number of bodies in the system, we reduce this problem. The rendering part for both system is not a problem. However, for SHIELD using DS-CUDA we found that for small amount of particles the communication (usign Wi-Fi) generates a bottleneck. Nevertheless, increasing the number of particles, the real bottleneck become the visualization. This is due to portable devices do not has a powerful GPU that can render a lot of polygons on the screen. One possible solution is to use textures instead of drawing triangles. This will be included as a

Particles	8	64	216	512	1000	1728	2744	4096	5832
T_{GPU}	3.55E-06	2.27E-04	2.59E-03	1.45E-02	5.55E-02	1.66E-01	4.18E-01	9.31E-01	1.89E+00
T_{CPU}	2.95E-06	2.36E-05	7.96E-05	1.89E-04	3.69E-04	6.37E-04	1.01E-03	1.51E-03	2.15E-03
T_{COMM}	9.28E-04	9.29E-04	9.33E-04	9.39E-04	9.49E-04	9.63E-04	9.84E-04	1.01E-03	1.05E-03
T_{DISP}	2.14E-04	1.71E-03	5.78E-03	1.37E-02	2.68E-02	4.63E-02	7.35E-02	1.10E-01	1.56E-01
Model K1	1.15E-03	2.89E-03	9.39E-03	2.94E-02	8.36E-02	2.14E-01	4.93E-01	1.04E+00	2.05E+00
Simulation K1	2.900E-02	3.300E-02	7.600E-02	1.040E-01	1.660E-01	2.560E-01	5.400E-01	1.022E+00	1.951E+00
T_{GPU}	6.840E-07	4.377E-05	4.986E-04	2.801E-03	1.069E-02	3.191E-02	8.047E-02	1.793E-01	3.635E-01
T_{CPU}	6.176E-07	4.941E-06	1.668E-05	3.953E-05	7.720E-05	1.334E-04	2.118E-04	3.162E-04	4.502E-04
T_{COMM}	4.033E-05	1.805E-04	5.608E-04	1.302E-03	2.523E-03	4.345E-03	6.887E-03	1.027E-02	1.461E-02
T_{DISP}	3.874E-05	3.099E-04	1.046E-03	2.479E-03	4.842E-03	8.367E-03	1.329E-02	1.983E-02	2.824E-02
Model Alien	8.037E-05	5.391E-04	2.122E-03	6.622E-03	1.813E-02	4.476E-02	1.009E-01	2.097E-01	4.068E-01
Simulation Alien	1.700E-02	1.800E-02	1.900E-02	1.900E-02	2.300E-02	4.300E-02	9.300E-02	1.730E-01	3.270E-01
T_{GPU}	6.840E-07	4.377E-05	4.986E-04	2.801E-03	1.069E-02	3.191E-02	8.047E-02	1.793E-01	3.635E-01
T_{CPU}	4.608E-05	3.686E-04	1.244E-03	2.949E-03	5.760E-03	9.953E-03	1.580E-02	2.359E-02	3.359E-02
T_{COMM}	7.187E-03	1.214E-02	2.557E-02	5.173E-02	9.485E-02	1.592E-01	2.490E-01	3.685E-01	5.219E-01
T_{DISP}	1.531E-03	1.225E-02	4.134E-02	9.800E-02	1.914E-01	3.308E-01	5.252E-01	7.840E-01	1.116E+00
Model SHIELD	8.765E-03	2.480E-02	6.865E-02	1.555E-01	3.027E-01	5.318E-01	8.705E-01	1.355E+00	2.035E+00
Simulation SHIELD	2.047E-01	2.170E-01	2.515E-01	3.477E-01	4.450E-01	5.549E-01	7.564E-01	1.014E+00	1.366E+00

Table 6.6: Model and simulations results of claret performance.

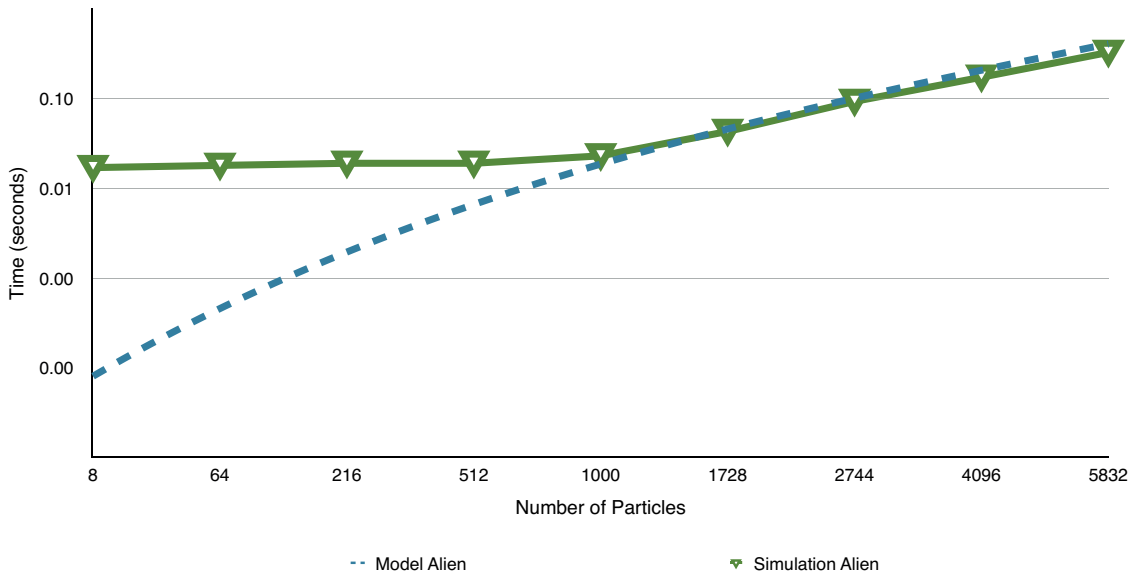

Figure 6.4: Total claret performance on Alienware - Model vs Simulation

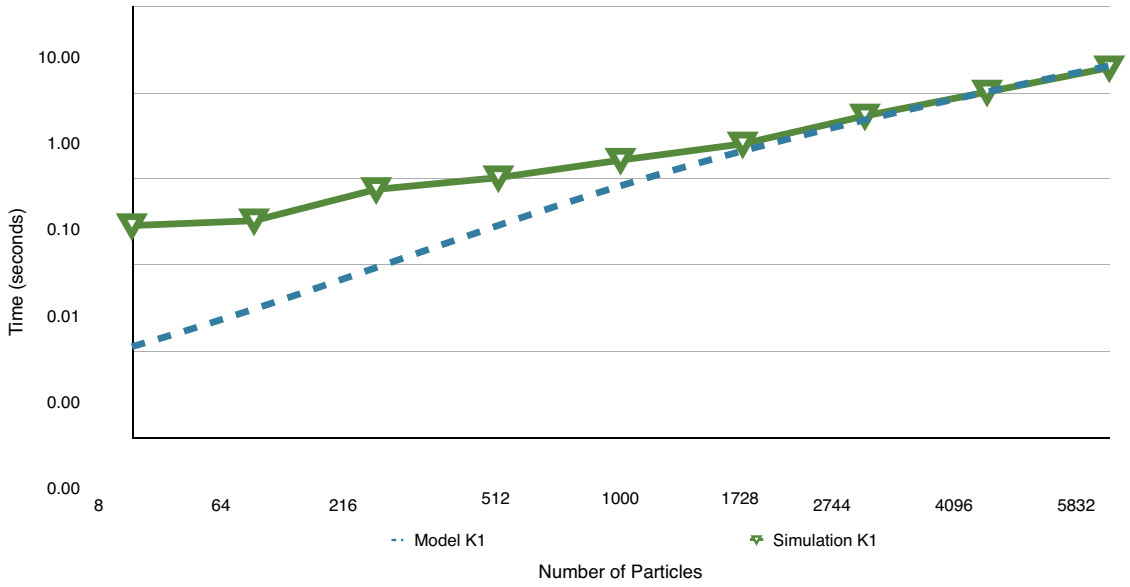

Figure 6.5: Total claret performance on Jetson K1 - Model vs Simulation

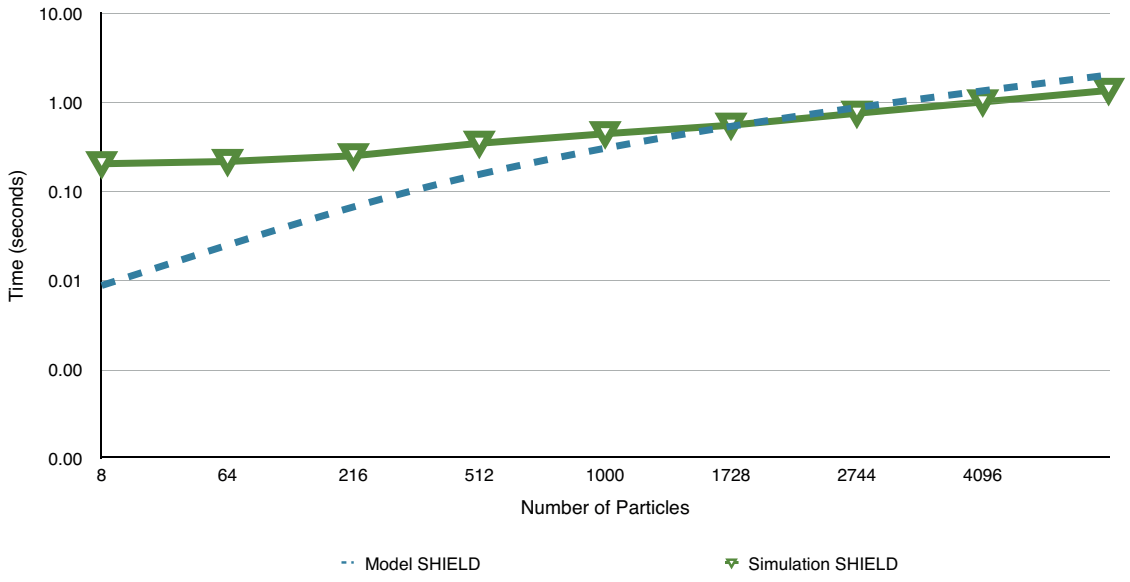

Figure 6.6: Total claret performance on SHIELD - Model vs Simulation

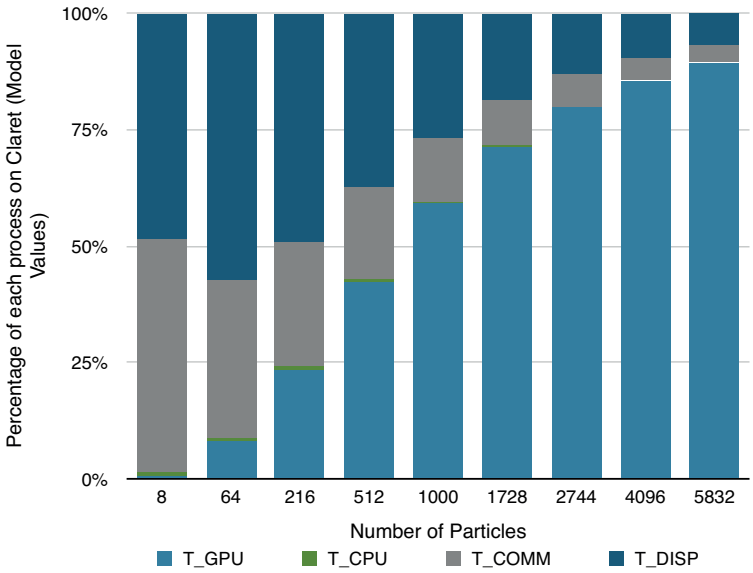

Figure 6.7: Percentage of each process on claret performance - Alienware.

future work.

In this section we also included the measurement of the claret application running only on mobile devices. This is due to principal motivation and objective of this dissertation. Figure 6.10 and Figure 6.11 shows the result between Jetson K1 and SHIELD running claret. We can denote that for the total time on Jetson for low amount of particles is one order of magnitude than in SHIELD using DS-CUDA. However, for large amount of particles, the curve shows better performance, even with the bottleneck over the communication and rendering part for SHIELD. The amount of floating operations per second is calculated as mentioned in Chapter 5, $n \times n \times 78/t$, where n is the number of particles, 78 the amount of operations inside Equation 5.1 and t the time measured between each step. For SHIELD and Jetson K1 we see similar trajectories due to usage of CUDA in both scenarios. However, SHIELD through DS-CUDA accelerates slowly compare with native CUDA due to the communication between server and client. Nevertheless, DS-CUDA shows better amount of operations per second when the amount of particles is more than 2744.

Last, we show a graph, Figure 6.12, which contain the measurement of the force computation in claret using only the CPU. Clearly the laptop is superior due to better processor inside. SHIELD shows more than one order of magnitud less performance with its ARM CPU. Interestingly, Jetson is located right in the middle which reduces the barrier for the future mobile devices to half due to similar processors will be included as Tegra K1.

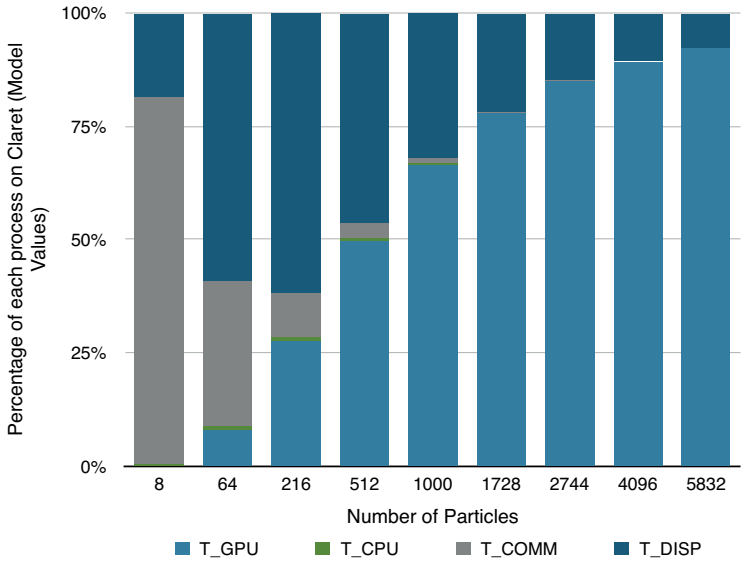

Figure 6.8: Percentage of each process on claret performance - Jetson K1.

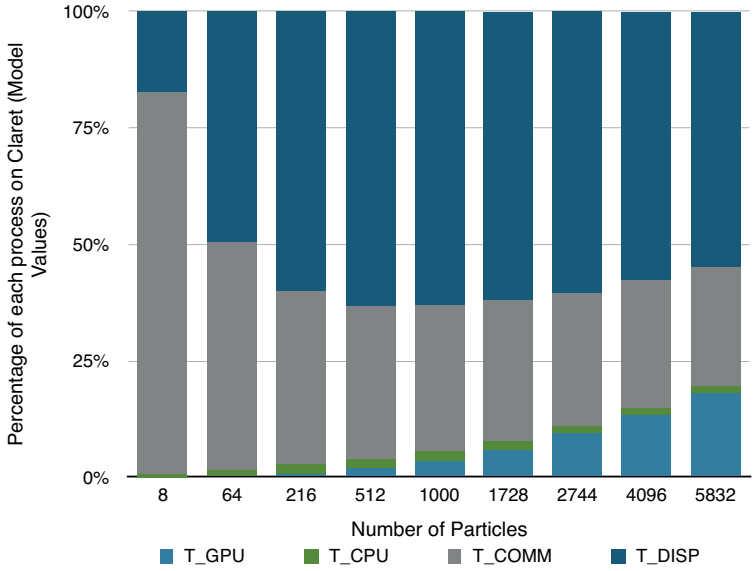

Figure 6.9: Percentage of each process on claret performance - SHIELD.

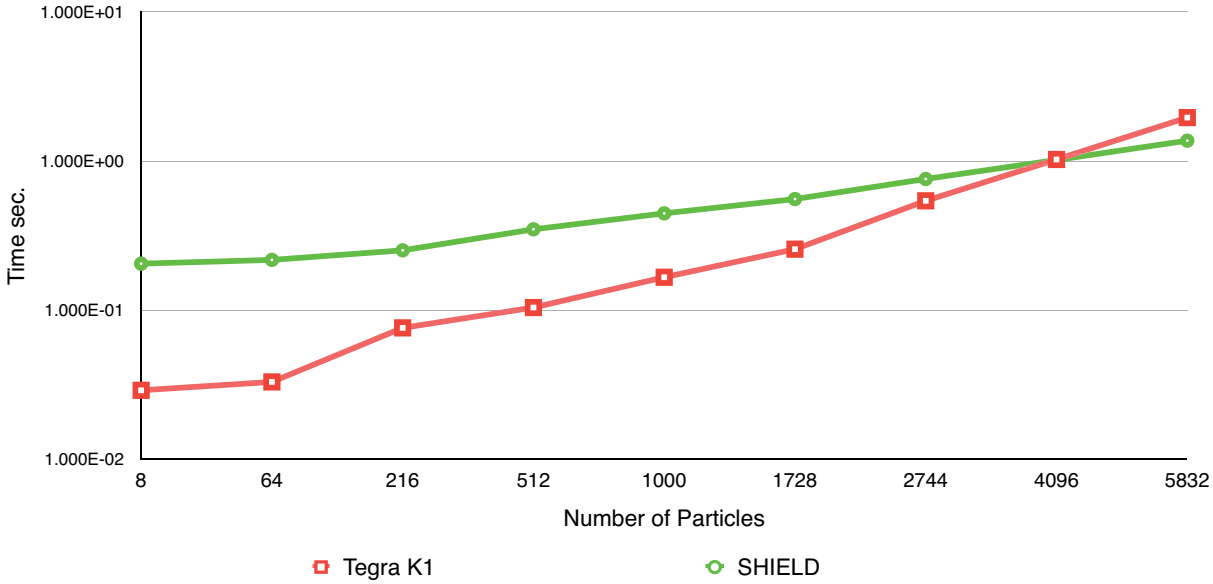

Figure 6.10: Real time claret performance on Mobile Devices.

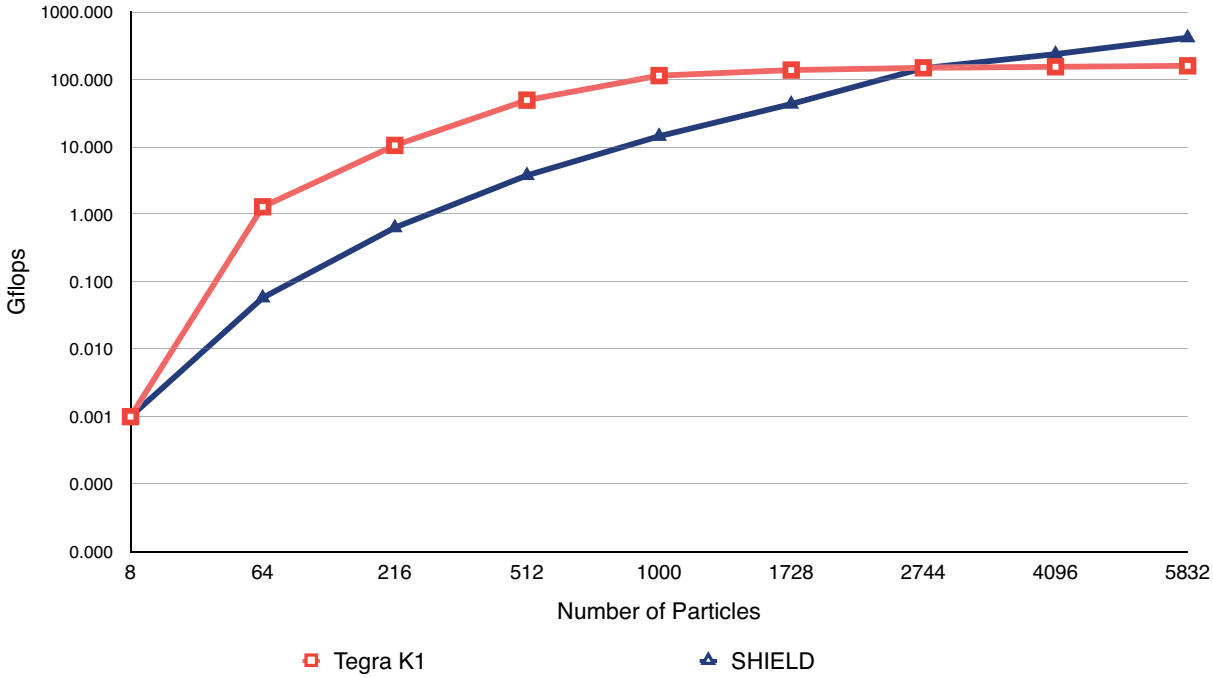

Figure 6.11: Force computation of claret on Mobile Devices. Accelerator GPU.

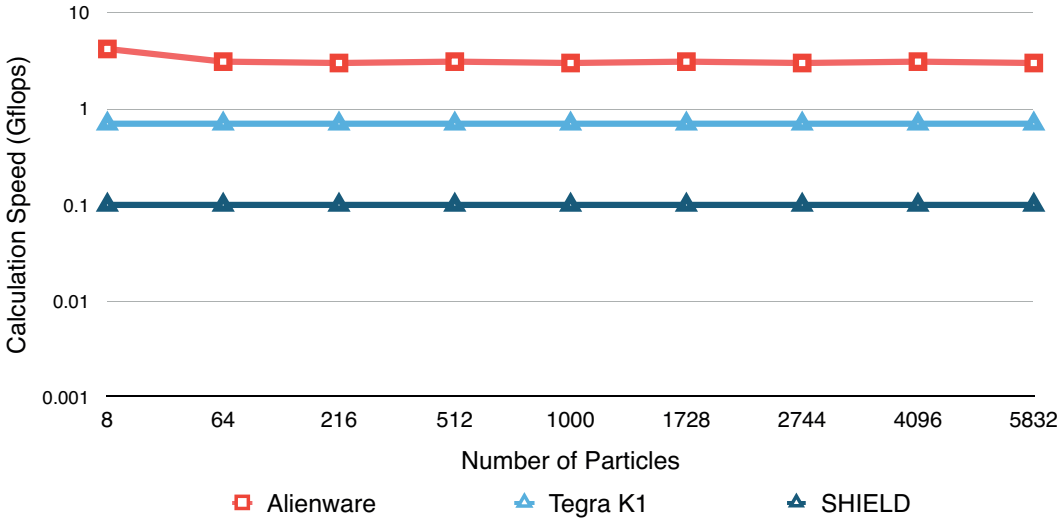

Figure 6.12: Force computation of claret using CPU only.

7

Conclusion

The usage of accelerators in high performance computing is more common now, such as the graphics processing unit through the hand of CUDA architecture. As a natural consequence, in order to handle a numerous GPUs in the cloud intermediate frameworks a software level has been developed such as DS-CUDA. This framework which visualizes a GPU inside of the local networks was our starting point to idealized the merge between high performance computing and mobile devices. In this work, we successfully achieved the acceleration and implementation of a molecular dynamics simulation and visualization on an Android tablet using a remote GPU to compute force between atoms inside of the system. We handle the cross compiling code for a client DS-CUDA library in order to use CUDA code inside of our Android application. We also achieved the porting of a molecular dynamics software, "claret" on Android tablet using OpenGL ES. Various differences were challenged as a result of versions inequalities between OpenGL and OpenGL ES such as rendering of spheres without supporting libraries and displaying font. As a result, claret has touching capabilities which makes totally different experience from the original version over desktop computers. We also provided a detailed analysis of the claret software including communication time, visualization time and the core process for molecular dynamics. From the results presented in Chapter 6 we can observe that DS-CUDA seems to be feasible solution in order to accelerate the n-body simulation. Even for a big number of bodies, DS-CUDA over wireless protocol works better than the embedded system Jetson K1. However, the actual problem in the simulation is the rendering time due to the low power GPU inside of the tablet. Nevertheless, this bottleneck could be alleviated if the number of primitives (triangles) per sphere reduces or changed for textures.

7.1 Future Work

A better implementation on OpenGL ES for rendering the spehere could be a future topic. Using textures or only rendering points to see the peak performance. Also, make an analysis of the power consumption over DS-CUDA system using the tablet. This could be compared between Jetson K1 and see how much flop/watt is given by the system. A interactivity application with the usage of tile display is also consider as a future topic.

DS-CUDA framework still need more functions to be implemented such as support for graphics interoperability with CUDA. As well as some Basic Linear Algebra Sub-programs (BLAS) which could be a future topic to implement. A complete system which could handle automatic recovering error and load balancing is also planned in future schedule.

References

[1] 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.

[2] 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. Kolossv?ry, 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.

[3] 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.

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] Stratton, J. A., Stone, S. S.,Hwu, W. W. *"MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs."*, In Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing LCPC. Canada: Edmonton. 2008.

[20] NVIDIA Corporation, *"The CUDA Compiler Driver NVCC"*, CUDA Compiler Driver ver ¡10-18-2011¿ pdf. 2011. pag. 22

[21] TOP500 Supercomputer Sites, . Available: http://www.top500.org/ [retrieved: November, 2014]

[22] Atsushi Kawai, Kenji Y asuoka, Kazuyuki Y oshikawa, 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

[23] M. Oikawa, A. Kawai, K. Nomura, K. Yoshikawa, K. Yasuoka, T. Narumi, *"DS-CUDA: a Middleware to Use Many GPUs in the Cloud Environment"*, Inter- national Workshop on Sustainable HPC Cloud at SC12, Salt Lake City, USA, 2012.

[24] 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.

[25] 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.

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

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

[28] Martin Cooper, et al., *"Radio Telephone System"*, US Patent number 3,906,166; Filing date: 17 October 1973; Issue date: September 1975; Assignee Motorola.

[29] Haire, Meaghan, *"A Brief History of The Walkman"*, Time. Retrieved 2010-12-24.

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

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