TALLINN UNIVERISTY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of General Informatics

PARALLEL COMPUTATIONS ON GRAPHICS CARDS USING CUDA

Bachelor final thesis

Student:	Erik Soekov
Studentcode:	112153
Supervisor:	Marko Kääramees

Tallinn
2015

Author’s declaration of originality.

I Erik Soekov declare that the following work is the result of my own effort and has not been presented for examination anywhere else.

Date: 20.12.2015

Name: Erik Soekov

Abstract

The purpose of this work is to examine if it is possible to accelerate a scientific computation 100 times when making it parallel and porting it to the GPU. The conditions in which this result can be achieved and how much effort it takes will also be examined.

Parallel applications using the CPU are popular and have helped create things that would be difficult to manage in a single threaded application. However a CPU does not have as many cores as a graphics card has, so perhaps it is possible to gain even more performance when the parallel computation is working on a GPU rather than on a CPU. The choice for implementing this kind of solution was a technology known as CUDA. The program that contained the functionality that was optimized was known as Molekel which is an open source program.

During this work CUDA was used to optimize a computation that at one point takes 44,664 seconds to complete into a computation that takes only 0,380 seconds to complete in that same point. This was roughly a 117 time speed-up.

However on a different computer, where there was a similar processor, but a card that had 15 times less cores, the computation came down from 39,158 seconds to 0.830 seconds, which resulted to a roughly 47 times speed-up.

This leads us to conclude that a programmer with a standard skillset in CUDA programming is capable of providing a remarkable performance boost; however an expert is probably capable of providing a lot more. In addition to that we must also take into account the hardware that the program will run on and the workload of the computation because this kind of a speed-up is not possible with every workload. In fact it decreases as the workload decreases.

Annotatsioon

Käesoleva töö eesmärk on uurida, kas on võimalik kiirendada teaduslikke arvutusi 100 korda, kui need teha paralleelseks ning programmeerida jooksma graafika kaardi protsessori peal. Uuriti ka tingimusi milles selline tulemus on üldse võimalik.

Paralleel programmid arvuti kesksel protsessoril ei ole midagi uut ning nad on aidanud luua võimalusi, mis oleksid rasked teha kasutades seriaalselt jooksvat programmi. Aga kesksel protsessoril ei ole sellisel arvul tuumasid nagu seda on graafika kaardil. Siit tekib küsimus, et kas oleks võimalik tõsta programmi võimekust kui seda jooksutada arvuti keskse protsessori asemel graafika kaardi protsessoril. Tehnoloogia mida selleks kasutati kannab nime CUDA ning on välja töötatud NVIDIA korporatsiooni poolt. Programm, mida optimiseeriti kannab nime Molekel ning on avatud lähekoodiga.

Selle töö käigus kasutati CUDA tehnoloogiat, et optimiseerida arvutus, mis ühes punktis kestab 44,664 sekundit arvutuseks, mis kestab 0.380 sekundit. Sellega saavutati umbes 117 kordne kiirendus.

Teisel arvutil, millel on sarnase võimsusega protsessor muudeti 39,158 sekundit 0.830 sekundiks. Sellega saavuti umbes 47 kordne kiirendus.

Saavutatud tulemus annab aimu sellest, et keskmiste oskustega CUDA programmeerija on suuteline andma programmi võimsusele arvestatava võimenduse, aga ekspert on tõenäoliselt suuteline palju enamaks. Sellele lisaks tuleb veel arvesse võtta kasutatud riistvara ning töö koorem, mida on vajalik kogu arvutuse läbimiseks, sest saadud tulemus ei ole võimalik igasuguse töö koormaga. Mida väiksem on töökoorem, seda väiksem on ka saavutatud kiirendus.

Table of abbreviations and terms

CUDA	Compute Unified Device Architecture. A parallel computing platform and application programming interface (API) model created by NVIDIA.
GPU	Graphical processing unit. The brain of a graphics card.
CPU	Central Processing Unit. The brain of a computer.
TFLOPS	Tera floating point operations per second. Tera means a figure of 10^{12}
OpenACC	Open Acceleration. Is a programming standard
OpenGL	Open Graphics Library. Is an API for rendering 2D and 3D vector graphics
API	Application Programming Interface. Is a set of protocols, routines and tools for building software applications
OpenCL	Open Computing Language

1. Introduction

Parallel computing is nothing new. In fact it has been around for a long time and has helped create applications that would be very difficult to make otherwise. For example we would not have the web and database servers in the shape as we know them without parallel computing. Scientific computations which take a long time to compute are also something that have been parallelized to the CPU much longer than 10 years. However much less is spoken about converting the computations to parallel programs that run on the GPU instead.

The graphics card is a part of the computer that handles everything visual that the user sees. It is used for video games, making and watching films and for every other graphical program we can think of. As it turns out a GPU is a powerful parallel processor and that is the reason why it is capable of processing all the workload that visual applications require. Due to the fact that it is possible to write programs to the GPU we might find ourselves asking a question: is it possible to use it for something other than playing computer games? Is it possible to use the GPU to optimize work intense scientific computations?

In this work we will be optimizing a certain functionality in a program known as Molekel. The computation that this functionality has can take up to 40 seconds or more to complete. Due to the fact that Molekel runs in a single thread the easiest way to improve it is to make it parallel on the CPU which would most likely improve performance but probably will not provide a 100 time speed-up which is the aim of this work. Instead we will be using a technology known as CUDA which is developed by the NVIDIA Corporation. This technology allows us to write programs to the graphics card with relative ease, but only works on NVIDIA graphics cards which is the downside. However it is expected to deliver the results we wish to achieve.

It is important to note that GPU computing is not a “Silver Bullet” and it is not capable of solving all the problems of the field. However the problems that it can solve usually get solved well, in other words the computations that can be parallelized and ported to the GPU usually gain a remarkable performance upgrade. It has been done before and it is exactly what we will be doing to Molekel.

In chapter 2.1 we will be learning what is CUDA and where it came from.

In chapter 2.2 we will be viewing the alternatives to CUDA. We will take a brief look at 3 reasonable technologies that could be used instead of CUDA.

In chapter 2.3 the author of the work will comment the choice of technology from his perspective.

In chapter 2.4 we will learn what are the options of using CUDA? We will talk about prebuilt libraries, compiler hints and writing custom code. We will also explain why we chose to write custom code for this work.

In chapter 2.5 we will familiarize ourselves with the most important concepts of writing custom CUDA code. We will look at how threads and GPU architecture works, what types of memories exist and what can be parallelized and what cannot. After that we will introduce Molekel in chapter 2.6.

In chapter 3.1 we will introduce the problem with Molekel that we will be solving in detail. There we will be talking about the problematic functionality and its properties.

In chapter 3.2 we will be presenting the solution to the problem both inside and outside the code. We will learn what it takes to get the base code of Molekel compiling without problems and adding CUDA support. After that we will be talking about the problems in the code and program architecture and how we solve these.

Chapters 4 and 5 contain the results of the optimization and a conclusion of the circumstances that made those results possible.

2. Background

It is a known fact that it takes a single skilled worker to build a house much longer than 10 workers all possessing the same skill or a little bit lower skill as the first. The 10 workers will each have to do more work than the single worker because they need to communicate and make sure that the details they made fit with the details that their coworkers made. However the house is still built in a shorter timeframe. This principle has been used throughout history to make big things happen.

The same principle applies to parallel computing regardless of the device that we are doing it on. We could be using human beings to compute something or a CPU or even a GPU. The problem with humans is that they are slow when compared to a CPU. We could be a using a group of humans but they will still probably do the computation slower than the CPU even if the CPU has less cores than the human group has workers. The GPU on the other hand has many cores to do the work. It might be possible that even more cores than a human group we are comparing it to. However each core does the work faster than a person and in some cases perhaps even as fast as a CPU core.

The GPU relies on the principle that we described earlier and due to that has attracted our attention in hopes that we can do something remarkable with it. Our choice to program on the GPU will be a technology known as CUDA.

2.1 What is CUDA?

CUDA is an abbreviation that stands for Compute Unified Device Architecture. It is a parallel computation platform that is supported by most NVIDIA graphics cards released after 23rd of June 2007, which is the official release date of the first CUDA toolkit. CUDA also stands for a programming model for writing parallel programs on NVIDIA GPUs supporting the said platform.

The point of CUDA is to make general purpose computing on graphics cards simpler and accessible to a wider audience of developers. Do not misunderstand this, general purpose parallel computing on graphics cards has been around longer than CUDA, however what CUDA does is provide everything needed in 1 package and with a simple API. CUDA works on Windows, Linux and Mac.

2.2 Alternatives

Now we will present the 3 alternatives that were considered before the start of this work. Each of them have their positive and negative sides and we will compare them briefly to the CUDA technology. After that the author will comment them on his own perspective.

2.2.1 OpenGL

Before CUDA [1], one of the things used to do scientific calculations on GPUs was and still is OpenGL [2]. However this technology presents a challenge: the developer has to translate the scientific problem into triangles and pixels and then program it using various libraries of OpenGL. It is important to know that OpenGL is a graphics programming tool and it is designed that way. If we want to program other things, then we have to be creative with what we have and we have to understand it well. However if we get all of this right then everything works without problems. On the other hand what a CUDA programmer sees is that it allocates some memory on the graphics card, copies the needed data there and then launches the necessary amount of threads to do the work and that is it. Description of this alternative is based on the author's experiences and on the official pages of OpenGL and CUDA. More about it can be read on the official CUDA website [1] and it is also briefly mentioned in the official parallel programming course by NVIDIA [3].

2.2.2 OpenCL

Another alternative to CUDA is OpenCL [4]. Released later than CUDA toolkit, it provides parallel computing on a variety of platforms rather than just NVIDIA GPUs. In fact it does not focus on only GPUs The performance isn't that different, however OpenCL is made out of packets released by different vendors when CUDA is purely lead by NVIDIA. This is the cause of many driver problems with OpenCL, because different vendors tend to interpret OpenCL specifications differently. This topic is discussed more in depth in an article written by Vincent Hindriksen [5].

2.2.3 DirectCompute

DirectCompute [6] is a product from Microsoft that makes use of GPUs supporting DirectX10 or higher. Due to this it is only available on windows. However it does not suffer problems with it's drivers like OpenCL [4] does and works as expected. Unfortunately it is missing quite a few features that CUDA [1] and OpenCL have, most importantly dynamic parallelism [7]. These topics are covered more in depth in articles written by Rahul [8] and Vincent Hindriksen [9].

2.3 Author's choice

Now it's time to talk about why I chose CUDA to learn about the world of parallel computing on GPUs.

I found out about the technology when I was examining the capabilities of my new computer. At first this technology did not look very tempting, but closer examination revealed an interesting field. I knew that doing time consuming scientific computations on powerful parallel processing CPUs was nothing new, but I never thought about doing it on the GPUs.

The documentation was easy to read and there was a full course that explained how the technology works. Installing and configuring the developer tools was rather straight forward and did not require me to find any extra information from third party sources. It became clear quickly that it was easy to get into the technology and the learning curve wasn't steep either.

After I had learned the basics I started to look for alternatives, because I was keen to know if there was something better. I had done OpenGL in the past so I knew that I could use that for these kind of things as well, but it definitely was not going to be easy. OpenGL has a documentation that I find difficult to process and it is composed of several packages released by different vendors. Also I do not know of a full OpenGL tutorial that explains most of it and is not deprecated. Setting up your development environment is also time consuming, so I decided to avoid that.

OpenCL was something I really considered, but after I found out that setting it up was similar to OpenGL, I decided to go down the easy path. However if one can get past setting it up, then it is a considerable alternative to CUDA, because the performance doesn't seem to be that different and it can run on other GPUs than NVIDIA GPUs. Just like CUDA it also works on Windows, Linux and Mac.

DirectCompute made me careful when I found out that it was only for Windows. The documentation was not exactly tempting. It also lacked a number of features including dynamic parallelism, which is one of the most interesting features I have found about parallel computing. This showed that DirectCompute is not a viable choice for this work.

To conclude this: I chose CUDA because it is documented well and is simple to learn. The development environment is something that can be pulled out of the box and it works. It enables you to do all the things you want if you have the graphics card that supports it.

2.4 How can we use CUDA?

Now that we know what CUDA is, we will talk about how we can use it in our programs to speed up their performance. We will cover the 3 ways that are introduced in the official CUDA developer training website [10] and we will start with the easiest and end with the most difficult way of utilising CUDA.

2.4.1 Use a prebuilt library

The easiest and fastest way to get CUDA support to your application is to use a prebuilt library which can be found from NVIDIA’s official list [11] or around the internet. Getting support this way usually means replacing a single library file in the program. After the replacement some changes and a recompilation may be needed, but most of the time it is not. For example when the user has the proper hardware then getting a faster video rendering speed with Adobe Premiere Pro [12] is just a matter of replacing the correct .dll file. When provided a system with the specifications that are described in chapter 4 after Table 1, then the average speedup tends to be 8 times when using the GPU. The actual figures from the experiment were 4 hours rendering when using only the CPU and 30 minutes rendering when using help of the GPU.

2.4.2 OpenACC Directives

In cases where the user cannot find the proper library to fit into their application, but when rewriting the program is out of the question then it is possible to use OpenACC Directives [13] to achieve GPU acceleration. This method is about putting hints for the compiler around the program’s code where it is possible to parallelize things. Then the code is compiled with an NVIDIA’s compiler which uses the hints to make the given code GPU accelerated. This will most likely involve changes to the code besides the compiler hints but those changes are usually minimal. The official OpenACC guide [14] suggest an average of 3 to 6 times speed-up.

2.4.3 Writing custom CUDA code

If the goal of adding CUDA support to the program is to optimize the program as much as possible, then usually the only way to do so is to write custom CUDA code. There is no special language for writing CUDA code, instead there are libraries for C++, Fortran, Java and Python. So in order to use this technology the programmer only has to learn to use the CUDA API, provided that they already know how to use one of the previously stated programming languages. To help programmers understand the API there is the official CUDA documentation [15], there is a course that introduces the concepts of parallel programming using CUDA [3] and there are various third party forums and webpages like Stack Overflow [16] where it is possible to receive help when the official sources fall short.

2.4.4 The choice for this work

The choice for this work is the third option: to write custom CUDA code. The reason is that the aim of this work is to learn if we can achieve a 100 time speed-up of the program and in what cases is that possible. The average speed-ups given in the first 2 options are clearly not enough to meet the set criteria, therefore we have only 1 option left and that is to rewrite a part of the program that we are trying to optimize.

2.5 How does custom CUDA code work?

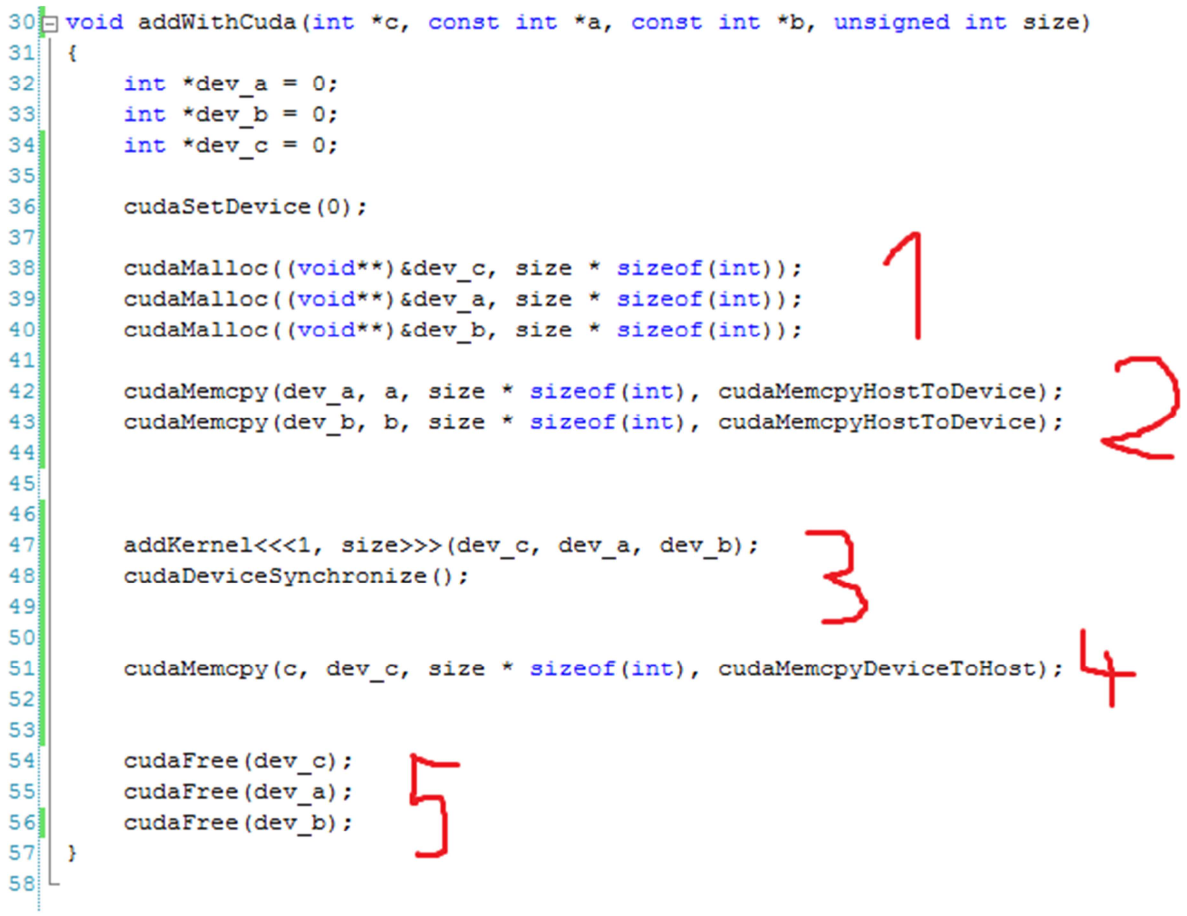
Before we begin introducing Molekel, it’s problems and solutions to those problems, it is important that we know how CUDA works. In the following chapters we will briefly look at the core principles of using CUDA with a small example. However we will only look at the program structure and some important GPU properties. We will not cover setting up the development environment. For those details please refer to the official CUDA documentation [15].

2.5.1 CUDA program structure

A typical CUDA program is made out of 5 parts:

1. Allocate memory on the GPU
2. Copy all necessary data to the GPU
3. Run the computational program on the GPU
4. Copy the results from the GPU to the host memory
5. Free all allocated space on the GPU

An example of how the program looks like in C++ code can be seen in Code Snippet 1:



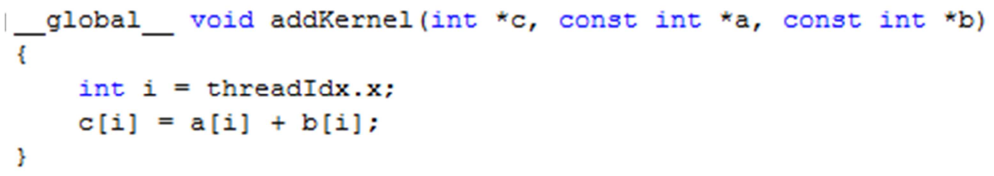
```
void addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;

    cudaSetDevice(0);

    cudaMalloc((void**)&dev_c, size * sizeof(int));
    cudaMalloc((void**)&dev_a, size * sizeof(int));
    cudaMalloc((void**)&dev_b, size * sizeof(int));

    cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);


    addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
    cudaDeviceSynchronize();


    cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);


    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
}
```

Code Snippet 1

In Code Snippet 1 we can see a simple program that takes 3 arrays as an input, then sums the corresponding elements of the last 2 arrays and writes the results in the first array(See picture

2 below for the actual operation). Each of the previously stated CUDA program structure parts are enumerated to make them stand out. It is important to note that the actual computation takes place in part 3 and the other 4 parts are simply supportive parts. The amount of the supportive code of the program depends on the complexity of the data structures that the computation uses. This is caused by the limited number of ways that we can move data between the CPU and the GPU. Mostly our tools are cudaMalloc, cudaMemcpy and cudaFree. If the given data structure cannot be copied with a single cudaMemcpy operation then the supportive code will grow rapidly.

It is important to know that the shown code only runs on the CPU. The code that runs on the GPU is presented on Code Snippet 2, where we can see a CUDA kernel: a function that the CPU commands the GPU to run.



```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

Code Snippet 2

As a contrast we can see that the code on the GPU is very minimal and reflects the nature of the operation, which is very trivial. All that is done here is the thread that runs this kernel calculates its index and then uses it to find the right data to do the calculation and write the result. It seems strange that we have to do a lot of work just for a simple operation. The next chapter will explain what we gained from this.

2.5.2 All Kernel instances run in parallel

The kernel we saw in Code Snippet 2 is launched in a set of instances called threads. Each instance runs independent from the others and in parallel. That is what we will win from this kind of program. The work we want to do is not done in a serial fashion, but in a parallel fashion. There may not be much gains when we have few array elements for example 5. But when we have 1000 elements in each array or 1 million elements, then it is already possible to distinguish the speed-up factor.

The threads are launched in a random order, but they are gathered into blocks, which can hold a maximum of 1024 threads and have 3 dimensions. The blocks are gathered into grids which can span up to 65535 X 65535 X 65535 blocks on 3 dimensions. This data is based on multicompute capability version 2.0 which is the oldest supported version. In Code Snippet 1 we launched the kernel in a 1 block grid using variable size to determine the number of threads in a block.

Each thread always knows where it is located inside a block and where its block is located inside a grid. This system of indexing is what we will be using to locate our computational data.

2.5.3 GPU architecture and its purpose

A high end CPU like the Intel i7 4930k has 6 cores that make up the processing unit. A high end graphics card like the NVIDIA gtx 780TI has 2880 CUDA cores, which are formed into 15 streaming multiprocessors, which in turn make up the processing unit. It is obvious that the CUDA cores do not have all the functionality and speed of a single CPU core, but what they lack in ability, they make up in numbers. The GPU is not designed to process a single instruction at the fastest speed possible, that is the job of the CPU. The GPU is designed to process a group of instructions at the fastest speed possible and it does that by dividing the workload between the cores and having each part of the work be processed in parallel. With that in mind, launching millions of threads is something that we will not do on a CPU, but it is a perfectly normal thing to do on a GPU.

2.5.4 How threads run on the GPU

In chapter 2.5.2 we learned that threads are grouped into blocks and blocks are grouped into grids. In chapter 2.5.3 we learned that the GPU CUDA cores are grouped into streaming multiprocessors also known as SMXs and those in turn make up the GPU. From this statement one might assume that the threads in a single block run on a single SMX and the blocks of a grid are distributed among the SMXs of the GPU. This is indeed the case and there are some important facts we must know before proceeding.

All threads from 1 block run on a single SMX and have no contact with other SMXs. This means that threads from different blocks have no way of communicating with each other even though it is possible that several blocks may run at the same time on the same SMX if the GPU can fit them there. However it is possible to synchronize the threads of a single block in relation to each other and even have them share memory.

2.5.5 GPU memory types

Individual cores do not have a dedicated cache, instead they use the cache of the SMX. This limits the number of threads and blocks that can run on a single SMX because each thread needs to store indexing and programmer specified computational data in the cache.

Furthermore there are 3 different types of memory on the GPU.

1. The fastest and the smallest is the local memory which is a cache in terms of the hardware. This is the place where the threads store their private data. That is all the indexing data and all the instanced variables within the kernel.

2. The second fastest and pretty much the same size is the shared memory which is slower than the local memory, but is also cache based and allows the threads of a single block to use it. It is also defined in the kernel, but has a special tag that makes it stand out for the compiler.

3. The third memory is the slowest and largest and is the GPU global memory. That is what we mean when we talk about how much memory our GPU has. It may be slower than the first 2 memory types but it is still remarkably faster than the RAM memory that the CPU has access to. This is the memory that cudaMalloc and cudaMemcpy move data to.

It is possible to access data in the RAM memory when the kernel is running, but this is a very costly operation in terms of time. This is one of the reasons why we copy the data onto the GPU when we run our computations.

It is important to note that the cores on the GPU can do computations at an extremely fast rate. If we are to use the full power of these cores then we must constantly feed them data to work with. The general formula of success is to spend most of the time computing and minimal time accessing memory. This does not mean that we have to limit the number of memory operations, we just have limit the time it takes to make those operations. Therefore if a piece of data is used more than once, then it might be a good idea to move it from a slower memory type to a faster memory type.

2.5.6 What can be parallelized?

We may know how the syntax of CUDA works, but if we are to optimize an existing program we must also be able to identify code that can be parallelized and code that cannot. The easiest code to parallelize are loops or nested loops, whose iterations are not dependent on each other. We just make each iteration a separate thread, write the indexing logic and that concludes the work.

The second case are loops or nested loops whose iterations are dependent on another random iteration. Usually this case is represented by a mathematical operation where a set of values are converted into a single value through an operation where the order of operands is irrelevant like adding or multiplying. This kind of operation usually requires thread synchronisation and atomic operations. When dealing with more complex parallel algorithms like sorting, then it is difficult and often even impossible to implement them without any synchronisation at all.

The algorithms that cannot be parallelized usually represent a clear serial fashion like loops or nested loops where every iteration is dependent on the results of its next or previous iteration.

2.6 What is Molekel?`

Molekel [17] is a molecular visualization program. It is used to simulate different kinds of situations in the field of chemistry and display them in a visual fashion to the user. The tools that the user has available to this task are numerous. Molekel also has an array of options for processing and saving the experiment data.

Molekel can run on Windows, Linux and Mac and it is open source which means that anyone can contribute to the program and distribute it freely.

The gallery of Molekel [18] demonstrates some of the said possibilities of the program.

3. The Problem and Solution

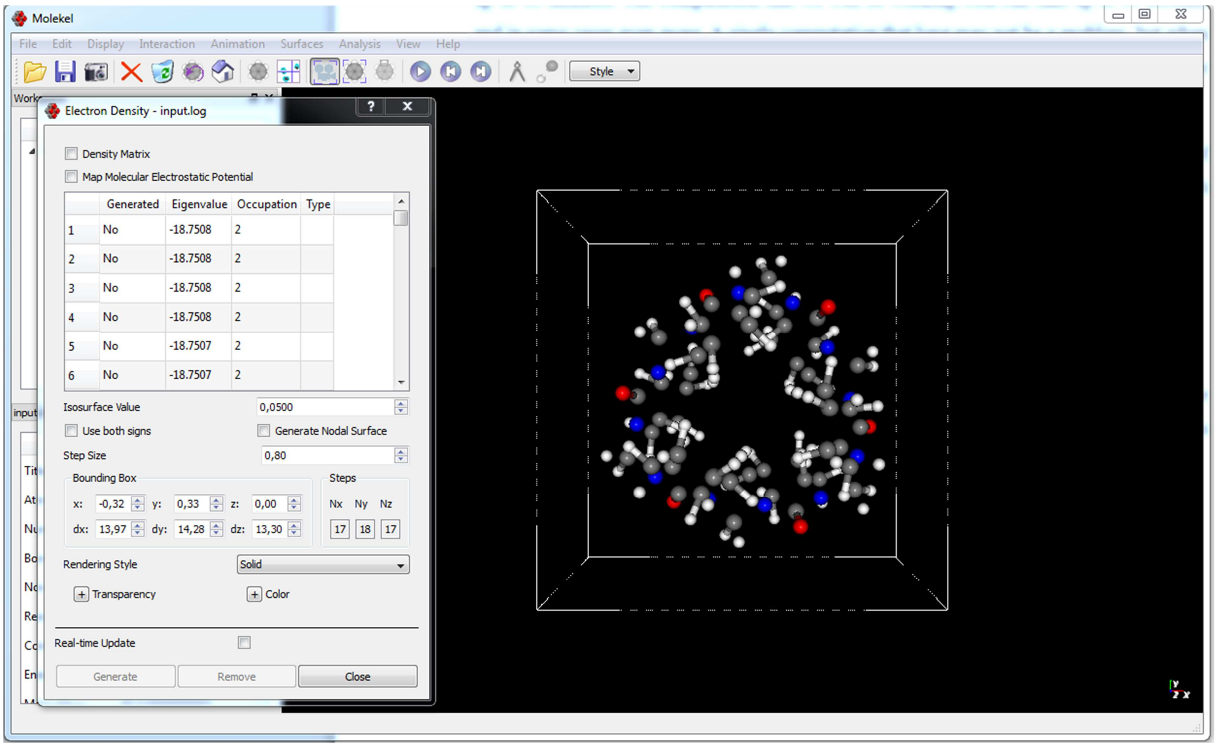
The problem that we will be solving in this work is related to one of the functions in Molekel. As stated in chapter 2.6 Molekel has a wide variety of tools to manipulate the molecular simulation. However when the user wants to receive extremely accurate data then simulation computation speed increases exponentially. It is confirmed that some computations can take up to 40 minutes. The computation that we will be working with can take up to 44 seconds and in some cases even more. A single computation that long may not be a problem, but when the user has to make many such computations then it will remarkably degrade the user experience of the program.

The solution that we are looking for this problem will be to decrease the calculation time of the said computation by a factor of 100 times. We will be examining under which circumstances it is possible to do this and under which it is not.

3.1 The Problem

The function that we will be optimizing and referring to as a problem can be reached in the program if the user selects Surfaces from the upper menu bar and then Electron Density from the menu that opened up. This will cause a dialog box to appear that is like the dialog in Schema 1. The computation is run by selecting a row from the scroll box and pressing generate. The rest of the options can be a little disorientating at first, however there are only 4 important variables that we must keep an eye on. The most import of them is the Step Size which is currently 0.80.

The step size variable controls how accurately the simulation is run. The smaller the step, the better. However decreasing the step size increases the values of Nx, Ny and Nz variables whose product is the amount of iterations that must be processed by an internal algorithm that computes the required data. Increasing the iterations obviously increases the time it takes to complete the computation. For example at step size 0.15 we have to complete 786 315 iterations and it takes 44.949 seconds to do so. This is our problem.


Molekel
File Edit Display Interaction Animation Surfaces Analysis View Help
Style
Electron Density - input.log

Density Matrix

Map Molecular Electrostatic Potential

	Generated	Eigenvalue	Occupation	Type
1	No	-18.7508	2	
2	No	-18.7508	2	
3	No	-18.7508	2	
4	No	-18.7508	2	
5	No	-18.7507	2	
6	No	-18.7507	2	

Isosurface Value 0,0500

Use both signs Generate Nodal Surface

Step Size 0,80

Bounding Box
x: -0,32 y: 0,33 z: 0,00
dx: 13,97 dy: 14,28 dz: 13,30

Steps
Nx 17 Ny 18 Nz 17

Rendering Style Solid

Transparency Color

Real-time Update

Schema 1

3.2 The solution

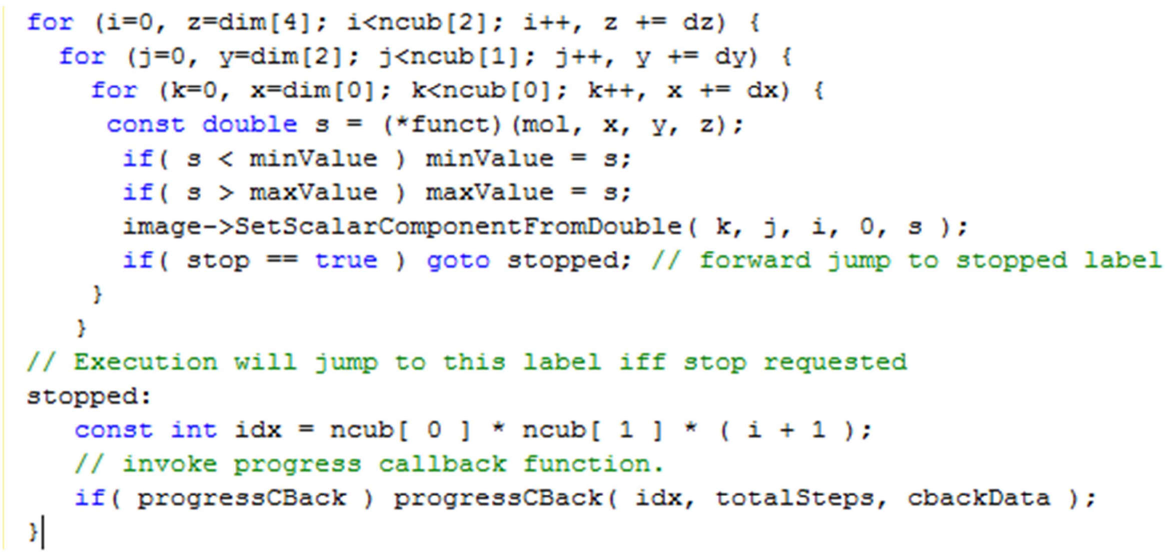
The solution to our problem is to make all those iterations run in parallel. Due to the fact that Molekel is entirely a single threaded application we will most likely gain a speed boost, however implementing the solution is not as straight forward as it might seem. There are numerous sub problems that need to be solved within the code and before the coding even starts.

3.2.1 Challenges before the coding

Molekel is an open source program which means that getting the code is not a problem. However what quickly becomes a problem is the program's dependencies and their configurations. Molekel has instructions on how to setup the work environment to compile the program, but these instructions are not entirely accurate and do not cover everything. Versions of a few dependencies like Qt were the biggest issues, because there were many versions and the newest ones did not fit with the program. For example Qt 4.4 seems to be the only one that fits with Molekel without breaking anything. It took a long time to get to that version because the versions that were between 4.4 and the newest 5.4 all had to be compiled and tested. Fortunately there was a place where most of the dependencies were precompiled and properly configured to allow Molekel itself to compile and run without errors. Adding CUDA support to the project took time as well, but that time is insignificant compared to the time it took to get the base program compiling and running without problems. In total it took 8 weekends to properly setup every single aspect of the working environment. That is the price to pay if there is no previous experience in working with projects this size.

3.2.2 Challenges in analysing the code and planning work

The challenges in finding the time consuming sections of the code were modest. It was clear that we were looking for loops and possibly nested loops, so it did not take long to reach the code that can be seen in Code Snippet 3.

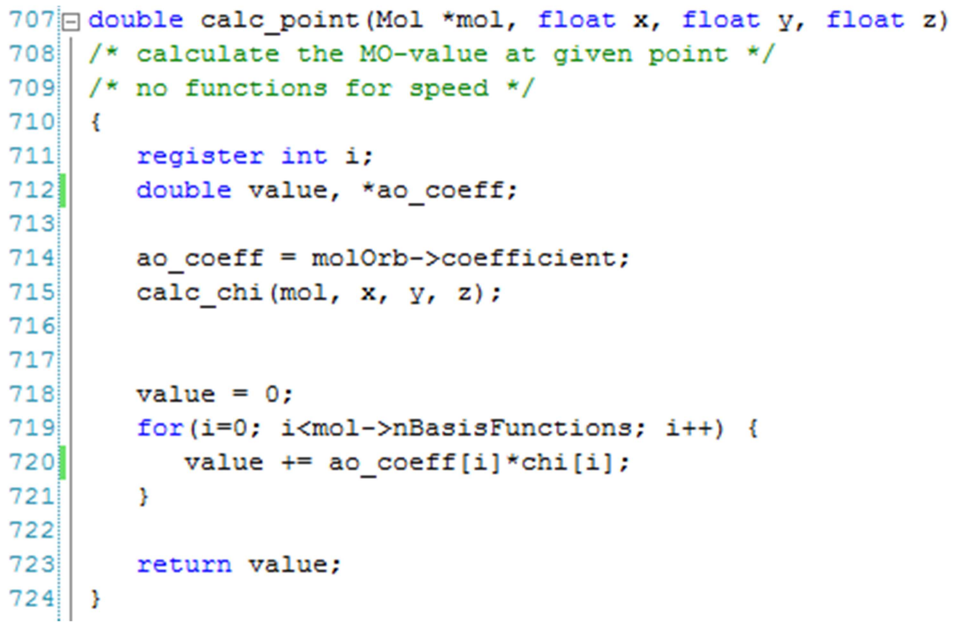


```
for (i=0, z=dim[4]; i<ncub[2]; i++, z += dz) {
  for (j=0, y=dim[2]; j<ncub[1]; j++, y += dy) {
    for (k=0, x=dim[0]; k<ncub[0]; k++, x += dx) {
     const double s = (*funct)(mol, x, y, z);
      if( s < minValue ) minValue = s;
      if( s > maxValue ) maxValue = s;
      image->SetScalarComponentFromDouble( k, j, i, 0, s );
      if( stop == true ) goto stopped; // forward jump to stopped label
    }
   }
// Execution will jump to this label iff stop requested
stopped:
   const int idx = ncub[ 0 ] * ncub[ 1 ] * ( i + 1 );
   // invoke progress callback function.
   if( progressCBack ) progressCBack( idx, totalSteps, cbackData );
}
```

Code Snippet 3

Here we can see 3 nested loops, a call to a function pointer that receives its value from a switch statement that precedes this code, variables minValue and maxValue that have nothing to do with the computation in question and writing results to an image object, which will display the data to the user. This is the moment to realize that all the iterations of these loops are independent which means that we can make this nest of loops parallel. In our case the function pointer resolves to function calc_point which can be seen in Code Snippet 4. This function will make the body of our kernel that we will write.

When looking closely at function calc_point we can see that it contains another loop, which would be the fourth loop of the nest. Unfortunately we can not parallelize this loop due to the fact that function calc_chi is a serial function and in the given circumstances it is wiser code the adding operation inside calc_chi to make it operate in the local memory rather than write intermediate results in the global memory. If we cannot make it parallel, then at least we can make it operate in the fastest memory available to us. This will improve our results. Due to the fact that calc_chi is a very long function it can be reached in the github repository [19] inside the file src/old/calcdens.cpp. The new GPU friendly function will be in the file src/CudaCodes/CalcChiCalcPoint.cu which essentially adds together the functionalities of calc_point and calc_chi.



```
double calc_point(Mol *mol, float x, float y, float z)
/* calculate the MO-value at given point */
/* no functions for speed */
{
   register int i;
   double value, *ao_coeff;

   ao_coeff = molOrb->coefficient;
   calc_chi(mol, x, y, z);


   value = 0;
   for(i=0; i<mol->nBasisFunctions; i++) {
      value += ao_coeff[i]*chi[i];
   }

   return value;
}
```

Code Snippet 4

It is important to note that the given code is written in a period of 10 years by different authors, whom all seem to have different styles in writing and naming code. This makes orientating in the code difficult. The biggest problems arise in the older parts of the code where all the computational logic is located. The functions and variables are mostly named using abbreviations which makes understanding the purpose of individual algorithms rather difficult. It is clear how the algorithms work and how to optimize them, but it is unknown why they have the shape that they have.

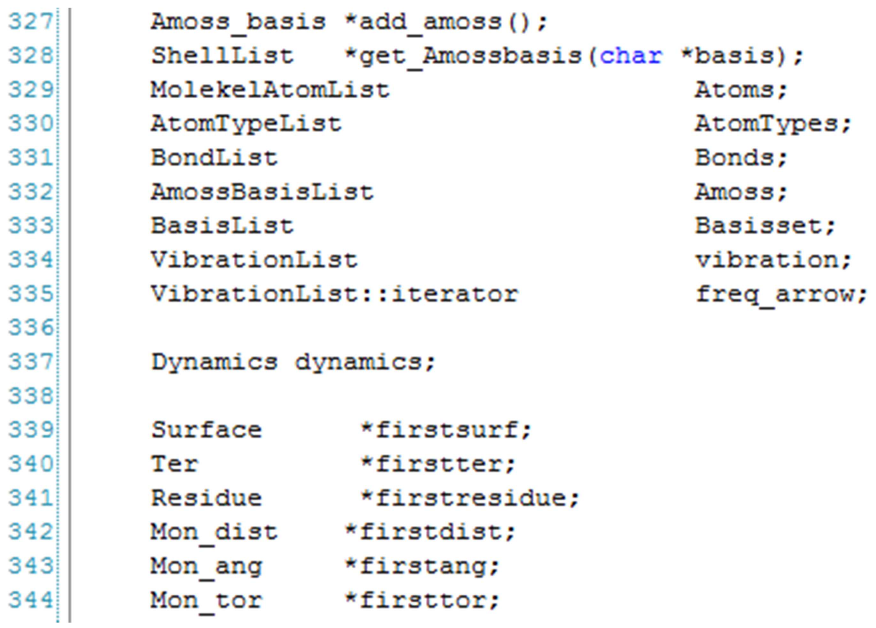
3.2.3 Challenges in writing the code.

The technical debt that Molekel has played a remarkable role when writing the CUDA extension. It was impractical to rewrite the whole program, so we had to use what could be used. For example: the general structure of the function vtk_process_calc that initiates the electron density computations in Molekel is as follows:

1. Initiate variables
2. Check input variable errors
3. Select the appropriate function to do the computation
4. Do the computation.

Due to the fact that all the calls for electron density computations end up reaching this function then it might be a good idea to add an extra step between steps 2 and 3 that call for a CUDA object that has its own selection of functions. When it is able to select a function which can return a proper image object then we terminate the vtk_process_calc function here. Otherwise we continue with the old program that is not optimized. This is good if we are actively developing the program while there are people already using it. It also removes the need for rewriting all the references to this function.

The next challenge was getting all the required computational data to the graphics card. Porting the calc_chi function was one problem, but getting structs Molecule and MolecularOrbital onto the graphics memory in 1 piece was another.



```
    Amoss_basis *add_amoss();
    ShellList   *get_Amossbasis(char *basis);
    MolekelAtomList                  Atoms;
    AtomTypeList                     AtomTypes;
    BondList                         Bonds;
    AmossBasisList                   Amoss;
    BasisList                        Basisset;
    VibrationList                    vibration;
    VibrationList::iterator          freq_arrow;

    Dynamics dynamics;

    Surface      *firstsurf;
    Ter          *firstter;
    Residue      *firstresidue;
    Mon_dist    *firstdist;
    Mon_ang     *firstang;
    Mon_tor     *firsttor;
```

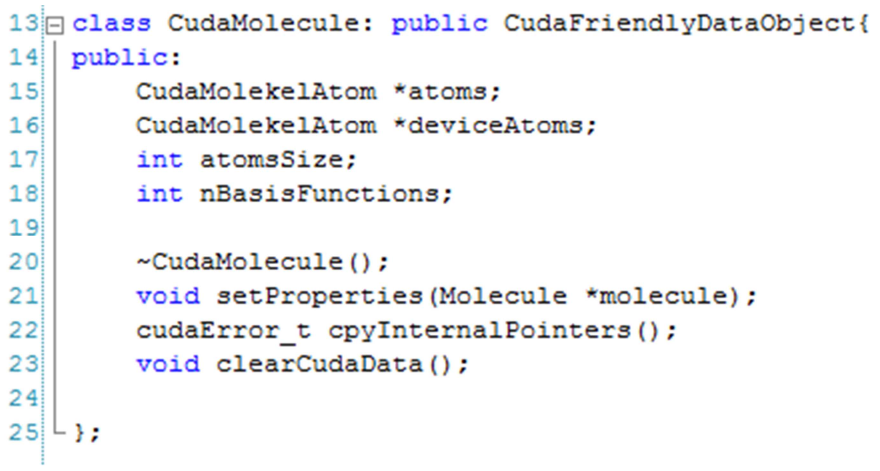
Code Snippet 5

When we look at a part of struct Molecule in Code Snippet 5 then we can see that there are a number of pointer variables which will be pointing at various arrays and linked lists when the program is running. Normally this will not be a problem because we would be operating in a single memory chunk. However in this case we need to use cudaMalloc and cudaMemcpy to move the data to the graphics card and then make sure that all the pointers we access on the graphics card point to the GPU memories rather than towards the RAM memory. So when we have a variable atoms, which is a linked list of object Atom, then first we must convert the linked list into an array, copy that array to the graphics card, set the atoms pointer in the Molecule struct to point to the corresponding graphics card memory address and then copy the rest of the Molecule object.

This method of working with pointers applies to any object. If there are pointers that belong to an object, then the data that those pointers point to must first be copied to the graphics memory, then those pointers must be changed to point to the graphics memory and then the rest of the object can be copied to the GPU memory. The real challenge begins when a pointer points to an object, which in turn has pointers that point to other objects which in turn have pointers and so on.

This brings us to the reason why a linked list must be converted to an array before a copy operation is made: it is faster that way. When we turn a linked list to an array we must copy every single element to another location in the memory, so at first sight we might as well do a cudaMempy operation for each linked list element. However what we might forget is that now we will have to do 2 additional operations for each container which is setting the pointers of the previous and next element to point to the corresponding elements in the GPU memory. Suddenly it is clear that trying to move the linked list onto the graphics card takes more time and if the pointers we need to overwrite are inaccessible from outside the object then this operation cannot be done. Let us not forget that when we start freeing the allocated memory we need to do an opposite operation.

The problem with pointers lead to the creation of a more “CUDA friendly” version of the Molecule object that has 2 pointers for each array: one for RAM memory and another for the graphics memory and that has the methods to move its contents to the graphics card. Its definition can be seen in Code Snippet 6.

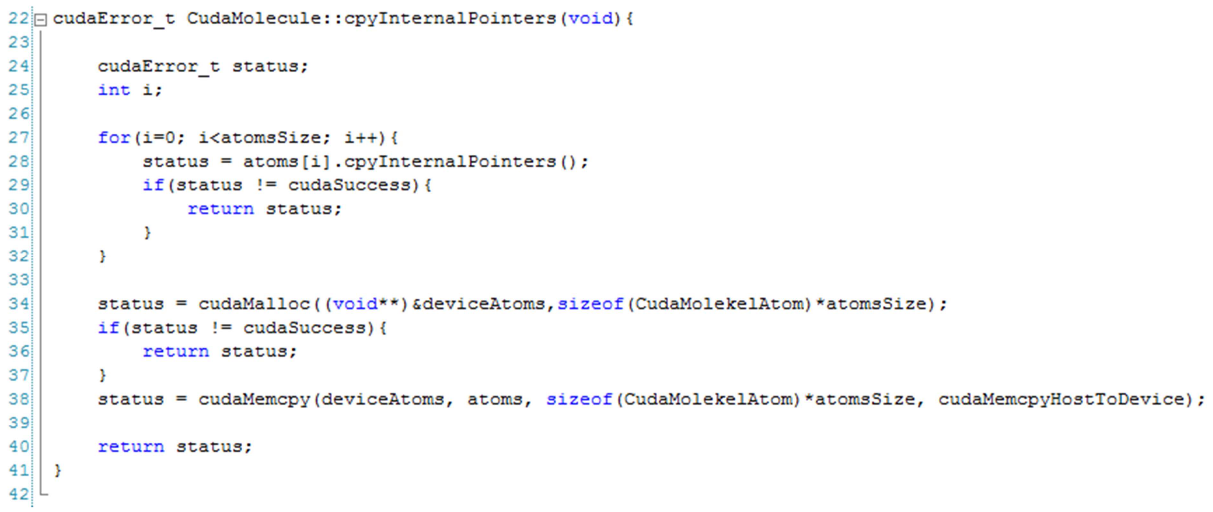


```
class CudaMolecule: public CudaFriendlyDataObject{
public:
    CudaMolekelAtom *atoms;
    CudaMolekelAtom *deviceAtoms;
    int atomsSize;
    int nBasisFunctions;

    ~CudaMolecule();
    void setProperties(Molecule *molecule);
    cudaError_t cpyInternalPointers();
    void clearCudaData();

};
```

Code Snippet 6

Clearly not all variables of Molecule made it into this object. The reason is that we do not need all of them for this particular functionality. The object will obviously grow as the program is further developed, but at this state it is all we need.

Furthermore what we can see from this definition is that there is a method for setting all the properties from the original Molecule object, a method for copying the internal pointers and their data to the graphics card and a method for clearing all the copied data from the GPU. The last 2 methods are overridden from an interface CudaFriendlyDataObject. The methods themselves are not very complex as we can see from Code Snippet 7.

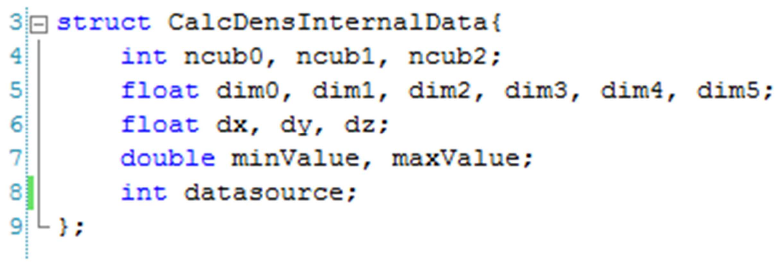


```
cudaError_t CudaMolecule::cpyInternalPointers(void){

    cudaError_t status;
    int i;

    for(i=0; i<atomsSize; i++){
        status = atoms[i].cpyInternalPointers();
        if(status != cudaSuccess){
            return status;
        }
    }

    status = cudaMalloc((void**)&deviceAtoms,sizeof(CudaMolekelAtom)*atomsSize);
    if(status != cudaSuccess){
        return status;
    }
    status = cudaMemcpy(deviceAtoms, atoms, sizeof(CudaMolekelAtom)*atomsSize, cudaMemcpyHostToDevice);

    return status;
}

```

Code Snippet 7

However because Molecule has Atoms, Atoms have Shells and Shells have Gaussians then we suddenly find ourselves on top of a tree like data structure where every branch must be copied separately. Making all those 4 objects implement the CudaFriendlyDataObject interface seems to solve the problem. The object MolecularOrbital was solved in an identical way.

In addition to that we also had to make a small data packet to carry the additional variables and constants which looks rather straight forward as can be seen in Code Snippet 8 . However what is important to keep in mind with all of the changed or created objects is that every single variable that was a part of the computation had to be analysed if it was really necessary and if so then made CUDA friendly. This took a remarkable amount of time.

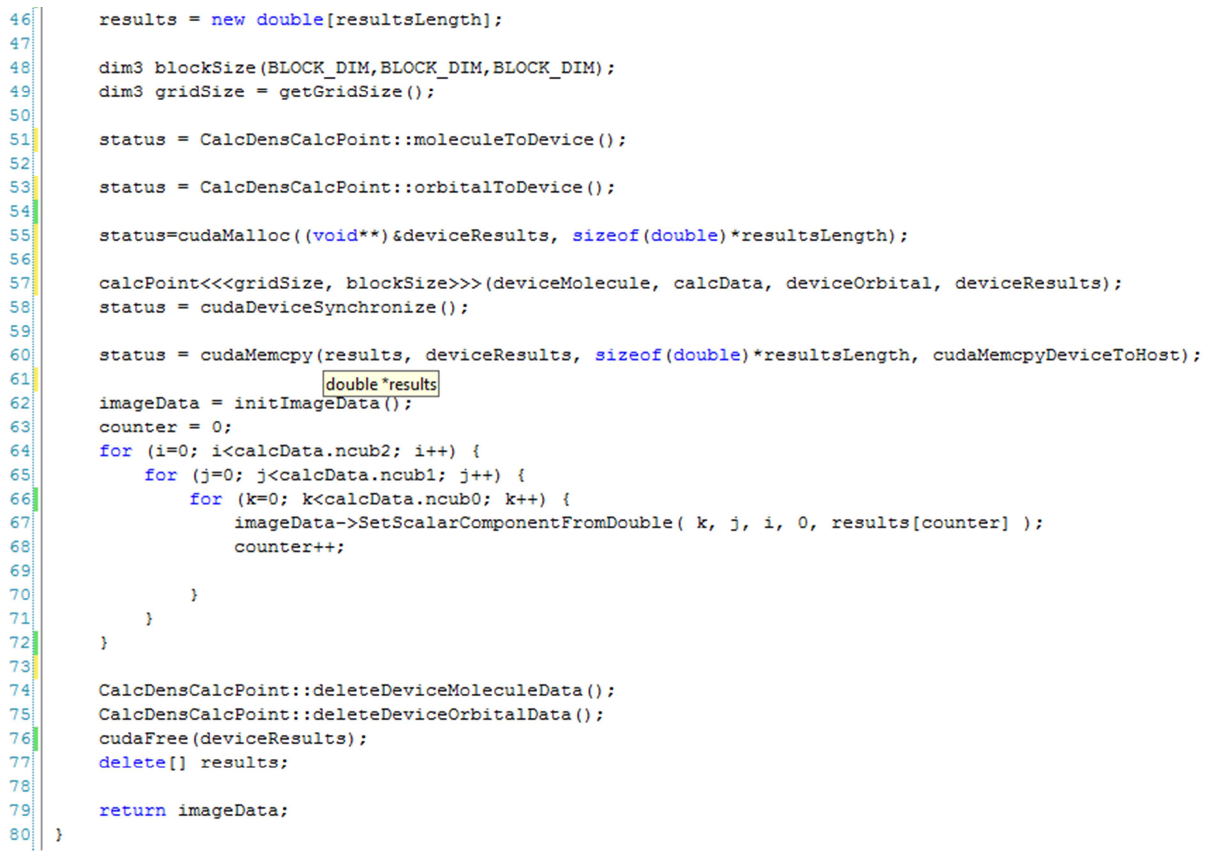


```
struct CalcDensInternalData{
    int ncub0, ncub1, ncub2;
    float dim0, dim1, dim2, dim3, dim4, dim5;
    float dx, dy, dz;
    double minValue, maxValue;
    int datasource;
};
```

Code Snippet 8

The CUDA code that orchestrates setting up the calculation data and retrieving the results can be seen in Code Snippet 9. This is the CUDA version of the code we saw in Code Snippet 3. It is important to note that the error handling has been removed in this function to provide a better overview of the actual algorithm. This is the reason why we are catching status from every CUDA function we use. However in an actual CUDA program error handling should never be ignored, because mostly CUDA functions tend to swallow the errors when they fail to carry out their tasks. When the error status is not explicitly caught and processed then the user may never know that the program has failed and the output is invalid.

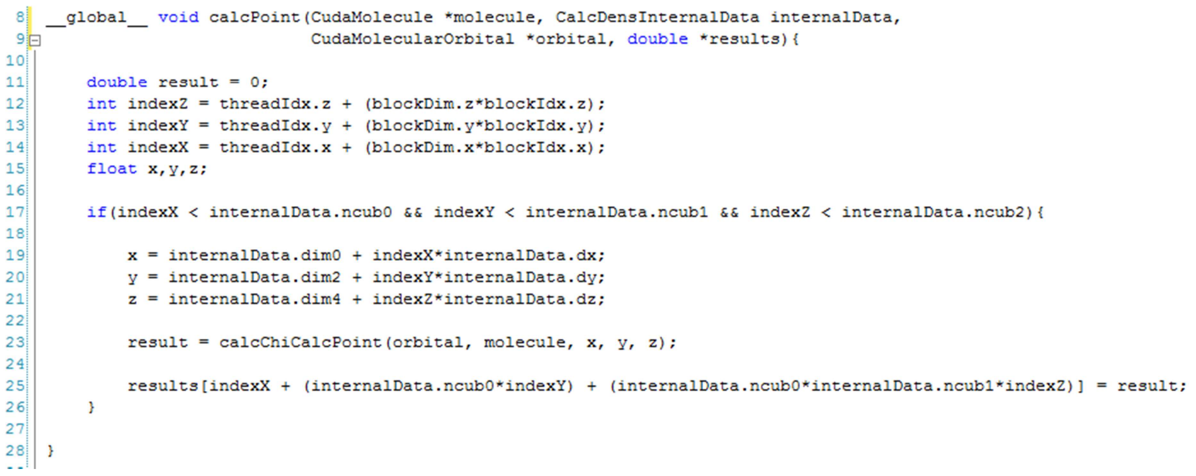
When we analyze Code Snippet 9 we tend to notice the typical CUDA program structure we saw in Code Snippet 1. First we send the calculation data to the graphics card. Here we are using functions moleculeToDevice and orbitalToDevice and a memory allocation operation for that. Then we activate the kernel function which is calcPoint. Finally we copy back the results, process them into an ImageData object that is used by the rest of the program and then deallocate all the memory space. The ImageData processing looks a lot like the old code in Code Snippet 3. However inserting the results to the object is not a very time consuming operation and due to the fact that the ImageData object is a very complex object then it is perfectly acceptable to do this operation on the CPU and not implement a GPU friendly object out of it.



```
    results = new double[resultsLength];

    dim3 blockSize(BLOCK_DIM,BLOCK_DIM,BLOCK_DIM);
    dim3 gridSize = getGridSize();

    status = CalcDensCalcPoint::moleculeToDevice();

    status = CalcDensCalcPoint::orbitalToDevice();

    status=cudaMalloc((void**)&deviceResults, sizeof(double)*resultsLength);

    calcPoint<<<gridSize, blockSize>>>(deviceMolecule, calcData, deviceOrbital, deviceResults);
    status = cudaDeviceSynchronize();

    status = cudaMemcpy(results, deviceResults, sizeof(double)*resultsLength, cudaMemcpyDeviceToHost);

    imageData = initImageData();
    counter = 0;
    for (i=0; i<calcData.ncub2; i++) {
        for (j=0; j<calcData.ncub1; j++) {
            for (k=0; k<calcData.ncub0; k++) {
                imageData->SetScalarComponentFromDouble( k, j, i, 0, results[counter] );
                counter++;

            }
        }
    }

    CalcDensCalcPoint::deleteDeviceMoleculeData();
    CalcDensCalcPoint::deleteDeviceOrbitalData();
    cudaFree(deviceResults);
    delete[] results;

    return imageData;
}
```

Code Snippet 9

After all the required data made it to the graphics card in one piece then it was time to write the logic which allowed each kernel to find the required data and write the result to the right location. How the kernel eventually looked like can be seen in Code Snippet 10.



```
__global__ void calcPoint(CudaMolecule *molecule, CalcDensInternalData internalData,
                        CudaMolecularOrbital *orbital, double *results){

    double result = 0;
    int indexZ = threadIdx.z + (blockDim.z*blockIdx.z);
    int indexY = threadIdx.y + (blockDim.y*blockIdx.y);
    int indexX = threadIdx.x + (blockDim.x*blockIdx.x);
    float x,y,z;

    if(indexX < internalData.ncub0 && indexY < internalData.ncub1 && indexZ < internalData.ncub2){

        x = internalData.dim0 + indexX*internalData.dx;
        y = internalData.dim2 + indexY*internalData.dy;
        z = internalData.dim4 + indexZ*internalData.dz;

        result = calcChiCalcPoint(orbital, molecule, x, y, z);

        results[indexX + (internalData.ncub0*indexY) + (internalData.ncub0*internalData.ncub1*indexZ)] = result;
    }

}
```

Code Snippet 10

Here we can see that the actual result is calculated in the function calcChiCalcPoint, which we talked about earlier in chapter 3.2.2. Every other piece of logic deals with finding the position of a thread in 3D space. Because the mathematical problem we are solving is in a 3 dimensional space then so is our indexing space. This is the reason why we have variables indexX, indexY and indexZ. These tell us exactly where we are in the located in the space of threads that we launched. After that we need to check if we are actually in range of the problem area. It gives faster results to make standard size blocks of threads and place them in a grid that covers the problem area rather than make many different size thread blocks and launch them separately. We want to leave the CPU and commence our calculations on the graphics card as fast as possible. So it is better to let each individual thread calculate if they are in the problem area and then act accordingly. After that we locate the proper position in the results array and write our result there.

After the threads have completed their work we need to copy the results array back to the RAM memory and write the results into the image object that will display them to the user. This is a fairly fast operation. After that we need to free all the memory we have allocated.

When keeping all of this in mind then we are quick to notice that the amount of work we do in the CUDA version of the program is substantially greater than the work we do in the serial part of the program. This raises the question: was all of this worth it?

4. Results and Analysis

The problem that we are trying to solve is that it can take up to 44 seconds to do a certain computation in a program called Molekel. This is caused by a group of nested loops which call upon a computation function during each of their iterations. That computation function is expensive in terms of time and the program is running in a single thread. Due to the fact that the loop iterations are independent we are trying to solve this problem by making all of those iterations run in parallel, in a separate thread on the GPU.

4.1 Results

The results that were received on the main testing computer can be seen in Table 1

Iterations	CPU time	Total CUDA time	Kernel run time	Speed-up
6498	0,376	0,053	0,002	7,094
7600	0,529	0,055	0,002	9,618
9240	0,549	0,072	0,004	7,625
12144	0,694	0,073	0,005	9,507
15600	1,003	0,074	0,005	13,554
21924	1,330	0,084	0,007	15,833
29760	1,774	0,074	0,009	23,973
41580	2,463	0,085	0,012	28,976
62320	3,612	0,116	0,017	31,138
99264	5,704	0,101	0,025	56,475
169176	9,711	0,151	0,044	64,311
332990	18,750	0,208	0,079	90,144
786315	44,664	0,380	0,180	117,537

Table 1

Table description:

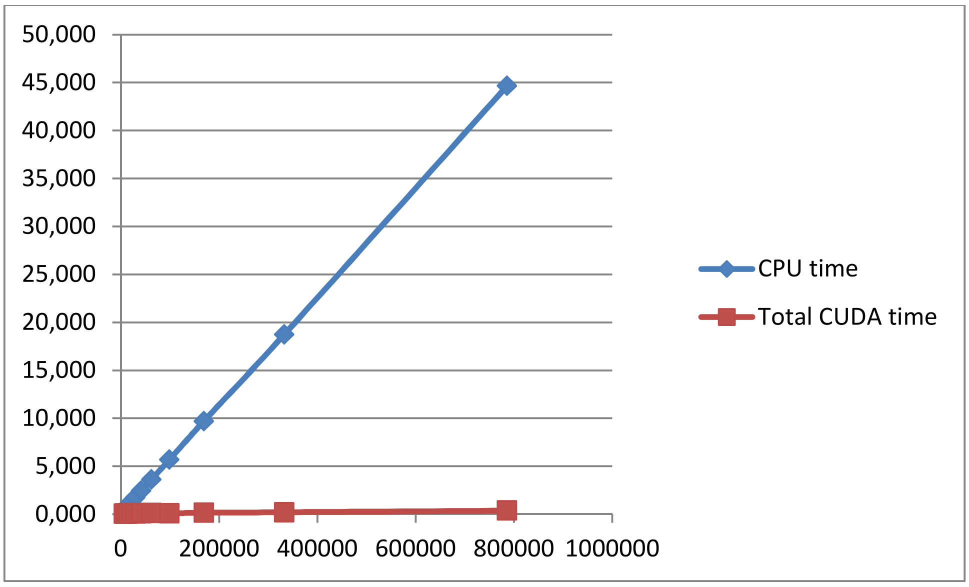
- The iterations column shows how many points were calculated. We use this to measure the amount of work that was done.
- The CPU time column shows how much time it took when the computation ran on the CPU.
- The Total CUDA time column shows how long did the entire CUDA part of the program ran.

- The Kernel run time column shows how long did the actual computation on the graphics card ran.
- The speed-up column shows how much faster did the total CUDA section of the program ran in comparison to the CPU section of the program.

The test was run on a computer with the following properties:

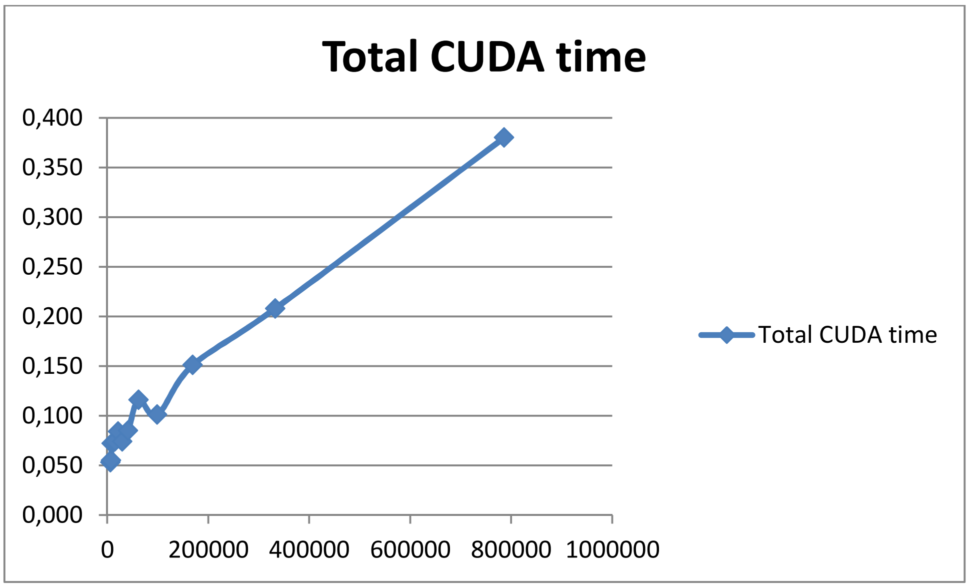
- Processor: Intel i7 4930k 6 cores 3.4GHz each [20]
- RAM: 16GB DDR3
- Graphics: NVIDIA GeForce gtx 780 TI [21]
- Storage: Samsung 1TB SSD
- Motherboard: ASUS Rampage 4 Extreme Black Edition [22]

When examining the table it is clear that the supporting operations that are needed to make the computations happen take up the bulk of the time when running CUDA code. What is surprising is that even when there are not that many iterations, the GPU can still provide a performance boost. The execution times are represented in plot 3.



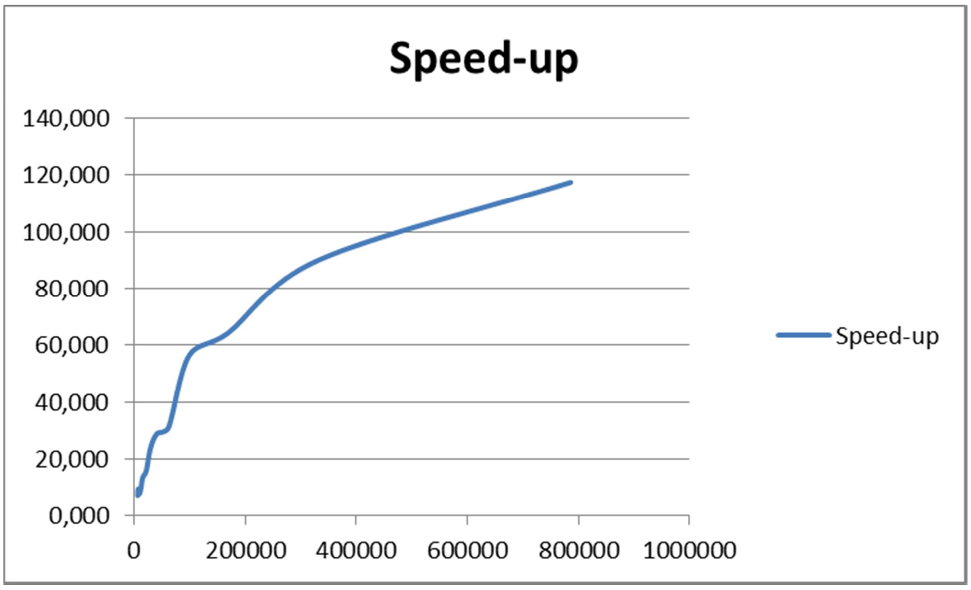
plot 1

From the plot we can see that the time it takes to execute the CPU section scales linearly in relative to the amount of work done and the GPU section seems to be a constant. The second part of that statement is an illusion and this is illustrated by plot 2 which shows that the execution times on the GPU also scale to a function that grows similar to a linear fashion. This is not exactly what was expected in the beginning of this experiment, due to the fact that GPU has many cores and using those should provide a more exponential growth.



plot 2

When taking a closer look at the workload it becomes obvious that as the workload increases so does the gained speed-up, because the CPU has only 1 core that does the work, while GPU has much more than that. This is represented in plot 3.



plot 3

It is clear that the speed-up factor will increase rapidly as the workload increases, but due to hardware restrictions it cannot increase to infinity. Never the less we have fulfilled the goal of this work which was to gain a speed-up of 100 times. We can now say that this is possible, but only when there is a sufficient amount of workload available to GPU and when there is a proper graphics card available.

However what would the results be if we would change our system configuration to the following:

- Processor: Intel i5 2500 4 cores 3.6GHz each [20]
- RAM: 8GB DDR3
- NVIDIA GeForce gtx 550 TI [24]
- Storage 500GB HDD
- Motherboard: GIGABYTE Z68P-DS3 [25]

In this case the processor has less cores but each of them runs at a faster rate which implies that we should get a faster CPU run time. However the graphics card now has 15 times less cores totaling 192 cores grouped into 4 SMXs instead of 2880 cores grouped into 15 SMXs. In addition to that each SMX now has 64kb of cache memory instead of the 128kb we had before and we have gone from a total computing power of 5.0 TFLOPS to 0.7 TFLOPS. It seems that we do not have much to expect, however the actual results can be seen in Table 2.

Iterations	CPU time	Total CUDA time	Kernel run time	Speed-up
6498	0,327	0,056	0,007	5,798
7600	0,390	0,044	0,007	8,864
9240	0,484	0,047	0,010	10,298
12144	0,624	0,051	0,011	12,235
15600	0,796	0,053	0,015	15,019
21924	1,107	0,061	0,020	18,148
29760	1,513	0,082	0,029	18,451
41580	2,091	0,082	0,039	25,500
62320	3,136	0,137	0,057	22,891
99264	4,961	0,164	0,090	30,250
169176	8,457	0,230	0,157	36,770
332990	16,616	0,380	0,283	43,726
786315	39,158	0,830	0,649	47,178

Table 2

The table columns have the exact same meaning as Table 1 columns. The iterations column is exactly the same, the CPU time column shows that the second testing computer really has a faster core, but the GPU results are disturbing. The speed-up results do not differ much from the beginning and in the end of the experiment they only differ by a factor of 2.5 which indicates that we are not using the full power of the stronger graphics card. Why does this happen?

It is obvious that the cores do not do as much computations as they could and this is probably caused by not getting access to the computational data fast enough. Gtx 550TI has 48 cores and 64kb memory in an SMX. Gtx 780 TI has 192 cores and 128kb of cache in an SMX. So the slower card has more cache memory per core, which raises the question that perhaps the problem lies in the shortage of cache memory which limits the computing power. However we have little idea of how the two cards are built and what more structural differences they have. There are also other factors in GPU computing like temperature which was not measured but does have an impact to the general performance. At the present moment it is unclear what causes this result, but due to the fact that the programmer was not a CUDA expert while writing this then most likely the problem lies within the code.

However if this problem could be found and corrected, then it is possible that the speed-up factor may increase even further.

4.2 Topics for further study

This work has solved a problem by making a single threaded application faster by porting it to the GPU, but it has also uncovered topics that are not quite clear and need further study.

- How does the GPU run thread blocks on SMXs? – If several blocks may run on a single SMX at once then the GPU must be playing some sort of “Tetris” to place them there. Uncovering how this works might also be a key to getting better performance from the GPU.

- Are there any databases that could be powered by a GPU? – If the database is properly built then queries usually do not take much time, but what if we absolutely have to do a full scan on an extremely large table? Can the GPU help?

- Is it possible to optimize computation intensive parts of modern games like path finding and physics calculations using the GPU? What are the obstacles?

- How does intensive computation affect the graphics card and how fast? – It is a known fact that mining Bitcoin [26] for long periods of time damages the graphics card. Is it possible to avoid that when writing our own computations, but still receive a proper performance upgrade?

4.3 The future of CUDA powered Molekel?

The development of Molekel to port it to the GPU will continue. The progress can be observed at the Github repository [19] which contains the codebase itself, dependencies and information on where to get stable compiled versions which can already be used in everyday work.

At the present moment the first computation we reviewed in this document is complete and work is focused on optimizing a computation which lasts up to 10 minutes. Bulk of the work is complete but certain technical issues still need to be addressed.

5. Summary

In conclusion we can say that CUDA is a helpful tool when it comes to increasing the speed of time consuming calculations. It is not alone in the field of GPU computing and it has competitors which can be taken seriously. The good side is that CUDA is easier to setup and learn but the bad side is that it requires specific hardware to use.

If we are looking to get CUDA support in our applications it is not necessary to write custom code. We could use a prebuilt library or compiler hints. However these do not provide the fastest speed-up possible.

However when we are looking for the fastest speed-up possible and when we are writing custom code then there are materials like the CUDA documentation [15] to which we can refer to for help. Writing simple CUDA code is not really difficult, but it gets harder as the amount and complexity of data structures that need to be used in the computations increase. Furthermore not all data structures are viable to use because they can be either impossible to implement or are simply not worth the effort of implementing in CUDA code. Linked list is one of them. In addition to the data structures we must also implement the logic of finding the correct computational data form the total amount that we sent to the GPU and using it correctly.

However when it is possible to get past the problems of writing custom code, then the results are definitely worth it. The minimum speed-up we gained from our main testing computer was 7,094 times and the maximum was 117,537 times. It was clear that the achievable speed-up was related to the amount of work that the computation involved. As the workload increased, so did the speed-up factor, but it was also clear that it cannot go to infinity and eventually will be limited by the hardware. The second testing computer which had a slightly stronger processor but a remarkably weaker graphics card performed better than expected. The minimum speed-up gained was 5,798 times and the maximum was 47,178 times. It is unclear why a graphics card with 15 times less cores performs only 2.5 times worse, but most likely the problem lies within the code.

This brings us to the conclusion that a 100 times speed-up for a computation is possible when porting it to a GPU. However the speed-up is affected by the workload, the hardware and most importantly the programmer’s skills. On a powerful graphics card it is possible to achieve the said speed-up with relatively standard skills, but much more can be achieved if the programmer is aware of more complex topics of how the GPU works.

6. References

[1] (2015, December) CUDA Home. [Online]. http://www.nvidia.com/object/cuda_home_new.html

[2] (2015, December) OpenGL Home. [Online]. https://www.opengl.org/

[3] (2015, December) NVIDIA Intro to parallel programming course. [Online]. https://www.udacity.com/course/intro-to-parallel-programming--cs344

[4] (2015, December) OpenCL Home. [Online]. https://www.khronos.org/opencl/

[5] Vincent Hindriksen. (2011, June) OpenCL VS CUDA comparison. [Online]. http://streamcomputing.eu/blog/2011-06-22/opencl-vs-cuda-misconceptions/

[6] Microsoft Corporation. (2015, December) DirectCompute Home. [Online]. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx

[7] Andrew Adinetz. (2014, May) Introduction to dynamic parallelism. [Online]. http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/

[8] Rahul. (2013, July) DirectCompute from an OpenCL and CUDA perspective. [Online]. http://codedivine.org/2013/07/25/directcompute-from-an-opencl-and-cuda-perspective/

[9] Vincent Hindriksen. (2010, December) DirectCompute's unpopularity. [Online]. http://streamcomputing.eu/blog/2010-12-28/directcomputes-unpopularity/

[10] NVIDIA Corporation. (2015, December) NVIDIA Accelerated Computing Training. [Online]. https://developer.nvidia.com/accelerated-computing-training

[11] NVIDIA Corporation. (2015, December) NVIDIA gpu accelerated libraries. [Online]. https://developer.nvidia.com/gpu-accelerated-libraries

[12] Adobe Systems inc. (2015, December) Adobe Premiere Pro CC home. [Online]. http://www.adobe.com/products/premiere.html

[13] NVIDIA Corporation. (2015, December) OpenACC Toolkit. [Online]. https://developer.nvidia.com/openacc

[14] NVIDIA Corporation. (2015, December) OpenACC Guide. [Online]. https://developer.nvidia.com/how-to-openacc

[15] NVIDIA Corporation. (2015, September) CUDA Toolkit Documentation. [Online]. http://docs.nvidia.com/cuda/index.html#axzz3uaS6V5Mz

[16] (2015, December) Stack overflow CUDA tag. [Online]. http://stackoverflow.com/questions/tagged/cuda

[17] (2010, June) Molekel Home. [Online]. http://ugovaretto.github.io/molekel/

[18] (2010, June) Molekel Gallery. [Online]. http://ugovaretto.github.io/molekel/wiki/pmwiki.php/Main/Gallery.html

[19] Erik Soekov. (2015, December) Molekel CUDA version repository. [Online]. https://github.com/erik132/MolekelCUDA

[20] Intel. (2015, December) Intel i7 4930k specifications. [Online]. http://ark.intel.com/products/77780/Intel-Core-i7-4930K-Processor-12M-Cache-up-to-3_90-GHz

[21] NVIDIA Corporation. (2015, December) NVIDIA GeForce gtx 780 TI Specifications. [Online]. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780-

ti/specifications

[22] ASUS. (2015, December) ASUS Rampage 4 Extreme Black Edition Specifications. [Online]. https://www.asus.com/Motherboards/RAMPAGE_IV_BLACK_EDITION/specifications/

[23] Intel. (2015, December) Intel i5 2500 specifications. [Online]. http://ark.intel.com/products/52209/Intel-Core-i5-2500-Processor-6M-Cache-up-to-3_70-GHz

[24] NVIDIA Corporation. (2015, December) NVIDIA GeForce gtx 550 TI specifications. [Online]. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-550ti/specifications

[25] GIGABYTE. (2015, December) GIGABYTE Z68P-DS3. [Online]. http://www.gigabyte.com/products/product-page.aspx?pid=3899#sp

[26] Bitcoin Project. (2015, December) Bitcoin home. [Online]. https://bitcoin.org/en/