# Final Report

**Matthew Nunes**

# Table of Contents

# Abstract

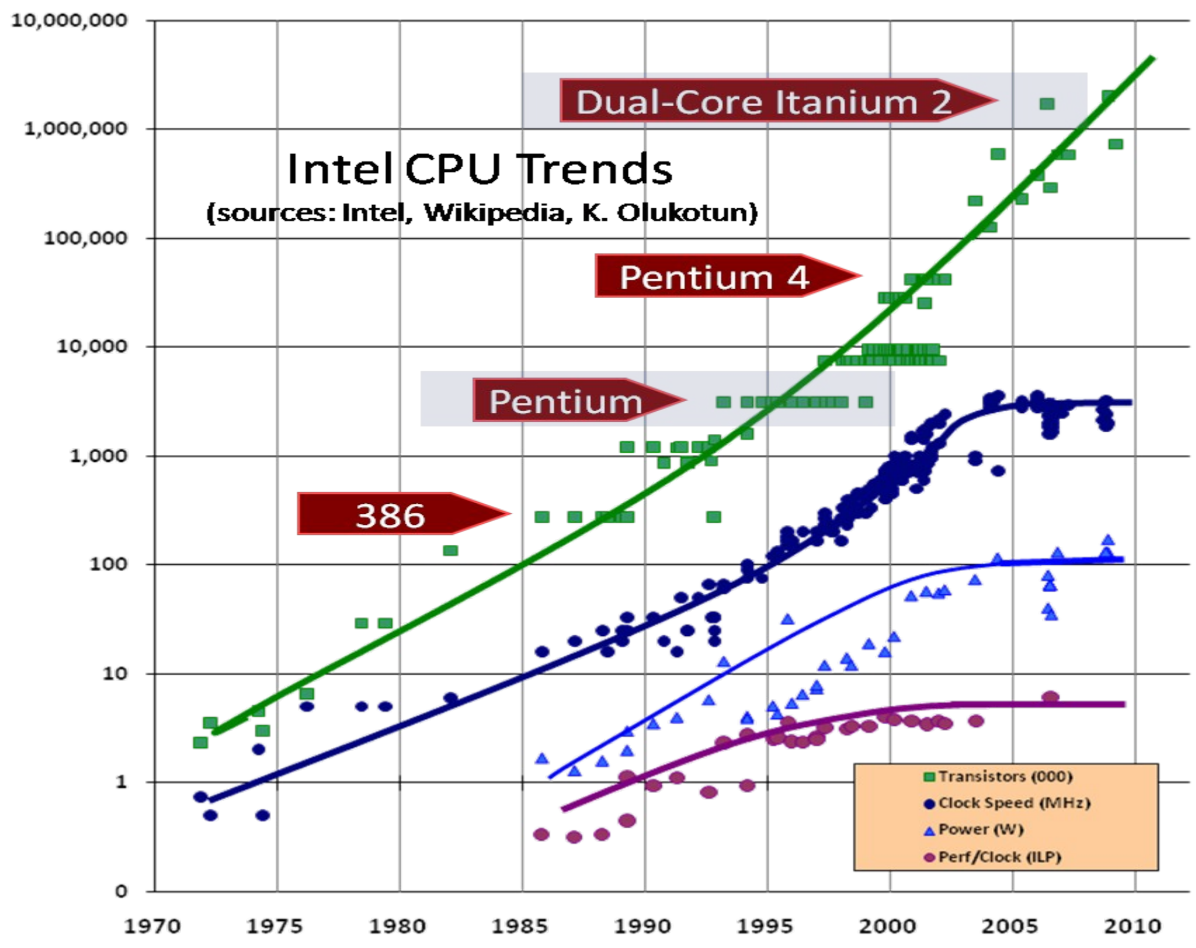
Most computers that are available for purchase nowadays contain CPUs with multiple cores, furthermore, due to advancements in technology, graphics cards are becoming increasingly affordable and commonplace. However, when not being used to render complex graphics on the screen, the Graphics Processing Units are usually sitting idly by. This is a dreadful waste of resources. Therefore, to fill this void, General Purpose computing for Graphics Processing Units was introduced (GPGPU). This allowed programmers to offload work to the GPU (as a coprocessor) using a language extension built for general purpose computing on a graphics card (as opposed to graphics computing). Two such language extensions are OpenCL and CUDA. CUDA was introduced by Nvidia specifically for their graphics cards as opposed to OpenCL, which, as its name suggests, is designed to run on graphics cards (among other devices) from a variety of vendors. The aim of this project was to study whether OpenCL provided portability at an unreasonable cost by implementing a number of algorithms using both OpenCL and CUDA and comparing the performance. The results show that while OpenCL is indeed slower than CUDA, it is not substantially slower than CUDA and hence is a good platform independent alternative to CUDA. [1][2][3][4][5][6][9]

# Introduction

In this report, I will evaluate OpenCL as a portable/platform independent alternative to CUDA for programming to Graphics Processing Units for general purpose computing. In order to ensure a fair comparison, I implemented a large variety of algorithms in both languages and then plotted the time taken against the problem size. The algorithms that I chose to implement in this project are the matrix multiplication algorithm, the Laplace equation, an image blurring algorithm and the molecular dynamics algorithm.

# Motivation and Background

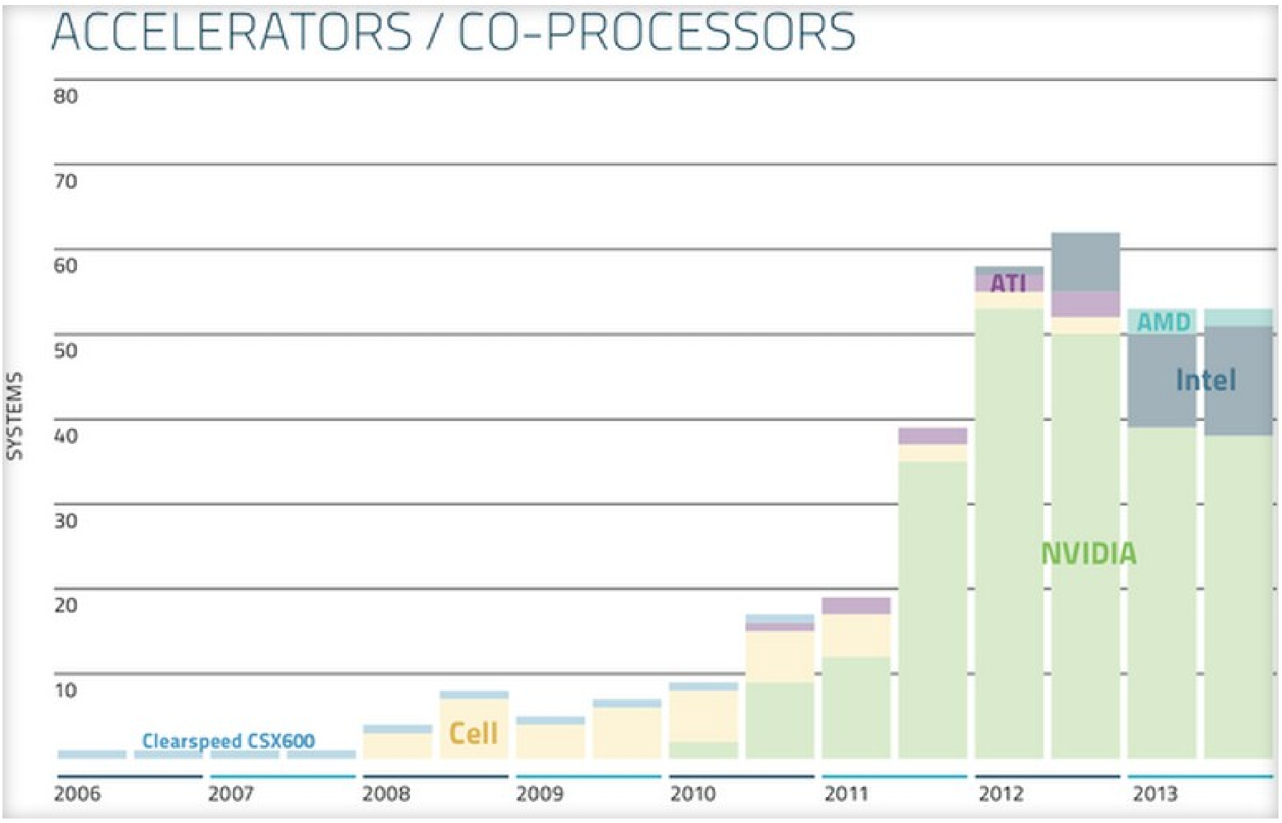
In 1965 Gordon Moore from Intel proposed that the number transistors on a chip would double every eighteen months (later altered to two years). Since his prediction, the number of transistors on a chip have indeed doubled every two years, however, as the size of transistors are reduced to allow more to be squeezed onto a single chip, heat dissipation and power consumption become ever increasing problems. Not to mention, cost. Experts predict that Moore's law will come to an end some time between 2017 and 2020. Already signs of that are beginning to show, as, from 2007 to 2011 clock speed rose by 33%, while, from 1994 to 1998, clock speeds rose by 300%. The graph below shows how various parameters concerning a CPU have varied over time:



[32]

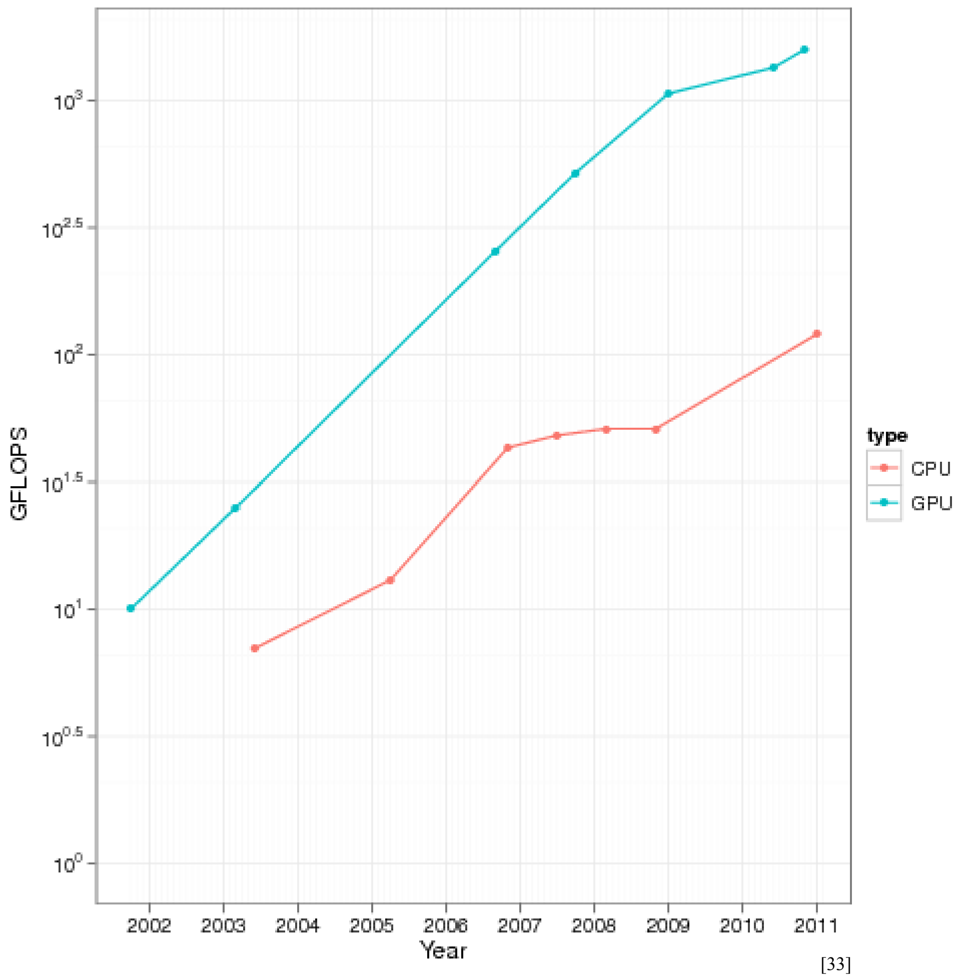
As is evident in the graph shown above, the clock speed has reached a plateau for many of the reasons I mentioned earlier and while it is true that the graph shows the number of transistors increasing, this is because the number of cores on a chip are increasing, but, the speed of each core is unchanged. As a result, one might expect that given the eventual death of Moore's law (as proven by the graph above), demand for faster computers will seize. Unfortunately, this has not been the case. [1][2][3][4][5][6][9]

In order to meet this growing demand, engineers have looked for alternative methods to improve performance. One such method (which I touched on earlier) is to add more cores to a processor. Multi-core processors have proven to greatly reduce the time taken to perform a number of tasks when utilised to their full potential. In addition, to supplement multi-core CPUs, manufacturers of Graphics Processing Units (GPUs) have made it possible for programmers to offload general tasks to the GPU (Prior to this, GPUs were generally only used for graphics programming). This is commonly referred to as General Purpose computing on Graphics Processing Units (GPGPU). To the untrained eye this may not seem like much to celebrate about, but GPGPU has opened the doors to a plethora of applications that can now be simulated/calculated/tested on computers due to the huge performance gains that are provided by employing the assistance of GPUs, as, unlike CPUs, GPUs contain tens to hundreds of cores and should therefore not be overlooked. Furthermore, what is worth noting is that as off November 2013, 53 of the computers in the top 500 list of fastest computers contained accelerators/coprocessors. A graph showing how the number of co-processors in the top 500 computers has varied over time is shown below:



[25]

Analysing this trend for computers in the top 500 to contain co-processors further highlights the importance of co-processor in improving performance. The graph also shows that while all co-processors were at one time only supplied by Nvidia to the top500, other manufacturers (such as Intel and AMD) are beginning to realise their potential and have obviously begun producing GPUs of comparable performance. Another reason why GPUs are growing in popularity with such pace is evident in the graph shown below which compares the Floating point Operations Per Second (FLOPS) of GPUs and CPUs over time:



[33]

In the graph above GFLOPS refers to Giga FLOPS. GFLOPS is just one measure that can be used to evaluate performance (or at the very least, provide a rough idea as to the performance one should expect). Knowing this, it is clear from the graph above that the potential for performance gain within an application that utilises the GPU is far greater than that which can be achieved by only using a CPU in an application (as is the case with sequential code).[4][5][9][25][26]

Before I get lost in praise for GPUs, it is important to remember that as the well known No Free Lunch Theorem (NFLT) suggests, all this performance gain does not come without a cost. Users only stand to see performance improvements if developers exploit the multiple cores and co-processors (GPUs) in their software. Essentially this requires developers to learn a new programming model, one, which, can be frustratingly difficult to use and debug. While this may be feasible for new applications, but for existing software (especially commercial software filled with legacy code), it can present quite a challenge to parallelise parts of the code to improve performance. This is especially true if the software is run on a variety of different machines with different capabilities. [1][4][5][9]

From the software models currently available, one of the more popular ones that allows programmers to employ a GPU as a coprocessor for general tasks is called Compute Unified Device Architecture (otherwise known as CUDA). CUDA was introduced by Nvidia in 2007 and runs specifically on Nvidia's line of GPUs. CUDA provides extensions to the programming language that allow the programmer to achieve parallelism through employing the GPU to perform some tasks alongside the CPU [7][8].

Another parallel programming architecture available is Open Computing Language (OpenCL). This differs from CUDA in that it aims at standardising software development on GPUs. OpenCL was introduced by Apple in 2008 as the first cross platform language for heterogeneous systems and it is currently managed by the Khronos Group which consists of industries such as Apple, Nvidia, AMD and Intel among others. In addition to running on a GPU, it can run on a CPU, FPGA (Field Programmable Gate Array) or DSP (Digital Signal Processor) [10][11][12].

While OpenCL boasts of being able to run on GPUs from any vendor (among other devices), through this project I attempted to find out whether such interoperability came at a cost. Essentially I wanted to draw a comparison between OpenCL and CUDA not just in terms of performance but also in relation to usability and online support [10][11][12].

# Literature Review

There have been a significant amount of papers published concerning the performance differences between OpenCL and CUDA. They differ either by the algorithms that they have used, the sections they have timed and/or even the manner in which they have chosen to implement the algorithms (For example, whether to use language specific optimisations or not).

Su et al compared C, OpenCL, the CUDA Driver API and the CUDA Runtime API. The CUDA Runtime API is built on top of the CUDA Driver API and it is easier to use. To add to its usability, it automatically links all your CUDA files into one executable (unlike the CUDA Driver API). On the other hand, the CUDA Driver API is much more challenging to use but it comes with the benefit that the developer has much more control. [14][15]

The algorithms with which each language was compared were a variety of image and video processing algorithms. The authors decided to use global memory for all memory transfers to keep things as fair and objective as possible. From their experiments they concluded that the CUDA Driver API was between 94.9% to 99 % faster than C and 3.8% to 5.4% faster than OpenCL. They felt that the difference between OpenCL and CUDA in performance was too small to dissuade one from using OpenCL to obtain the inter-platform characteristic it has to offer. In addition, they observed that OpenCL follows a similar model to CUDA such that once one is learnt, learning the other is trivial. [14][15]

Fang et al investigated the cost in performance as a result of the portability provided by OpenCL. To distinguish themselves from other research in a similar area, they measured the performance for a large number of algorithms and did an incredibly in-depth analysis to discover the reasons behind the performance difference between OpenCL and CUDA whenever it occurred during their experimentation. [16]

They started by analysing how the peak performance of OpenCL and CUDA varied for a number of operations. The first operation which they measured the peak performance for was the bandwidth of device memory. They found that OpenCL proved to be between 2.4% - 8.5% faster than CUDA (depending on the GPU used). On the other hand, when it came to peak Floating Point Operations per Second (FLOPS), they noticed that OpenCL produced a similar performance to CUDA, in fact, they even claimed that OpenCL was slightly better suggesting that it has the potential to use the Hardware just as efficiently as CUDA. [16]

After evaluating the peak performance of CUDA and OpenCL, the authors chose to compare the performance against a large number of real-world algorithms. The results from this brought them to the conclusion that differences in performance between OpenCL and CUDA were due to distinct programming models, differences in the kernels, architecture differences and compiler differences. Notable among these was the fact that CUDA provides access to Texture memory giving it a slight edge over OpenCL in performance. [16]

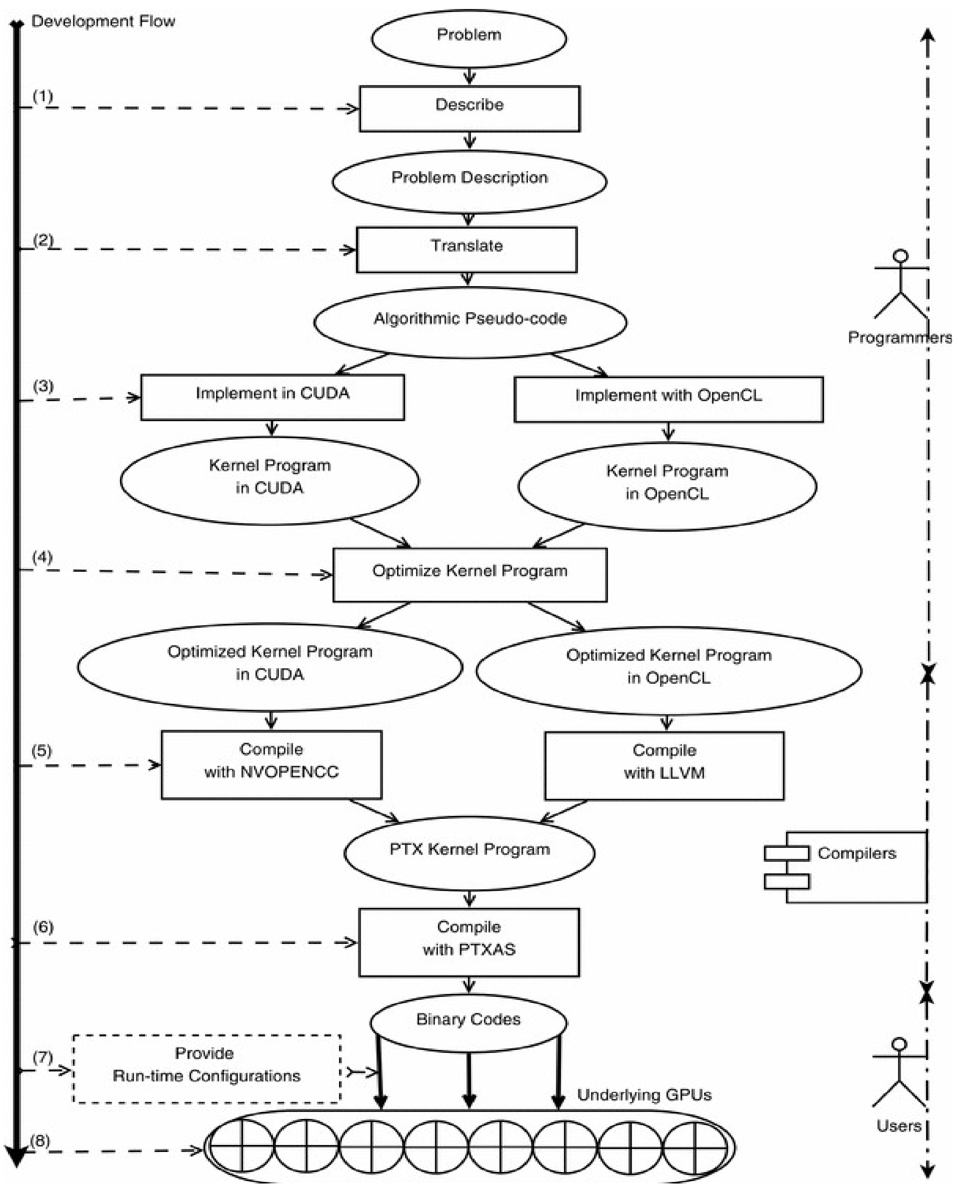
To detect whether the compiler hand anything to do with the differences in performance, the authors analysed the code produced by the compiler and were able to conclude that the CUDA compiler was much better equipped to optimise code. This becomes obvious by analysing the table below which shows the number of instructions each compiler generated for different classes of operations for the same algorithm:

| **Class of Instruction** | **Number of CUDA instructions** | **Number of OpenCL instructions** |
|---|---|---|
| Arithmetic | 220 | 521 |
| Logic Shift | 4 | 163 |
| Data Movement | 1131 | 351 |
| Flow Control | 4 | 188 |

While it's true that the CUDA contained many more move instructions, the authors pointed out that most of these instructions involved moving values between registers and so they weren't very costly. [16]

Additionally, the authors pointed out that the OpenCL kernel takes longer to launch than a CUDA kernel (since OpenCL compiles its kernels dynamically) which could explain why OpenCL seems to take slightly longer to execute a kernel than CUDA to the casual observer. [16]

From the results of these experiments, the authors were able to derive eight variables that must be kept similar to ensure fair comparison between OpenCL and CUDA. They stressed that if these eight points are adhered to, OpenCL will show similar performance to CUDA. From the eight points, four are the responsibility of the programmer writing the code, two are in the compiler's control and two are in the user's hands. A diagram giving more detail concerning each step that needs to be taken to ensure a fair comparison is shown below:



Finally, the authors conclude by stressing that there is no reason that OpenCL should show worse performance than CUDA under fair comparison conditions. The differences in performance that were initially observed were due to reasons mentioned earlier such as the developer, compiler and the users. [16]

Komatsu et al. compares the performance of OpenCL and CUDA on a few algorithms using a variety of compilers and Graphics Cards. From the five algorithms that were written in OpenCL and CUDA, the CUDA implementations was faster for all except one where it took just as long as the OpenCL implementation. However, it should be noted that the algorithm in which the time taken was the same did not perform any calculations within the Kernel. It simply called memory transfer routines to test the bandwidth between the CPU and GPU. [17]

Given that the authors faithfully translated the CUDA algorithms into their exact representation in OpenCL, they scrutinised the PTX code (Parallel Thread Execution; the compiled code) in order to shine a light on the reasons for the performance difference. After detailed analysis, they noticed that the PTX code for the CUDA version of the algorithm was only generated after the compiler performed a number of optimisations while the PTX code for the OpenCL version was relatively simplistic. For example, they noticed that the loops in the CUDA code had automatically been unrolled (to increase the amount of computations per iteration and reduce the amount of iterations) while those for the OpenCL version had not been touched. Additionally, with CUDA, common sub-expression elimination (whereby a compiler evaluates the result for an expression only once even if the same expression is being redundantly repeated in the code) had been used but with OpenCL the calculation had been performed repeatedly. Another technique that was employed when the CUDA code was compiled is called loop invariant code motion. This refers to a technique that removes a calculation from within a loop if its value remains unchanged during the execution of the loop. [17][18][34]

When the developers manually applied all of these techniques to the OpenCL code, they noticed that the OpenCL version of the algorithm took the same amount of time as the CUDA implementation. Furthermore, they noticed that by enabling the cl-fast-relaxed-math when compiling their OpenCL programs, many of the optimisations which they performed manually were performed automatically and the OpenCL programs showed a similar performance to the CUDA programs (for all but one algorithm). [17]

Finally, they concluded by performing their tests on a variety of GPUs whilst varying the block widths (number of threads per block). From this they concluded that Nvidia's Tesla outperforms AMD's Radeon and that the block width severely affects performance (more than the manual optimisations). [17]

Sanden does an in depth analysis of OpenCL's performance and portability. The author measures OpenCL's performance on both GPUs and CPUs from different manufacturers, namely, Nvidia, AMD and Intel. To be more specific, Sanden tested performance for AMD's GPU, Nvidia's GPU and Intel's i7 processor. [31]

When comparing OpenCL with CUDA on an Nvidia GPU, Sanden found that CUDA performs 16% faster. This was primarily due to the additional time OpenCL takes to actually compile the kernel before launching it and because of the differences in the PTX codes generated by the compiler. The author also points out that as newer GPUs are used, the gap in performance between CUDA and OpenCL narrows. [31]

In terms of architectural differences, the results from Sanden's experiments show that for three of the algorithms tested, OpenCL ran faster on an AMD GPU than on Nvidia's GPU. On Intel's i7 processor, though, while the algorithms were very efficient for a CPU, they couldn't compete with the GPUs due to the massive amount of threads that GPUs can work with compared with the CPUs (which show optimal performance with just a few threads).[31]

Sanden then compared the performance from running the OpenCL algorithms optimised for Nvidia's GPUs on AMD's GPUs and Intel's i7 processor. The results indicated that for three of the algorithms, the AMD GPU showed the same performance as the Nvidia GPU and was slower for the other three indicating partial portability. However, when run on Intel's CPU, the performance was poor due to the huge number of threads that the CPU had to schedule. When the algorithms optimised for AMD's GPU are run on Nvidia's GPU and Intel's CPU, they displyed similar results to the previous experiment except for one of the algorithms that relied on the fact that a warp contains 64 threads (as is the case on AMD's GPUs where they are called wavefronts) which is not the case with Nvidia's GPUs where a warp only contains 32 threads. As direct consequence of that, this algorithm failed to work without first being modified.[31]

Finally when the algorithms that were optimised for Intel's CPU were run as is on the GPUs they exhibited poor performance. This is due to the fact that the CPU demonstrates optimal performance with just a few threads. Too many threads increase the scheduling overhead for the CPU. From this, the author concluded that OpenCL isn't perfectly portable and that there is still a lot of work to be done to bring it to the point where it can tune itself to run optimally on the device it is being executed on.[31]

The author concludes his report by showing one way in which OpenCL programs can be manually written to run optimally on more than one architecture and device. He uses compiler directives (such as #define and #if #endif) to ensure that the most optimal code is dynamically chosen depending on the device it is being run on.[31]

Karimi et al compared OpenCL and CUDA by first writing their kernels in CUDA and doing the least amount of transformations necessary to port that code into OpenCL. They also described how the OpenCL code written to run on Nvidia's GPUs differs from the code written for ATI's GPUs. The main difference noted between ATI's OpenCL and Nvidia's OpenCL is that the programmer cannot statically allocate global memory in the kernel using an ATI compiler. All memory must be dynamically declared outside the kernel and then passed as a pointer to the kernel.

On OpenCL's portability front, the authors were able to achieve source level compatibility between ATI and Nvidia but the executables were not compatible. In terms of performance, the authors found that OpenCL took longer to transfer the data to the GPU than CUDA did, however, they did point out that the data transfer time didn't change significantly for different problem sizes. In addition, their results indicated that CUDA was able to process more variables per seconds than OpenCL. In relation to execution time, OpenCL's kernel is 13% - 63% slower and the overall execution time for the algorithm was between 16% -67% slower than CUDA.[30]

From the literature reviews that I have carried out a common theme seems to emerge. Namely, that OpenCL suffers in performance due to differences in the code produced by the compiler. Some have attributed this to the immaturity of OpenCL (and its compiler) while others have claimed that it is due to the fact that OpenCL has to cater to such a huge multitude of devices and therefore it can't possibly be expected to make the same assumptions about the OpenCL code that it can about CUDA code (which are required in order to apply the optimisations). For the doubters amongst us, some of the authors went further to manually add every optimisation to their OpenCL implementation that was added to the CUDA implementation automatically by the compiler. Astonishingly, this did actually make all the difference and bridged the gap between the two implementations.

# Problem Description

If OpenCL is indeed to become the standard for GPU programming as the Khronos group envision, it needs to have a relatively low barrier to entry, in addition, it should provide reasonable performance benefits for the cost of implementation (in terms of time and effort). Therefore, the best way to assess its usefulness is to compare it with a widely used but heavily restricted programming model, namely, CUDA.

However, this project is not limited to simply comparing both the programming models but goes further to investigate how easily the algorithms chosen lend themselves to parallelism. In addition, the art of adding parallelism to a chosen sequential algorithm will be scrutinised to determine whether it is worth the effort.

# Approach to problem

In order to ensure a fair comparison of both OpenCL and CUDA, a variety of algorithms were parallelised using both architectures and the host code was implemented in plain C for both programming architectures. The algorithms I used in order to evaluate the efficiency of OpenCL and CUDA are the matrix multiplication algorithm, the Laplace equation, the image blurring algorithm and the molecular dynamics algorithm.

To determine whether the algorithms were implemented correctly, the results produced by the parallel version were compared with those produced by the original sequential version to ensure that they were near identical. For most of the timing experiments the time measured was the time taken for the kernel to execute. For the sake of consistency a simple C method was used to compute the time taken for the Kernel to execute as opposed to using any of OpenCL and CUDA's timing mechanisms provided (since they may differ in their implementation). Each individual time recorded was measured ten times after which the average was taken to reduce any inconsistency in the results from background processes or other forms of noise. After a significant amount of values were gathered, graphs were drawn to visualise the results and make accurate conclusions.

In keeping with consistency, all algorithms were tested on the same graphics card of the same machines. The main machine on which most of tests were run is called Tesla and the Graphics card on which the code was run provides the following specifications to CUDA when queried:

**Major revision number:** 2
**Minor revision number:** 0
**Name:** Tesla C2070
**Total global memory:** 1341587456 bytes
**Total shared memory per block:** 49152 bytes
**Total registers per block:** 32768 (There are four bytes per register)
**Warp size:** 32 threads
**Maximum threads per block:** 1024
**Maximum dimension 0 of block:** 1024
**Maximum dimension 1 of block:** 1024
**Maximum dimension 2 of block:** 64
**Maximum dimension 0 of grid:** 65535
**Maximum dimension 1 of grid:** 65535
**Maximum dimension 2 of grid:** 65535
**Clock rate:** 1147000 kHz
**Total constant memory:** 65536 bytes
**Number of multiprocessors:** 14

The other machine on which one of the algorithms were run is called Pomegranate. The graphics card in this machine lists the following specifications:

**Major revision number:** 3
**Minor revision number:** 5
**Name:** Tesla K20Xm
**Total global memory:** 1744371712 bytes
**Total shared memory per block:** 49152 bytes
**Total registers per block:** 65536 (There are four bytes per register)
**Warp size:** 32 threads
**Maximum threads per block:** 1024

**Maximum dimension 0 of block:** 1024
**Maximum dimension 1 of block:** 1024
**Maximum dimension 2 of block:** 64
**Maximum dimension 0 of grid:** 2147483647
**Maximum dimension 1 of grid:** 65535
**Maximum dimension 2 of grid:** 65535
**Clock rate:** 732000 kHz
**Total constant memory:** 65536 bytes
**Number of multiprocessors:** 14

All the code (both OpenCL and CUDA) was compiled with the same compiler, namely, release 4.0, V0.2.1221 of Nvidia's nvcc compiler on Tesla and release 5.5, V5.5.0 on Pomegranate.

In terms of CPUs, below is a summary of Tesla's specifications:

**Number of Processors (including cores):** 12

**<u>Details for one Processor:</u>**
**Vendor ID:** GenuineIntel
**CPU Family:** 6
**Model:** 44
**Model Name:** Intel(R) Xeon(R) CPU E5649 @ 2.53GHz
**CPU MHz:** 1600.000
**Cache Size:** 12288 KB
**Physical ID:** 0
**Siblings:** 6
**Core ID:** 0
**CPU Cores:** 6
**FPU:** yes
**CPU-ID Level:** 11
**Address Sizes:** 40 bits physical, 48 bits virtual

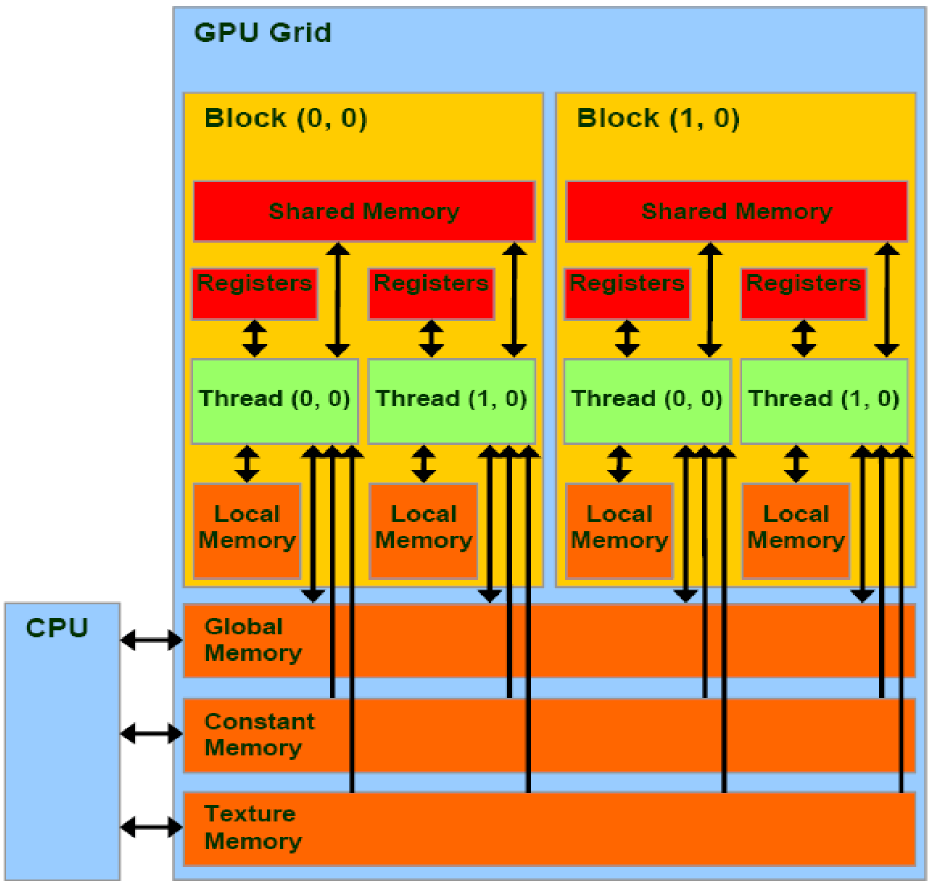
The CPU in Pomegranate has the following specifications:

**Number of Processors (including cores):** 12

**<u>Details for one Processor:</u>**
**Processor:** 0
**Vendor ID:** GenuineIntel
**CPU family:** 6
**Model:** 45
**Model Name:** Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
**CPU MHz:** 1200.000
**Cache Size:** 15360 KB
**Physical ID:** 0
**Siblings:** 6
**Core ID:** 0
**CPU Cores:** 6
**FPU:** yes
**CPU-ID Level:** 13
**Address Sizes:** 46 bits physical, 48 bits virtual

# CUDA Memory Model

In CUDA all threads that are created in a device are placed into a grid. The threads within a grid are organised into blocks of threads. The smallest unit of scheduling is a warp which is fixed at 32 threads. The figure below shows the CUDA memory model:



[20]

Texture and Constant memory are read-only memory. Constant memory is fixed at 64kB and since the compiler knows it will remain constant, it is cached and therefore it can be beneficial to use it over Global memory.

Texture memory is slightly more complex to understand. Essentially texture memory is the same as Global Memory, however, it is treated differently to Global Memory. Like constant memory it is cached, however, it is particularly useful when memory accessed by a thread is spatially near memory accessed soon after. Most importantly, though, the memory locations can be spatially near to each other in two or three dimensions (not just horizontally as is the case with one dimension). Texture and Constant memory's lifetime extends from allocation to deallocation (of the actual memory) [21].

Global memory is the largest and slowest memory. It is shared by all blocks of threads within a grid and lives from allocation to deallocation (of the actual memory). It is useful for storing large complex structures such as arrays.

Shared memory is shared by all threads in a block. One block cannot access the shared memory of another block. Shared memory is faster than global memory but it quickly runs out of space as it is quite small in comparison. In addition, it only lasts as long as the block is alive. It can be used to store parts of global memory (such as parts of an array) that are used more than once by different threads within the same block.

Registers are the fastest and smallest memory, they are private to each thread and only last as long as the thread is alive. Since they can be filled up quite quickly, any excess is stored in a form of memory referred to as local memory which is equivalent in speed of access to global memory but only lasts as long as the thread is alive and so the programmer needs to be very careful in how he/she uses this memory.

To find out the position of a thread, CUDA provides the following properties that are callable within the kernel by every thread:

**ThreadIdx.x**: Gives the local ID in the x dimension of the thread within the block. '.y' or 'z' can be used to find the thread's y and z values.

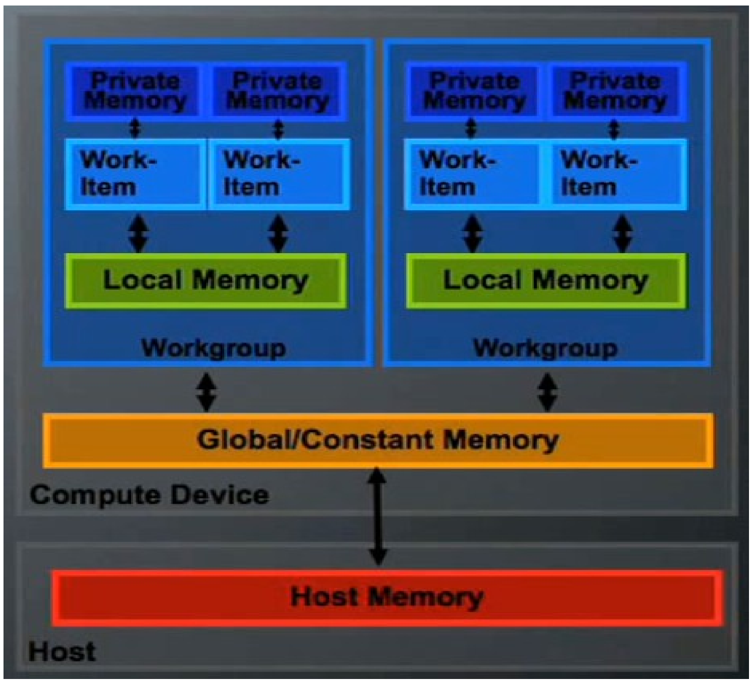
**BlockIdx.x**: Gives the local ID in the x dimension of the block within the grid. '.y' or 'z' can be used to find the block's y and z values.

**GridDim.x**: Gives the number of blocks within the grid in the x dimension. '.y' or 'z' can be used to find the number of  blocks in the y and z dimensions.

**BlockDim.x**: Gives the number of threads within a block in the x dimension. '.y' or 'z' can be used to find the number of threads in a block in the y and z dimensions.[20][22][23][28]

## OpenCL Memory Model

OpenCL follows a very similar model to CUDA except that it is more general so that it can support the variety of devices that it does. The memory model of OpenCL is shown below:



[19]

There are a few differences to notice in OpenCL's memory model. Firstly, shared memory is called Local Memory and secondly, there is no texture memory (as it cannot assume that a device has texture memory since it supports more than just GPUs). Threads are referred to as Work Items and Blocks are Workgroups. The methods that OpenCL provides within a kernel for each thread to access locational and dimensional information are given below:

**get_work_dim()**: Gives the number of dimensions in use (a number between one and three)

**get_global_size(0)**: Gives the number of work items (or threads) within the x dimension of the entire grid. If "1" or "2" is passed as an argument instead, then get_global_size will return the number of Workitems (or threads) in the y and z dimension.

**get_local_size(0)**: Gives the number of work items (or threads) within the x dimension of a Workgroup (or Block). If "1" or "2" is passed as an argument instead, then get_local_size will return the number of Workitems (or threads) in the y and z dimension.

**get_num_groups(0)**: Gives the number of Workgroups (or Blocks) within the x dimension of the entire grid. If "1" or "2" is passed as an argument instead, then get_num_groups will return the number of Workgroups in the y and z dimension.

**get_group_id(0)**: Gives the Workgroup (or block) number of the Workgroup within the x dimension of the grid. If "1" or "2" is passed as an argument instead, then get_group_id will return the number of the Workgroup within the y and z dimension.

**get_local_id(0)**: Gives the Workitem (or thread) number of the Workitem within the x dimension of the Workgroup that it resides in. If "1" or "2" is passed as an argument instead, then get_local_id will return the number of the Workitem within the y and z dimension of the Workgroup in which it resides.

**get_global_id(0)**: Gives the Workitem (or thread) number of the Workitem within the x dimension of the entire grid. If "1" or "2" is passed as an argument instead, then get_global_id will return the number of the Workitem within the y and z dimension.[19][24][29]

The table summarizes the methods used in OpenCL kernels to find spatial and dimensional information along with the equivalent method in CUDA:

| **OpenCL** | **CUDA** |
|---|---|
| get_work_dim(0) | Check gridDim.x, gridDim.y and gridDim.z to see which dimensions are active |
| get_global_size(0) | gridDim.x |
| get_local_size(0) | blockDim.x |
| get_num_groups(0) | gridDim.x |
| get_group_id(0) | blockIdx.x |
| get_local_id(0) | threadIdx.x |
| get_global_id(0) | blockDim.x * blockIdx.x + threadIdx.x |

# Basic CUDA programming model:

When writing a CUDA program, a programmer starts by declaring the variables, pointers or arrays that are going to be sent to the device. These are declared like normal C variables. This is followed by allocating memory on the device for the variables that are going to reside in device global memory. This is done using cudaMalloc. After allocating space for the variables (usually pointers or arrays), the programmer copies the values held by the corresponding host variable to the equivalent device variable using cudaMemcpy. This method is also used to copy values back to the host after the kernel has executed. When the kernel is launched, the programmer must specify the total number of blocks and threads per block within the “<<< >>>” parentheses. Additionally the programmer can use the optional third argument within the “<<<” parentheses to dynamically specify the size of a variable residing in shared memory by declaring a shared variable in the kernel without specifying its size (for example: __shared__ float myArray[]). However, the programmer can only dynamically allocate one variable (or array) in shared memory, so if he/she wishes to dynamically allocate more than one array in shared memory, the subsequent arrays must be indexed by adding an offset to the index where the offset is an array's size. If an array is statically allocated in shared memory, the programmer is not limited by the compiler in terms of how many arrays he/she may allocate in shared memory. The arguments to the kernel are specified in circular brackets after the “<” brackets. Single value variables are usually passed as is without being copied into device global memory since they are good candidates for thread private memory.[22][23][28]

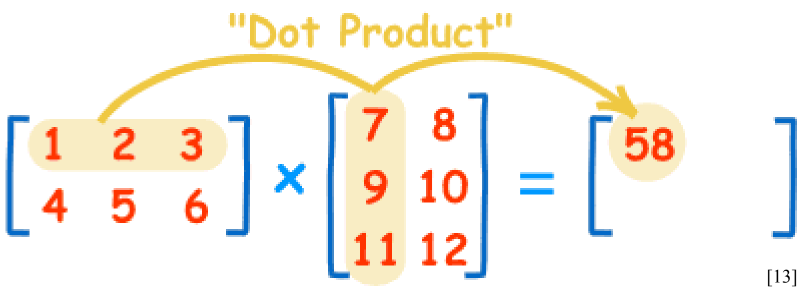
## Basic OpenCL programming model

With OpenCL, a reference to the device has to be created first. When doing this, the programmer must specify what type of device he/she would like a reference to (for example: CPU, GPU etc). Assuming that a device of the right type is found, the programmer must then create a context for the device. A context is essential for creating and managing objects to be passed to device memory among many other things.  If that is successful, the programmer must load the kernel(s) code from a file with a “.cl” extension and then compile it. As long as the compilation of the kernel doesn't produce any errors, a kernel can then be “extracted” from the compiled code using its name. The programmer must then create buffers to transfer memory objects (such as arrays) to device global memory. The size of the buffer (in bytes) should be the size of the object being transferred to the device's global memory. The next step is to set the kernel arguments so that the correct buffer or variable corresponds to the correct argument in the kernel. After this a queue must be created using the context and device. Finally, the kernel must be added to the queue for the device alongside a specification of the total number of threads that the programmer wants in the grid and the total number of threads in a block. After the kernel finishes executing, the programmer may read from the buffers created earlier and/or write new data to the buffers for the next time the kernel is called. [24][29]

## Matrix Multiplication

### Problem Description:

The matrix multiplication algorithm calculates an element of the product matrix '$M_{i,j}$' from matrix A and B by adding the product of every element in row 'i' of matrix A with every element in column 'j' of B. This is shown in the image below where the first element of the result matrix is calculated by doing the following: (1 x 7) + (2 x 9) + (3 x 11) = 58.



[13]

This process is repeated until all elements of the result matrix are calculated after which the result matrix is returned.

Matrices are used widely in the real-world in the fields of maths, engineering, finance and, ofcourse, computer graphics [27].

## Code Description:

The Pseudocode for the sequential algorithm is shown below:

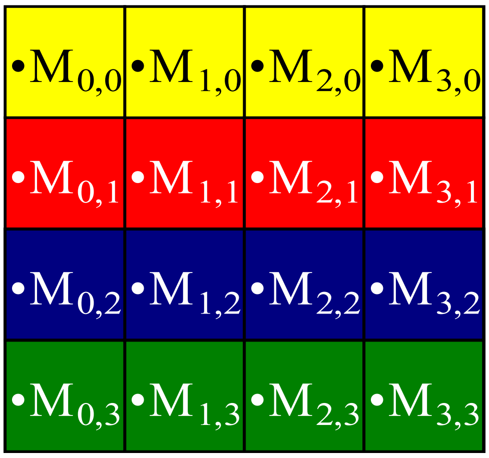
```
MatrixMultiplication(A[m][n], B[n][p]):
    Input: Two matrices A and B
    Output: The matrix from multiplying the two matrices A and B

    Array M[m][p] = new Array[m][p]();

    For i = 1 to m Do:
        For j = 1 to p Do:
            For k = 1 to n Do:
                M[i][j] += A[i][k] * B[k][j];

    return M
```

Now that we have the Pseudocode for the algorithm, we can decipher its computational complexity. If we assume that square matrices of width/height "n" are used for simplicity, its computational complexity becomes $O(n^3)$ due to its triple nested for loop. Consequently, as the matrices get extremely big, the time taken to perform this operation grows at an infeasible rate.

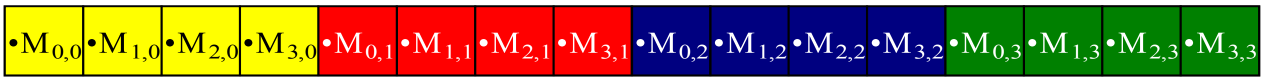
The first step to parallelising an algorithm such as this is to convert the matrices into one dimensional matrices since CUDA and OpenCL do not allow the programmer to declare two dimensional matrices in the global memory. The new sequential algorithm will now look more like this:

```
MatrixMultiplication(A[m x n], B[n x p]):

    Array M[m x p] = new Array[m x p]();

    For i = 1 to m Do:
        For j = 1 to p Do:
            For k = 1 to n Do:
                M[(i x m) + j] += A[(i x n) + k] * B[(k x p) + j];
    return M
```

While this may seem confusing at first glance (and probably even second glance), an easy way to figure out how to index one dimensional arrays as two dimensional arrays is to remember that the row index is multiplied by the width (number of elements in a row) and added to the column index. The diagram shown below should make this clearer:



•$M_{0,0}$ •$M_{1,0}$ •$M_{2,0}$ •$M_{3,0}$
•$M_{0,1}$ •$M_{1,1}$ •$M_{2,1}$ •$M_{3,1}$
•$M_{0,2}$ •$M_{1,2}$ •$M_{2,2}$ •$M_{3,2}$
•$M_{0,3}$ •$M_{1,3}$ •$M_{2,3}$ •$M_{3,3}$



•$M_{0,0}$ •$M_{1,0}$ •$M_{2,0}$ •$M_{3,0}$ •$M_{0,1}$ •$M_{1,1}$ •$M_{2,1}$ •$M_{3,1}$ •$M_{0,2}$ •$M_{1,2}$ •$M_{2,2}$ •$M_{3,2}$ •$M_{0,3}$ •$M_{1,3}$ •$M_{2,3}$ •$M_{3,3}$

The diagram above shows a two dimensional matrix laid out flat to convert it into a one dimensional matrix. A simple way to parallelise this algorithm is to let each thread compute one element of the result array (or matrix). In order to make the algorithm (especially the indexing) more understandable, the threads can be launched as a two dimensional set of blocks since the matrices being used are originally passed in to the host as two dimensional arrays. For the sake of simplicity, the algorithm assumes square matrices. This is shown in the Pseudocode below:

```
MatrixMultiplication(A[n x n], B[n x n], M[n x n], WIDTH):
    Input: Matrices represented as one dimensional arrays to multiply together, the result matrix
    (M) and the width of  a row in the matrices
    Output: The resulting matrix M (as a one dimensional array) from multiplying A and B (the
    input matrices)

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int value = 0.0;
    IF row < WIDTH and col < WIDTH THEN
        For k = 1 to n Do:
            value += A[row * width + k] * B[k * width + col];

        M[row * WIDTH + col] = value;
```

The two outer for loops have been removed as the kernel above is executed by every thread that is launched and the host tries to launch as many threads as there are elements in the result matrix.

Subsequently, for this to work, it has to be feasible to create as many threads as there are elements in the resulting matrix. The if statement provides error checking as there is a possibility that more threads are launched than there are elements in the resulting matrix (since we would prefer more rather than fewer threads to be launched) which would cause a segmentation fault as the extra threads would be writing to memory locations that haven't been allocated to the kernel.

The implementation of the Matrix Multiplication algorithm above doesn't make full use of the features provided by the memory model of CUDA and OpenCL such as shared memory.

Given that shared memory is very limited in size, matrix elements have to be loaded into shared memory a chunk at a time where the chunk is small enough to fit in shared memory. The reason shared memory is particularly useful in this instance is because each element in the matrices passed to the kernel are accessed more than once. For example, to compute $M_{0,0}$ the thread will have to access $A_{0,0}$ among other elements, however, this is also the case for the thread computing $M_{0,1}$ as it will also need access to $A_{0,0}$ among others. The shared memory kernel is shown below:

```
SharedMatrixMultiplication(A[n x n], B[n x n], M[n x n], WIDTH, TILE_WIDTH)
    Input: Matrices represented as one dimensional arrays to multiply together, the result matrix
    (M) the width of a row in the matrices and the width of a row each matrix to be loaded into
    shared memory.
    Output: The resulting matrix (as a one dimensional array) from multiplying A and B (the
    input matrices)

    __shared__ float Ashared[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Bshared[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int row = by * TILE_WIDTH + ty;
    int col = bx * TILE_WIDTH + tx;

    float value = 0.0;

    FOR i = 1 to WIDTH/TILE_WIDTH DO:
        Ashared[ty][tx] = A[row*WIDTH+i*TILE_WIDTH+tx];
        Bshared[ty][tx] = B[(i*TILE_WIDTH+ty)*WIDTH+col];
        __syncthreads();
        FOR j = 1 to TILE_WIDTH DO:
            value += Ashared[ty][j] * Bshared[j][tx];
        __syncthreads();
    M[row*WIDTH+col] = value;
```

For the sake of simplicity, it is assumed that WIDTH (the height/width of the input matrices) is exactly divisible by TILE_WIDTH (the height/width of the input matrix). The first 'for' loop iterates through each input array in blocks of size TILE_WIDTH loading the elements from A and B into shared memory. The __syncthreads() is a CUDA statement that doesn't allow any thread in a block to proceed until all threads in that block have reached that statement (So it only provides synchronisation to threads within the same block). This is done to ensure that all values have been loaded into shared memory before any thread of that block proceeds to perform a computation. The second __syncthreads() ensures that all threads of a block have finished performing their computation before replacing the values in shared memory.[28]

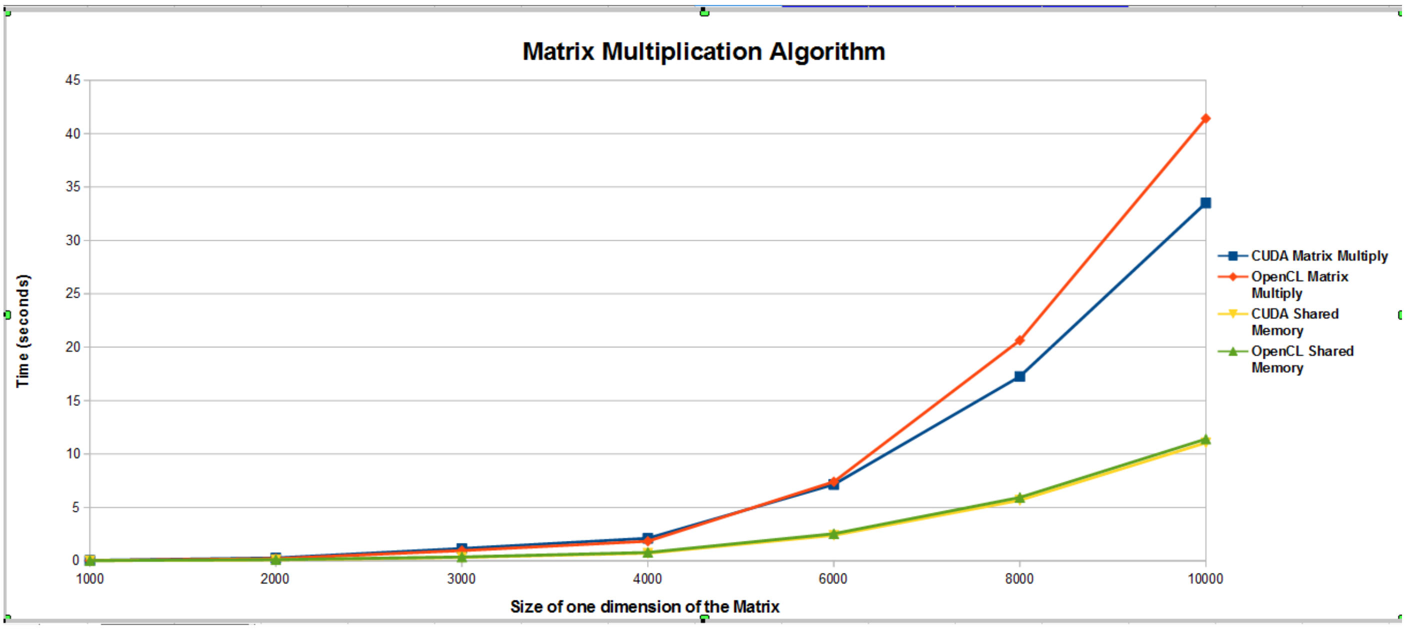
## Experiments performed:

The matrix multiplication algorithms both with and without the use of shared memory were implemented in CUDA and OpenCL. The BLOCK_WIDTH and TILE_WIDTH (in the case of the shared memory algorithm) are kept the same for both the CUDA and OpenCL implementations to keep it consistent. The BLOCK_WIDTH and TILE_WIDTH are both set to 16. The size of matrix is gradually increased from 1000 x 1000 to 10,000 x 10,000. The time taken for the kernel to execute is measured using a simple C function for both the OpenCL and CUDA implementations. The time taken for matrices of each size was measured ten times after which the average was taken and eventually plotted on a graph so that the results could be visualised.

To be certain that the code I had implemented was working as it should, I compared the results from the parallel version of the algorithm with that of the sequential algorithm using small matrices. I didn't check for an exact match since there could be discrepancies due to rounding errors. Instead I checked that the answers were within 0.00001 of each other. If the code is run without the validity check enabled, the output observed will be the time taken by the kernel to perform the matrix multiplication in the command line. However if the check is added in after the code has finished executing, the program will either report that everything was successful (if there were no errors in the matrix multiplication), or it will print out the elements in the result matrix that weren't correct.

## Results from experiments:

Figure 1 in Appendix A shows the graph obtained from the experiments performed. A smaller version of the graph is also visible below:

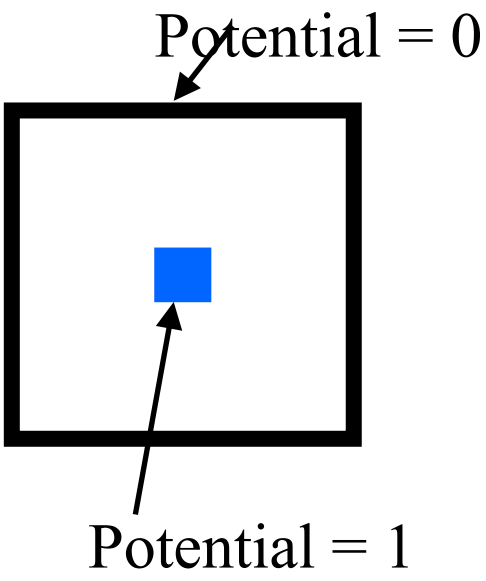


As can be seen in the graph above, the shared memory algorithm is much faster than the algorithm that simply uses global memory due to the fact that shared memory is much faster to access than global memory. What is also visible is the fact that OpenCL takes longer to execute than CUDA. This is due to the fact that as mentioned in other papers, the compiler automatically performs some optimisations that it does not perform on the OpenCL code. Furthermore, the OpenCL code is first translated into CUDA code before being translated into machine code which makes it unlikely that the compiler will be able to write as optimal (not to mention, elegant) code as me.

# Laplace Equation

## Problem Description

The problem being solved in this experiment is that of finding the electrical potential around a conducting object at a fixed potential inside a box whose borders are also at a fixed potential. The diagram below illustrates this:



In order to express this problem, three matrices are used. The phi matrix contains the current values of the solution, the old_phi matrix contains the values of the previous solution and the mask array shows which values can be updated. The mask array is “false” in the centre (where the object is placed) and at the border. The 'false' value indicates that those values should not be modified. The initial states of these arrays are shown below:

| | |
|---|---|
| 0 0 0 0 0 0 0 0 | F F F F F F F F |
| 0 0 0 0 0 0 0 0 | F T T T T T T F |
| 0 0 0 0 0 0 0 0 | F T T T T T T F |
| 0 0 0 1 1 0 0 0 | F T T F F T T F |
| 0 0 0 1 1 0 0 0 | F T T F F T T F |
| 0 0 0 0 0 0 0 0 | F T T T T T T F |
| 0 0 0 0 0 0 0 0 | F T T T T T T F |
| 0 0 0 0 0 0 0 0 | F F F F F F F F |
| **Phi/Old Phi** | **Mask** |

## Code Description:

Since the code for this algorithm uses quite a few variables of which most are pointers, I thought that it would be more palatable to just show the code as opposed to the Pseudocode for this algorithm (as they would probably be quite similar anyways). The entire program does quite a bit, in fact, it even outputs an image after executing (as mentioned previously). For the sake of simplicity, though, I'll focus on the code for the part of the program performing the actual updates. The sequential code to do this is shown below:

```
void performUpdatesSequential(float *d_phi, float *d_oldphi, int *d_mask, int nptsx, int nptsy)
```

```
{
  int x = 0;
  int xp;
  int xm;

  for (x = 0; x < (nptsx * nptsy); x++)
  {
    xp  = x + nptsx;
    xm = x - nptsx;
    if (d_mask[x]) d_phi[x] = 0.25f*(d_oldphi[x+1]+d_oldphi[x-1]+d_oldphi[xp]+d_oldphi[xm]);
  }

  for (x = 0; x < (nptsx * nptsy); x++)
  {
    if (d_mask[x]) d_oldphi[x] = d_phi[x];
  }
}
```

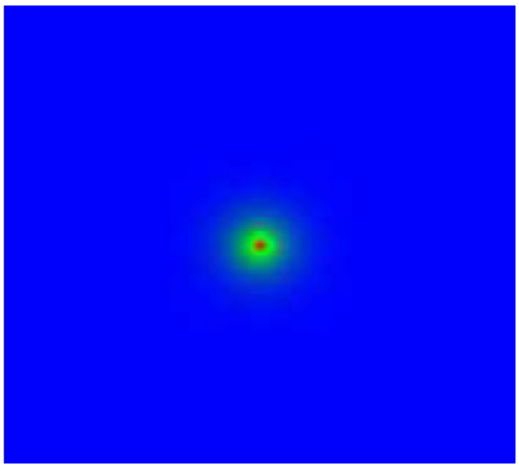
Since phi and old_phi represent two dimensional arrays (despite being one dimensional arrays), they have to be indexed in a manner similar to the arrays in the matrix multiplication algorithm. The variables “xp” and “xm” represent the elements above and below the current element under consideration in a two dimensional representation. As can be seen, a simple way to parallelise this code would be to remove the for loops. The kernels created as a result of this are shown below:

```
__global__
void performUpdatesKernel(float *d_phi, float *d_oldphi, int *d_mask, int nptsx, int nptsy)
{
   int Row = blockIdx.y*blockDim.y+threadIdx.y;
   int Col = blockIdx.x*blockDim.x+threadIdx.x;
   int x = Row*nptsx+Col;
   int xm = x-nptsx;
   int xp = x+nptsx;

   if(Col<nptsx && Row<nptsy)
      if (d_mask[x]) d_phi[x] = 0.25f*(d_oldphi[x+1]+d_oldphi[x-1]+d_oldphi[xp]+d_oldphi[xm]);
}
__global__
void doCopyKernel(float *d_phi, float *d_oldphi, int *d_mask, int nptsx, int nptsy)
{
   int Row = blockIdx.y*blockDim.y+threadIdx.y;
   int Col = blockIdx.x*blockDim.x+threadIdx.x;
   int x = Row*nptsx+Col;

   if(Col<nptsx && Row<nptsy)
      if (d_mask[x]) d_oldphi[x] = d_phi[x];
}
```

The reason for having two kernels in this instance is because there is no way to synchronise all threads in a grid of a kernel, therefore, the only way to ensure that all threads have finished using the old_phi array before replacing its values  is to update it in a separate kernel (as that way we can be sure that all the threads in the previous kernel have finished executing all the statements).

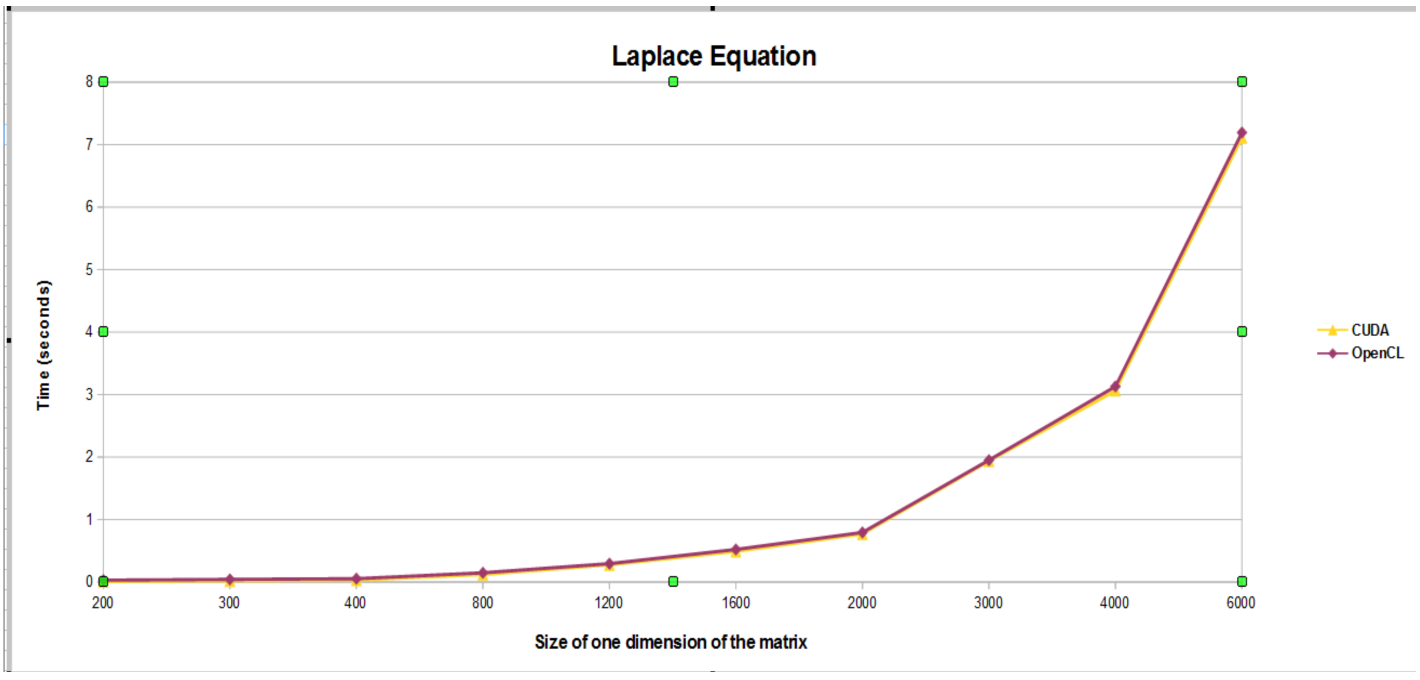
## Experiments Performed:

In order to compare the performance of the OpenCL implementation of the Laplace equation with that of the CUDA implementation of the Laplace equation, the size of the arrays were varied from 200 x 200 up to 6000 x 6000. The time taken for both kernels to complete was measured and since the kernels are called iteratively the time taken is added to a running total which was output once the program had completed. This was done ten times for each array size and then the average was taken and plotted. On running the algorithm, it should produce a file called outCUDA.ps in the same folder as the one within which the algorithm was run. If everything was successful, the image should look somewhat like the one shown below (This is also useful as a simple test to ensure that the algorithm ran correctly):



## Results Obtained:

Figure 2 of Appendix A shows the graph obtained by plotting the time taken on the y axis against the size of one dimension of the array on the x axis. A smaller version of the graph is also shown below:

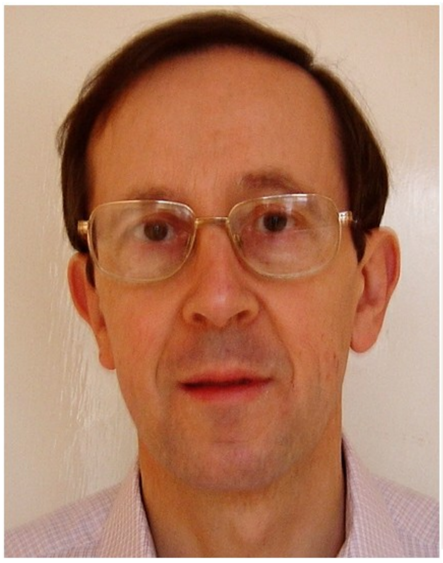


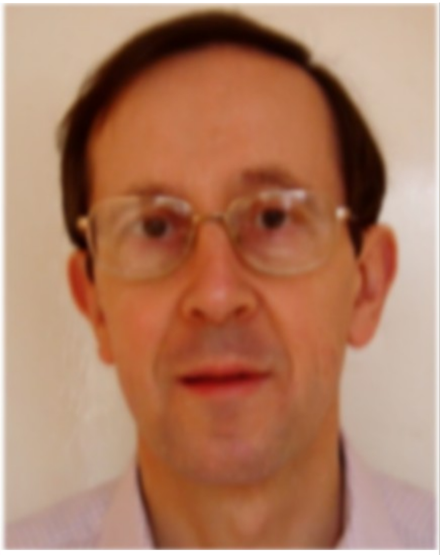
As is evident in the graph above, there is very little between the OpenCL and CUDA implementations of the Laplace equation. This makes sense since the amount of computation done in the kernels is very minimal. Furthermore, the computation done is very simplistic, as in, there isn't a huge mix of operations (such as multiplication, division etc) and the order of precedence for the operations is rather obvious. Due to this, there is very little the compiler can do to optimise the CUDA code to give it an edge over the OpenCL code.

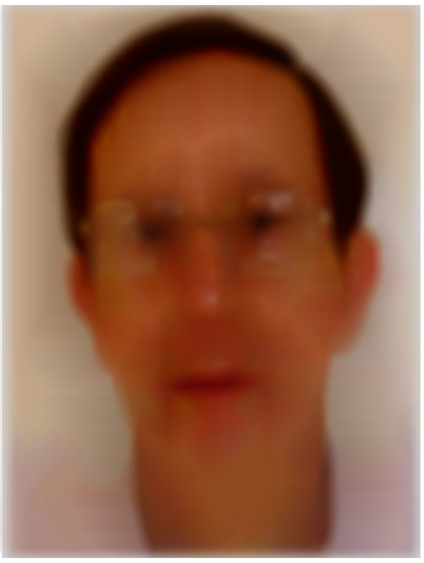
# Image Blurring Algorithm

## Problem Description:

The image blurring algorithm works by taking an image and replacing each pixel value by the average of its neighbours. The image blurring algorithm that I have implemented works solely with png images since it would require a large amount of time and effort to make it work with other formats as well. That would not only be beyond the scope of this project, but a separate project in its own right. The algorithm can blur an image to different degrees depending on how many times the blurring part of the algorithm is called. Shown below is the effect of running the code on a randomly chosen picture:


**Original Picture**


**Blurred ten times**


**Blurred hundred times**

One can simply use their eyes to judge whether the algorithm worked correctly, otherwise, for fine-grained details (such as minute differences in Red, Green and Blue values), there are plenty of free online tools which allow you to upload two images (in most formats) and carry out a detailed comparison to highlight any differences between the images. I compared the image produced by the parallelised code with that produced by the sequential code using one of the freely available tools provided online to ensure that my code was running correctly. For the paranoid amongst us, if you want to ensure that the tool really is doing what you think it should be, you can upload identical images to ensure the results are as expected, in addition, if you're still not convinced, you can upload two images which you know have differences (minor or major) just to ensure that they are highlighted by the tool as they should be.

## Code Description:

In order to process png images, the freely available lodepng class is used. The reason I chose to use this class library is because it is lightweight (you just need to include lodepng.h and compile lodepng.c) and portable, not to mention, easier to use than any of the other libraries available (as far as I know). Its biggest selling point for me was the fact that it didn't have to be installed on the server.

To begin with, the code finds the dimensions of the image as that will decide the size of the arrays. The main arrays used are R, G and B arrays (short for Red, Green and Blue). The RGB values of each pixel is loaded into the arrays and then the kernel is called to replace each pixel value by the average of its neighbours. Once it has computed the new values for each pixel, a second kernel is called to copy the values from the new R, G and B arrays to the old R, G and B arrays (This is in a different kernel for the same reasons that the Laplace equation used two kernels).

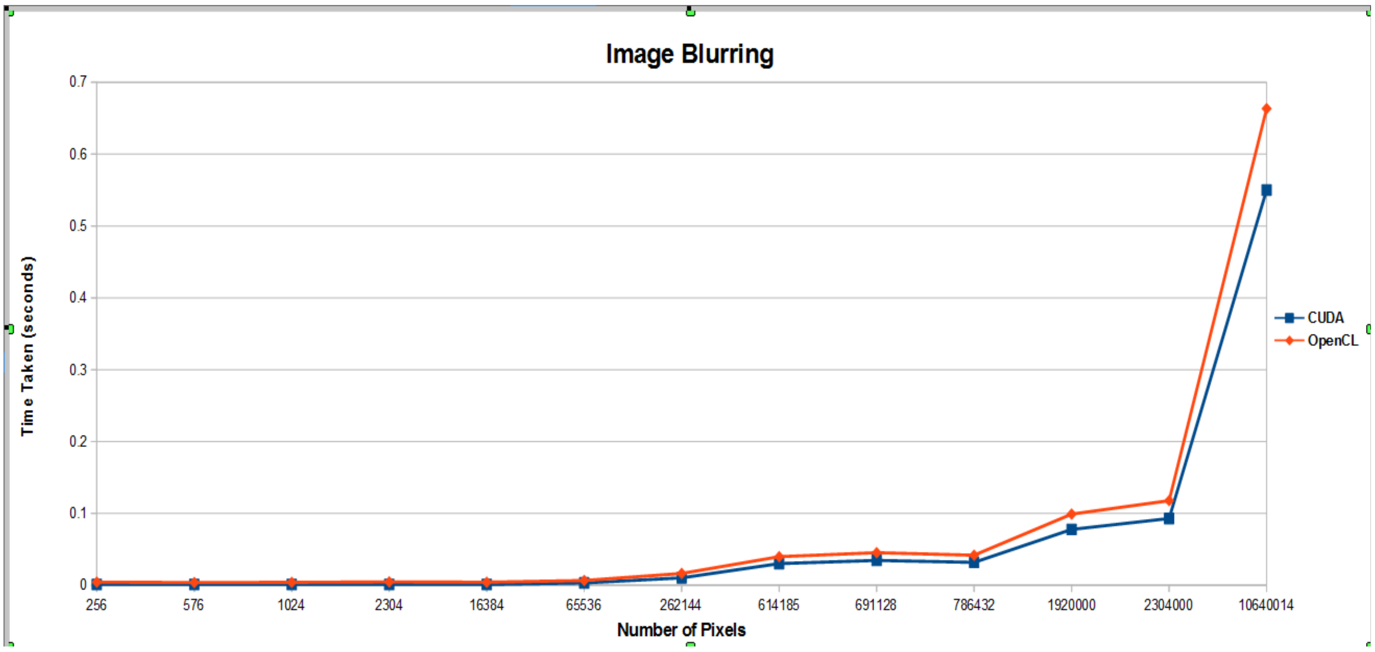
The actual kernel code for this is very similar to the kernel code for the Laplace equation (each thread deals with one pixel), however, one big difference is that there is a significant number of "if" statements to check if a thread is handling a pixel at a boundary, as, in those cases its value will not be replaced by the values of its four neighbours since it won't have four neighbours (but two or three). Unfortunately the large number of "if" statements tend to cause thread divergence since CUDA follows a Single Instruction Multiple Data (SIMD) execution model and therefore all threads of a warp must execute the same instruction at any time. This means that first the threads of a warp will execute the if part of the statement and in the next step the threads will execute the else part of the statement. thereby doubling the time it would take for the kernel to complete.

## Experiments performed:

Both the OpenCL and CUDA implementations of the algorithm were subjected to pictures of different sizes to analyse how they performed as the images got bigger. Like with the previous experiments, the time for the kernels was recorded for each image ten times and then the average was taken and plotted on a graph.

## Results Obtained:

Figure 3 in Appendix A shows how CUDA and OpenCL compare in the image blurring algorithm. The image shown below is a smaller version of the same graph:



Again, from the graph above, it is clear that the difference between the two lines for the most part is rather minute. Nevertheless, it is more pronounced than the difference in performance that was observed for the Laplace equation. However, given that there are a larger number of conditional statements and calculations, it doesn't come as too much of a surprise that the difference in performance between OpenCL and CUDA is more significant, as, when possible, the CUDA code is more likely to be optimised by the compiler to a greater extent (especially because I am using Nvidia's compiler) than the OpenCL code since the OpenCL code is first converted to CUDA by the compiler before being translated to machine code. Furthermore, the larger the kernel, the more pronounced these minor compiler optimisations (since there will be more of them).

# Molecular Dynamics

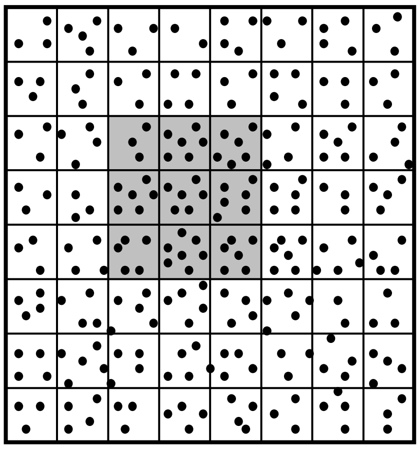
## Problem Description:

The molecular dynamics algorithm simulates the movements of "n" particles in a cube. The particles interact according to the following rules:

1) If two particles are close to each other, they repel one another
2) If two particles are far apart, they attract one another
3) If two particles are a certain distance $r_0$ apart, they have no effect on one another.

## Code Description:

Consider a single particle. In order to calculate the force on that particle, you could iterate over every particle in the space and add the force that each particle under consideration will have on that particle. However, if the size of the problem space is n, that would have a runtime of $O(n^2)$ which is very slow. A better way of calculating the force on a particle is to make use of the third rule mentioned above and only consider particles within an distance, $r_0$ , of the particle under consideration since all other particles outside this area would not have an effect on the particle under consideration. This way of processing the particles is illustrated in the image below:



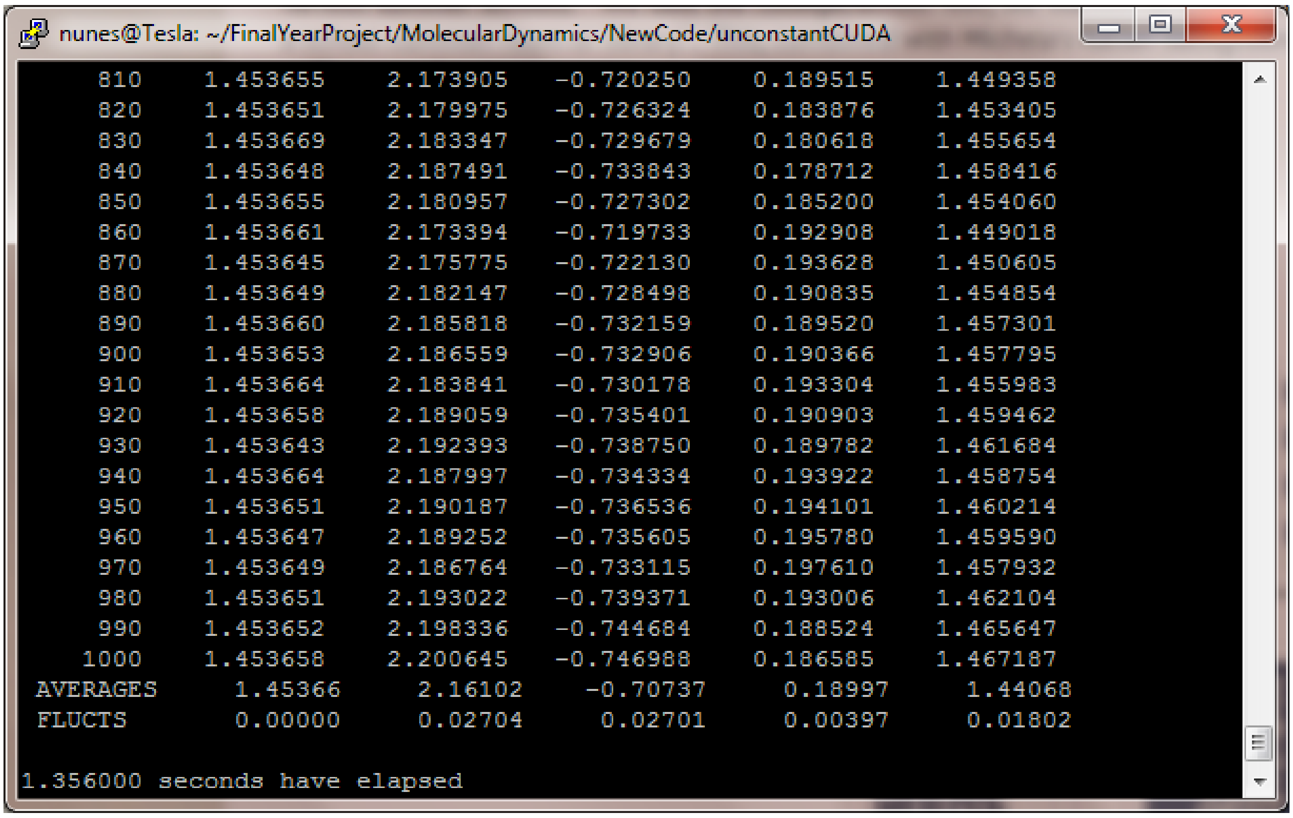
The shaded area represents the distance $r_0$. This problem has been implemented in two different ways. The first way assigns a particle to each thread and each thread calculates the force on its particle exerted by the particles around it. The second method assigns a cell (for example, one of the smallest boxes in the image above) to each thread and each thread calculates the force on the particles in its cell. In order to ensure that it does this in the most optimal manner possible, the thread places its cell in shared memory.

The code in the kernels for both methods of solving the problem is rather complex and involves a large number of mathematical calculations, conditional statements and even iterative statements. Due to this, the code is even more dependant on the compiler for optimal performance and as discussed previously, the more reliant it is on the compiler, the less likely it is for the OpenCL code to show a similar performance to CUDA code (especially when using Nvidia's nvcc compiler).

## Experiments Performed:

For both experiments the number of particles were varied to ensure fair comparison and the time taken by the kernel in the entire simulation was recorded. This was repeated ten times for each amount of particles after which the average was taken and plotted on a graph.

Like with some of the previous algorithms, I ensured that my code was running correctly by comparing the output produced by the sequential code with that produced by the parallelised version of the algorithm. A sample of the output received when the number of particles is 16384 is shown below:

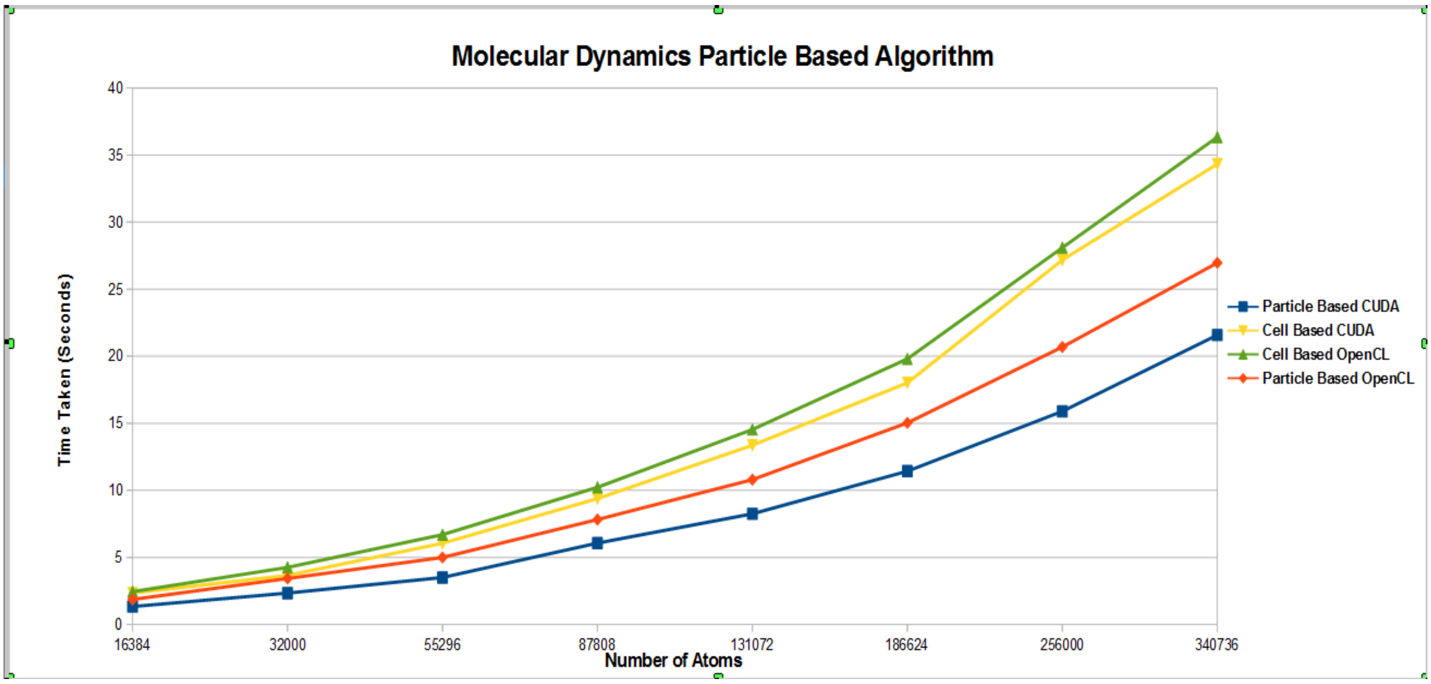


nunes@Tesla: ~/FinalYearProject/MolecularDynamics/NewCode/unconstantCUDA

```
     810    1.453655    2.173905   -0.720250    0.189515    1.449358
     820    1.453651    2.179975   -0.726324    0.183876    1.453405
     830    1.453669    2.183347   -0.729679    0.180618    1.455654
     840    1.453648    2.187491   -0.733843    0.178712    1.458416
     850    1.453655    2.180957   -0.727302    0.185200    1.454060
     860    1.453661    2.173394   -0.719733    0.192908    1.449018
     870    1.453645    2.175775   -0.722130    0.193628    1.450605
     880    1.453649    2.182147   -0.728498    0.190835    1.454854
     890    1.453660    2.185818   -0.732159    0.189520    1.457301
     900    1.453653    2.186559   -0.732906    0.190366    1.457795
     910    1.453664    2.183841   -0.730178    0.193304    1.455983
     920    1.453658    2.189059   -0.735401    0.190903    1.459462
     930    1.453643    2.192393   -0.738750    0.189782    1.461684
     940    1.453664    2.187997   -0.734334    0.193922    1.458754
     950    1.453651    2.190187   -0.736536    0.194101    1.460214
     960    1.453647    2.189252   -0.735605    0.195780    1.459590
     970    1.453649    2.186764   -0.733115    0.197610    1.457932
     980    1.453651    2.193022   -0.739371    0.193006    1.462104
     990    1.453652    2.198336   -0.744684    0.188524    1.465647
    1000    1.453658    2.200645   -0.746988    0.186585    1.467187
 AVERAGES      1.45366     2.16102    -0.70737     0.18997     1.44068
 FLUCTS        0.00000     0.02704     0.02701     0.00397     0.01802

1.356000 seconds have elapsed
```

## Results Obtained:

Figure 4 in Appendix A shows the results of the timing experiments performed. The image below is smaller version of the same graph:



From the graph above it is clear that the particle based algorithm is significantly faster than the cell based algorithm for both the OpenCL and CUDA implementations. This is probably due to the fact that with the cell based molecular dynamics algorithm, each thread will be doing quite a bit more work than with the particle based molecular dynamics algorithm and it will not be evenly distributed among the threads since the particles are constantly moving around.

Furthermore, the cell based molecular dynamics kernel is extremely complex in comparison to the particle based molecular dynamics algorithm and as we have discovered previously, the simpler the kernel, the more likely it is that the kernel will perform better. The cell based algorithm has many more nested conditionals thereby causing huge delays (One if statement will reduce the speed of an algorithm by 50% since all threads first execute the if part of the statement and then the else part of the statement).

In addition, while each thread in the cell based approach loads makes use of shared memory, it isn't the most efficient use of shared memory. This is because each thread loads the cell it is responsible for into shared memory and attempts to obtain the particles from its neighbouring cell from shared memory (loaded in by threads in its block), however, it can't do that for every neighbouring cell, as, if it is responsible for a border cell in a block, then at least one of the neighbouring cells that it has to gather particle information from will be the responsibility of another thread from a different block. Consequently, it will have to access global memory for that cell's data thereby adding to the time taken (not to mention the thread divergence).

Finally, due to the large amount of shared memory used by each block in the cell based approach, there is a limit to the size of each block and to the number of blocks that can run in parallel, in fact, it is roughly half that of the particle based approach. This significantly reduces the schedulers ability to optimise the work done per unit time (since it is limited in how much it can schedule in parallel).

Besides time taken, the efficiency of the result of parallelising the force routine can also be analysed in terms of Floating point Operations Per Second (Flops). In order to do this, Lets assume that the force routine performs 32 floating point operations and one square root operation to evaluate the force between two particles. Let N be the number of particles and M, the number of cells. We can assume that each particle interacts with approximately 27(N/M) particles (since there are approximately N/M particles in each cell and 3 * 3 * 3 other cells that it must interact with). Therefore the total number of floating point operations in the force routine approximates to:

N * (32 + SR) * (27 * (N/M))

SR is the number of floating point operations needed to perform a square root. Knowing this, the number of floating point operations can be approximated to: 1000 * N * N/M. Given that I already know that the force routine takes time t to run n times the final formula becomes:

Floating point Operations Per Second = (1000 * N * (N/M) * n) / t

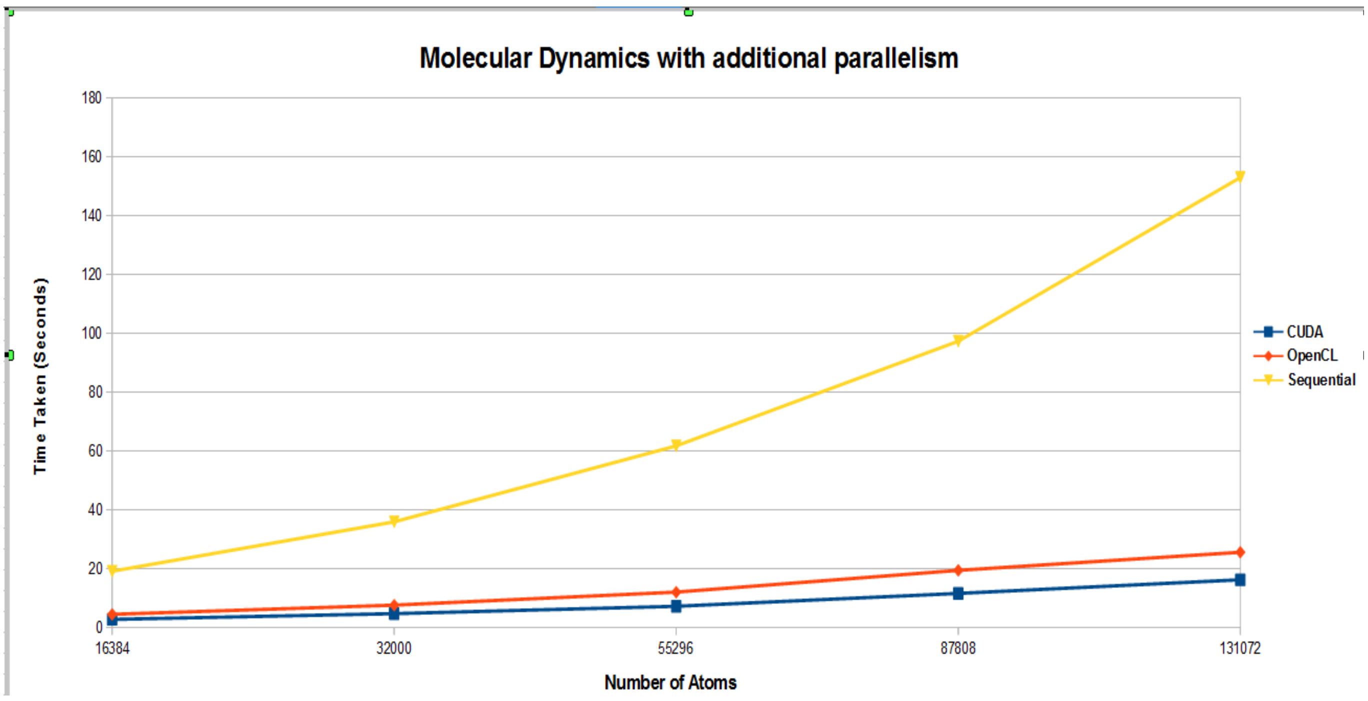
If the result from this is divided by the peak single precision floating-point performance for the GPU used (1.03TeraFlops) and multiplied with 100, we get the efficiency of the kernel.

Performing the calculation for the force routine using the timings obtained from the CUDA implementation of the force routine gives a 3.6% efficiency while with OpenCL the efficiency amounts to 2.7%.

## Extension of the Molecular Dynamics algorithm:

In the parallel algorithm above, the force routine was the only one that was parallelised. However, the molecular dynamics algorithm contains many more methods that lend themselves quite well to parallelism, namely, movea, moveb, scalet and a part of movout (called initialMovout in my code). Each of these methods contain iterative statements that loop over each atom. Therefore, like with the force routine an easy way to parallelise these methods is to simply assign one atom to each thread.  Additionally, to speed up the algorithm further, given that many of the kernels make redundant calls to the same value in global memory (such as fx[element]), these values can be put into thread private memory to reduce the total amount of calls to global memory.

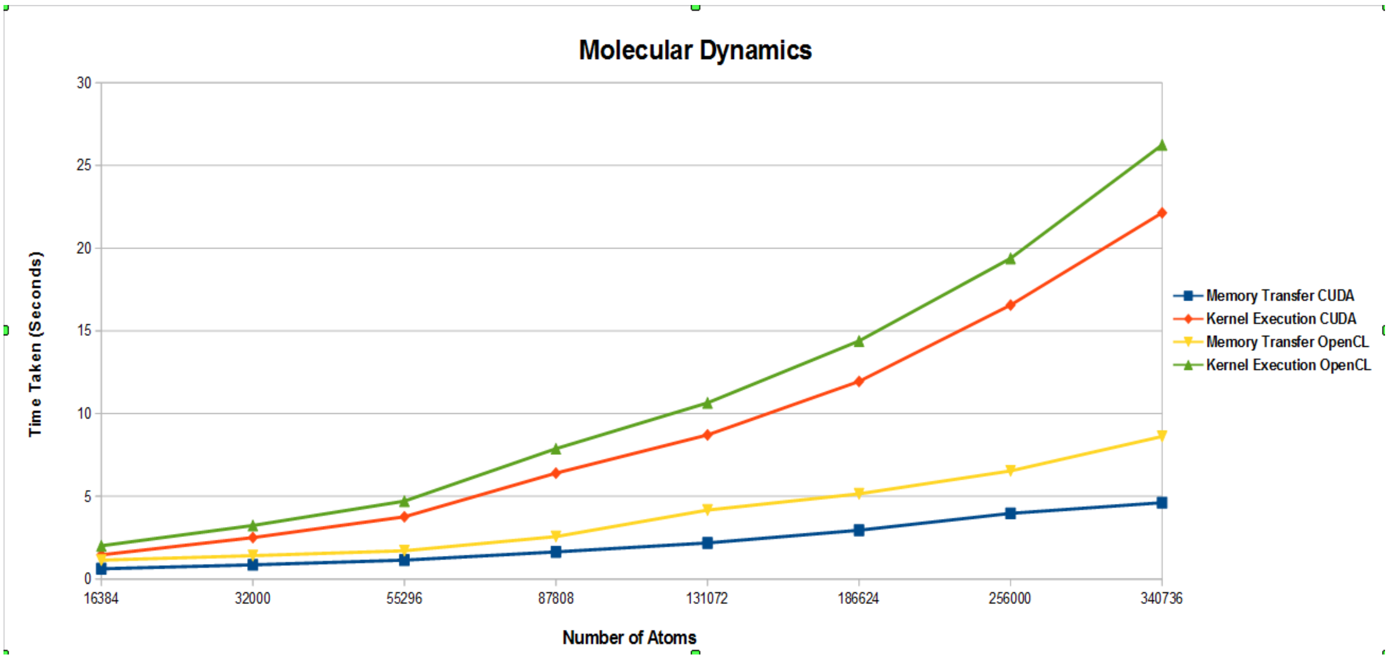
After implementing all of the above in both OpenCL and CUDA, the time for the entire outer for-loop to complete was measured and recorded. The outer 'for' loop includes all the kernels in addition to most of the methods doing the bulk of the computation for this simulation. The only statements not timed in this section are those which obtain input from the command line and set up the arrays. The algorithms parallelised in their entirety were compared with the sequential version in order to see whether any benefit is actually gained from parallelism and, if so, whether the benefit offsets the cost (in time). The graph produced from my experiments is shown in figure 5 of Appendix A. A smaller version of the same graph is shown below:



As is evident in the graph, parallelism does greatly reward those who take the time to add it to the molecular dynamics algorithm. Even the OpenCL version, despite being slower than the CUDA version, is still significantly faster than the Sequential implementation hence proving its worth.

## Detailed Performance Analysis:

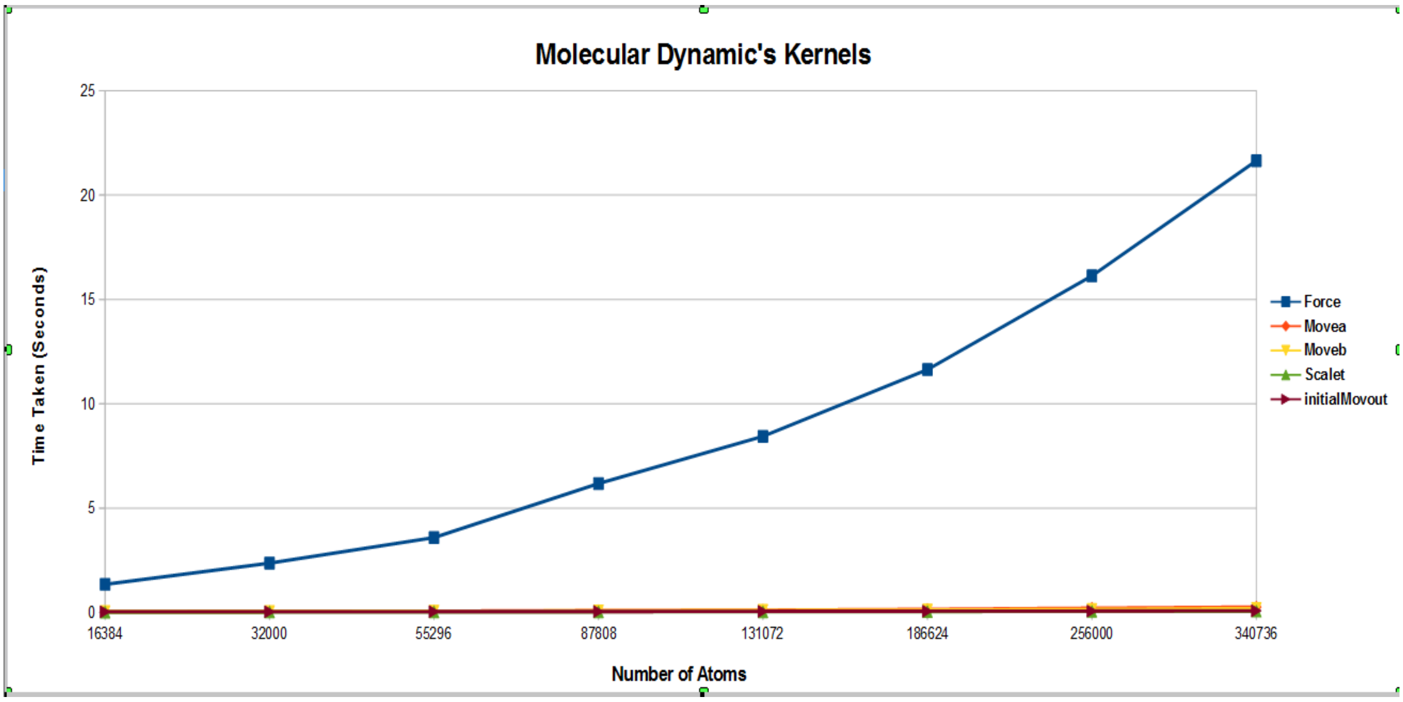
While I have analysed how OpenCL and CUDA compare with one another and with the equivalent sequential implementation (for the molecular dynamics algorithm), I haven't broken down the time taken by the parallel algorithm into its individual components. This would be useful to do as it would highlight which parts of the algorithm are the most time consuming. In addition, it may provide hints as to which parts should be focused on when trying improve the performance of a particular algorithm. This detailed analysis can be done on any algorithm, however, I have chosen to perform it on the molecular dynamics algorithm (in which the movea, moveb, scalet, force and part of movout have been parallelised) since it has the most individual parts of all the algorithms I have implemented. I broke down both the OpenCL and CUDA version in order to also get an idea as to how they compared and narrow down what was responsible for the difference in timings between the two. The graph I obtained from doing this is visible as figure 6 in Appendix A. A smaller version of the same graph is shown below:



As is visible, the time taken for the kernels to execute is much longer for both implementations thereby suggesting that, in order to improve performance, the kernels should be further optimised. Furthermore, for both the data transfer and kernel execution time, the OpenCL version is slower. While it is obvious (and has been explained previously) why the OpenCL version is slower when it comes to the kernel execution, it may not be so obvious as to why it is slower when it comes to transferring data from the host to device. Again, like with the kernels, the compiler may be able to optimise the memory transfer from host to device for the CUDA version. However, what also slows down the OpenCL implementation is that, unlike with the CUDA implementation, instead of writing to the previously allocated memory on the device after/before a kernel is called, I had to deallocate and then reallocate the memory on the device with the OpenCL implementation since the write buffer method refused to cooperate with me. This would have been much slower than simply writing to the memory once it has been allocated (as I would have liked to have done).

Another factor skewing the timings slightly is that with CUDA, it is very clear which statements belong to kernel execution and which statements belong to memory transfer, in OpenCL this is less obvious. With OpenCL, in order to do anything on the device, that command must be queued to the device. This includes commands to set certain parts of memory as certain arguments for a kernel. Therefore some may argue that these methods belong to the kernel execution since setting arguments is what is done when a kernel is called, however, others may say that this should be included with the time to transfer memory since it involves data as opposed to any execution. Due to this, it is hard to know how each part of the code should be classed (with respect to timing).

Once I knew what was causing most of the delay with the parallel code, I wanted to figure out which kernel in particular was adding to the kernel execution time and why since quite a few kernels are called in the molecular dynamics simulation. I only studied this for the CUDA implementation since doing so for the OpenCL implementation as well would have been redundant as they are practically carbon copies of each other (with respect to algorithmic method, not syntax). The results of my experiment are shown in figure 7 of Appendix A. A smaller version of the same graph is also shown below:



The graph makes it quite obvious that the force kernel seems to be taking up the most time of all the kernels. On closer inspection of the code, the reason for this becomes obvious. The force kernel is the most complex of all the kernels (in that it contains the most conditional and iterative statements), and as we have seen previously, massively parallel code does not take well to complex kernels. Furthermore, each thread does not just focus on one atom that it is assigned, but it has to use values from other atoms around it to calculate all the necessary values of the atom it is assigned. Then finally at the end, a reduction has to be performed to calculate the energy of the system which adds to the complexity and potential for thread divergence.

# CUDA Occupancy Calculator

On the topic of performance, Nvidia provides an Excel application called the CUDA occupancy calculator that can be used to determine the peak performance that one would observe for a kernel. On opening the CUDA occupancy calculator (in Excel), you enter the compute capability that you are compiling your code with and the shared memory size of your device. Both of these properties can be obtained by calling the cudaGetDeviceProperties method. The user then has to input their resource usage. This includes the block size (threads per block), registers per thread and the Shared memory per block. The first property is set by the user and the last two properties are obtainable by compiling the code with the following flag –ptxas-options=-v. This will print out for each kernel the registers used per thread and the amount of shared memory used. A sample of the output I received when compiling the molecular dynamics algorithm with this flag enabled is shown below:

```
nvcc -gencode arch=compute_20,code=sm_20  --ptxas-options=-v  -c moveb.cu
ptxas info    : Compiling entry function '_Z5movebPfS_S_S_S_S_S_fi' for 'sm_20'
ptxas info    : Function properties for _Z5movebPfS_S_S_S_S_S_fi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 21 registers, 2048+0 bytes smem, 96 bytes cmem[0]
```

The important parts have been highlighted. The yellow highlighted part indicates the method these details relate to (which in this case is moveb.cu). The blue highlighted part shows how many registers are used per thread (21) and the shared memory used per block (2048 bytes)

After filling in these details, the CUDA occupancy calculator provides many useful details. Notable among these, is the ideal block size, or in other words, the block size that will give the best performance. It also shows the occupancy of each multi-processor at that block size (expressed as a percentage). Another useful detail it provides is the maximum number of blocks that can be scheduled per streaming multiprocessor. In addition, it tells you what is limiting that amount (for example, the amount of registers or shared memory used) so that if you wanted to increase that amount, you know what needs to change in order to do that.

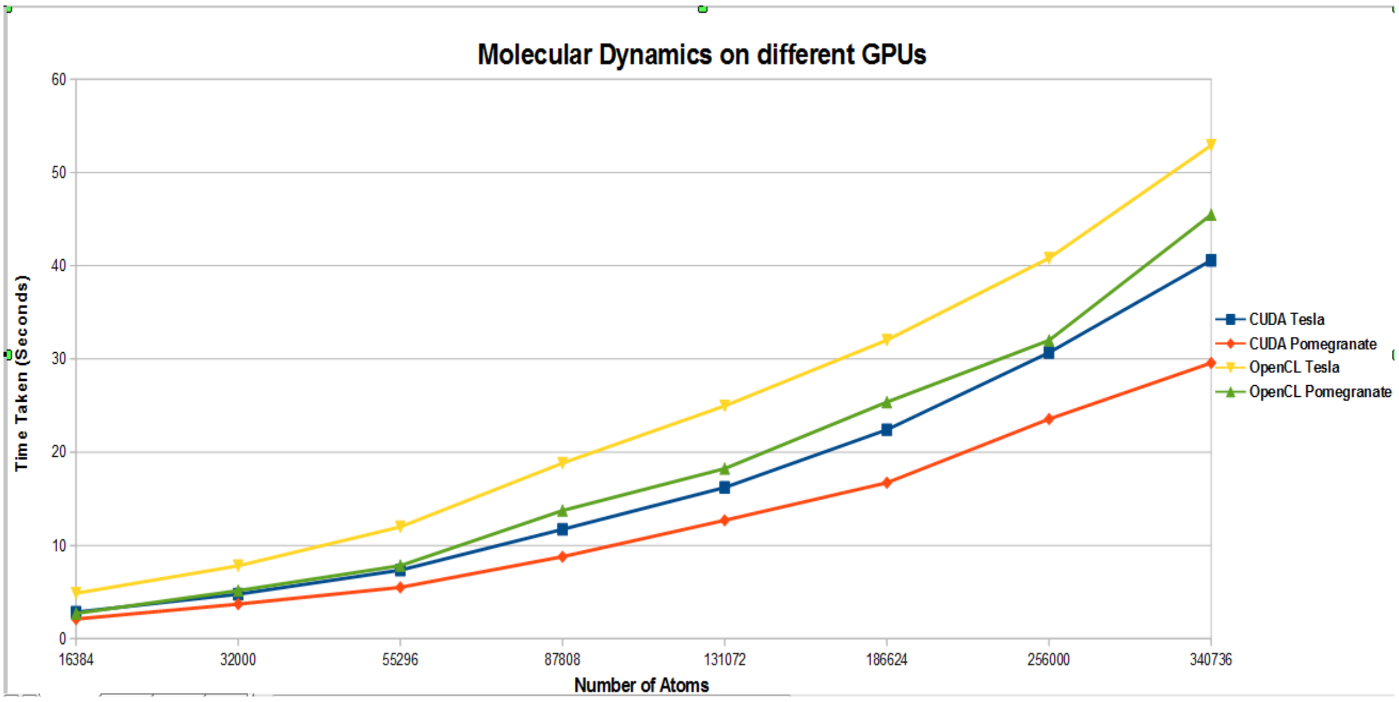
## Pomegranate vs Tesla

As mentioned previously, all of the algorithms were run on the machine called Tesla. However, I also have been given access to another machine equipped with an Nvidia GPU called Pomegranate (the machine, that is). Therefore I thought it may be interesting to see how the molecular dynamics algorithm compares when run on Pomegranate and Tesla. Just as a reminder, the specifications of both devices as provided by CUDA are shown in the table below:

| | **Tesla** | **Pomegranate** |
|---|---|---|
| **Major revision number** | 2 | 3 |
| **Minor revision number** | 0 | 5 |
| **Name** | Tesla C2070 | Tesla K20Xm |
| **Total global memory** | 1341587456 bytes | 1744371712 bytes |
| **Total shared memory per block:** | 49152 bytes | 49152 bytes |
| **Total registers per block** | 32768 (There are four bytes per register) | 65536 (There are four bytes per register) |
| **Warp size** | 32 threads | 32 threads |
| **Maximum threads per block** | 1024 | 1024 |
| **Maximum dimension 0 of block** | 1024 | 1024 |
| **Maximum dimension 1 of block** | 1024 | 1024 |
| **Maximum dimension 2 of block** | 64 | 64 |
| **Maximum dimension 0 of grid** | 65535 | 2147483647 |
| **Maximum dimension 1 of grid** | 65535 | 65535 |
| **Maximum dimension 2 of grid** | 65535 | 65535 |
| **Clock rate** | 1147000 kHz | 732000 kHz |
| **Total constant memory** | 65536 bytes | 65536 bytes |
| **Number of multiprocessors** | 14 | 14 |

|  |  |  |
|---|---|---|
| **Compiler** | release 4.0, V0.2.1221 of Nvidia's nvcc compiler | release 5.5, V5.5.0 of Nvidia's nvcc compiler |

In order to compare how the different GPUs performed, I ran the highly parallelised molecular dynamics algorithm on both machines and plotted the time taken for the outer for-loop to finish executing. The result from my experiment is shown in figure 8 of Appendix A. A smaller version of the same graph is shown below:



The graph above shows that Pomegranate is faster for both CUDA and OpenCL and the performance of OpenCL on Pomegranate is almost the same as that of CUDA on Tesla. Firstly, this highlights the importance of a good compiler in OpenCL and CUDA's case. As on Pomegranate the code was compiled with compute capability of 3.0 while on Tesla the compute capability was 2.0. Furthermore, since Pomegranate's compiler is more up-to-date, it is more likely to produce more optimal code.

Also, in terms of hardware, Pomegranate's GPU is much better than Tesla's GPU. The Nvidia website says that the GPU on Tesla has a peak single precision floating point performance of 1.03 TeraFlops (Floating Point Operations per Second) while the GPU on Pomegranate reaches 3.95 TeraFlops. In addition, the GPU on Tesla has 448 CUDA cores while the GPU on Pomegranate has 2688 cores. Finally, to hammer the final nail in the coffin, the GPU on Tesla has a memory bandwidth of 144 GigaBytes/second while Pomegranate's GPU memory bandwidth is 250 GigaBytes/second. These specifications alone should be sufficient to explain why the graph is as it is.

## My experience programming with OpenCL and CUDA

Before starting this project I was familiar with the basics of CUDA and High Performance Computing which proved to be extremely beneficial when it came to learning OpenCL. While most of the papers that compared OpenCL with CUDA reported that there is a one to one mapping between OpenCL and CUDA, I would strongly disagree with that. OpenCL has an extremely steep learning curve to mount and I definitely would not recommend that someone attempt to learn it before learning CUDA. A good analogy (of which I am the proud inventor) is that if we assume CUDA is Java, then OpenCL is C. This is due to the fact that OpenCL has to be general enough to be compatible on many devices from a variety of manufacturers. Many of the things that you can take for granted with CUDA (since it handles them for you) have to be explicitly specified in OpenCL. To provide some examples, in OpenCL you have to actually go through the trouble of creating a reference to a device (which doesn't simply involve calling one method) followed by dynamically loading and compiling the source code for the kernel (which, again, is a fair amount of work) which also adds to the total execution time. To give you an idea of the complexity of OpenCL, the matrix multiplication code in CUDA is approximately 105 lines of code, while the OpenCL implementation is approximately 274 lines of code.

What further complicates things is the way that OpenCL passes arguments to a kernel. Unlike with CUDA where you just specify them after the kernel name and kernel configuration parameters, you have to first create a buffer, queue the data onto the buffer (from a queue you have created previously) and finally tell the kernel which buffer corresponds to which argument. Not only does this add to the lines of code but it isn't as intuitive as CUDA's manner of passing arguments which is very similar to the C way of calling a function (and most other normal programming languages).

To further rub salt in the wound, I found that some of OpenCL's methods didn't work as documented. To provide an example, in relation to the molecular dynamics algorithm, since the arguments passed to the kernel changed every time the kernel was called, I attempted to use a method to write to the buffer corresponding to each argument. Unfortunately, after days (in the literal sense) of wrestling with my code to get it to function, I managed to fix it by free-ing and then re-creating the buffer with the new data as opposed to simply copying to it. One might argue that this would seem like an error on my part as opposed to one that is caused by OpenCL, however, in that case I challenge such a person to prove me wrong.

## Ease of Debugging

Code written for GPUs is hard to debug. There is no way around that at the moment and it's due to a number of reasons. When debugging sequential code, the first thing most developers try is printing out values of variables to find where the code is going wrong. This can be achieved on Nvidia's line of GPUs as long as the GPU is of compute capability 2.0 or higher. A few extra flags must be added when compiling the code but otherwise print statements can be inserted normally like they would be for simple C programs. The extra flags that need to be added when compiling the code are: -gencode arch=compute_20,code=sm_20. I have compiled my code on compute capability 2.0 which is why the values of both flags end with 20. If you were to compile it with a different capability, that value would have to be changed.

One thing to remember though is that every thread will call the printf statement which can quickly fill up your terminal with data. One way around this is to use an if statement and only print if the thread's ID is a certain ID. Another debugging trick is to make all threads store their values in an array and copy it over to the host where the values can be analysed. All this said, printf statements are nowhere near as useful as they are in sequential code simply because the error could be caused by one thread or certain threads which makes it very hard to detect (especially if you're only printing the values of a thread with a specific ID).

The second (and perhaps most important) way in which GPU code can be debugged is by checking the error code of each and every CUDA or OpenCL specific method called on the host. Thankfully each method returns an error code which as long as its '0' indicates no error. However, if it is ever anything other than '0', it can be beneficial to actually look up the error code to see what error the code corresponds to as it may put you on the right track when it comes to debugging. Another thing to remember is that  CUDA kernels do not return error messages when called. It is possible to  work around this by calling cudaPeekAtLastError() (which returns the last error that has been produced) after calling a kernel along with cudaDeviceSynchronize() (which ensures that the kernel has finished doing all its work). Both these methods return an error code that can be checked.

Unfortunately, though CUDA and OpenCL provides plenty of error codes, the error codes aren't particularly informative or generous. One particularly annoying property about them is that in order to figure out what error message the error number corresponds to, the developer has to print out the decimal value of each and every possible error that can be associated with that method. To provide an example:

```
printf("%d\n", CL_INVALID_PLATFORM)
```

In the print statement above CL_INVALID_PLATFORM is a constant integer. Therefore if the constant is the same value as the error code, then you can consult the documentation for more details on the error. This is the case with both OpenCL and CUDA.

When it comes to tools to assist in debugging, given that all my code was run on Nvidia's line of GPUs and compiled using Nvidia's compiler, I am familiar with some of the debugging tools provided by Nvidia. The first one is cuda-memcheck. This tool is a mixed blessing, it can tell you exactly which line the error is occurring on (if it is an error related to accessing invalid memory locations), however, it can also waste a huge amount of your time by incorrectly suggesting where it thinks the error resides. In addition, from personal experience, I've always found that when it does give me accurate information, it's information I already knew. Consequently, this should always be used with caution (or not at all).

The last debugging tool provided by Nvidia that I made use of was cuda-gdb. When using cuda-gdb, one should remember that if the program being debugged takes any arguments from the command line, these need to be hard-coded for it to function properly (as was the case with the molecular dynamics algorithm). Like cuda-memcheck, cuda-gdb is a mixed blessing. It can point you to the exact location of the error or somewhere else that has nothing to do with the error. In addition, it has the same problem of frequently stating the obvious. Therefore, it should also be used with caution (but preferably, not at all).

When it comes to online support, both OpenCL and CUDA have a sufficient amount of documentation, tutorials and books. Furthermore, there are plenty of forums for both languages where you can get help with demystifying some of the error messages. This was extremely helpful to me throughout the development process. It was also very encouraging to see the extensive support available for OpenCL despite its immaturity.

Personally, the debugging technique that has worked best for me through this project is to methodically scrutinise each line of code for errors multiple times over. As even when the debugging tools and online support are helpful, I've found the help to be extremely generic and therefore it only serves to narrow down your search for errors. As a result, it is my opinion that in order to actually figure out what is wrong with your program in particular, you must patiently sift through the code with a pencil and paper in hand.

Finally, in comparison to CUDA, it has been my experience that one is far more likely to encounter errors when using OpenCL. Partly because OpenCL is still rather immature and partly because of the sheer volume of code that is required even for the simplest of programs thereby increasing one's opportunity to make mistakes.

# Limitations

Initially, I had proposed to run one of my algorithms on a mobile phone, however, I was unable to implement that functionality for a number of reasons. Firstly and most importantly because I discovered that OpenCL was not supported on all Android devices and any iOS devices for that matter at this point in time (Refer to: http://streamcomputing.eu/blog/2011-08-19/is-opencl-coming-to-apple-ios/ and http://streamcomputing.eu/blog/2013-08-01/google-blocked-opencl-on-android-4-3/). Therefore it seemed like it would be much more work than I had anticipated. In addition, there was a meagre amount of documentation online for developing applications using OpenCL or CUDA. What I also hadn't realised is that I would have familiarise (or re-familiarise) myself with mobile application programming in a fair amount of depth which was far beyond the scope of this project.

# Future Considerations

When it comes to future considerations, the possibilities are endless. Mainly because performance is something that can always be adjusted and improved and since the bulk of this project concerns performance, one can obsessively spend months fiddling with parameters and code to observe minute (and occasionally large) changes in performance. Another reason why this project lends itself so well to future possibilities is that the technologies used are very new. As a result there are many intriguing and unexplored avenues in the massively parallel programming world that could be useful to delve into.

In relation to the algorithms that have been implemented in this project, most only use private memory, shared memory and global memory (if even that). For this reason there is potential for further gain in speed by making use of constant and texture memory. Both these memories are similar in nature to global memory but they are read-only and hence cached. Constant memory is fixed at 64KB while texture memory is limited to the size of global memory (since it is the same as global memory except that it is treated differently). To provide an example of its possible use, in the cell based molecular dynamics algorithm, the positions of the particles in the cells could be loaded into texture memory. Doing this would mean that when a thread has to access the particles in a cell belonging to a thread from another block, rather than consulting global memory, the thread could access the much faster texture memory. In addition, texture memory is spatially optimised. This means that cells that are near to the cell currently under consideration in two or three dimensional representations (but not necessarily one dimensional representations) are fast to access. [21]

Another opportunity for performance improvement in the cell based molecular dynamics algorithm relates to the way that the threads are launched. Currently the threads are launched as a one dimensional set of blocks, however, it may prove more beneficial to launch the threads as a three dimensional set of blocks. This is because the problem itself a three dimensional one and therefore to organise the threads in a manner similar to the problem might increase the use of shared memory and decrease the frequency with which threads have to access global memory (not to mention the fact that it would lend itself beautifully to texture memory).

In regards to the image blurring algorithm, in its current state, it is very restrictive. At present, it only works for png images which means that any image that you would want to blur must first be converted to a png image (if it isn't already in that format). It could be extended to accept images of other popular formats such as jpeg, tif and gif. However, even when it comes to performance the image blurring algorithm suffers from thread divergence on an extreme scale. One way I tried to combat this was by adding two extra rows and columns in the boundaries and then making every thread in the actual image boundary copy their values to these added boundaries. After that, the threads performed the calculation as usual except this time every thread had four values to compute the average from. This removed the need for any if statements whatsoever. Unfortunately, though, while this method produced no visible difference in the final picture when compared to the sequential algorithm, when the pixel values were analysed down to the pixel level (their Red, Green and Blue values) and compared with the picture produced by the sequential algorithm, the border values in the picture were found to differ slightly. As a result, I had to scrap that algorithm. I had initially hoped to come back to it, however, time was against me.

In regards to the Laplace equation, I don't feel that I spent enough time on it and that it was implemented in haste. In hindsight, it could probably benefit from the use of shared memory. It is hard to say for certain as it would require careful thought rather than the bull in a china shop approach, but there is a possibility that it could benefit from loading the old_phi array into shared memory for example.

On the language specific front, OpenCL provides the ability to use vectors of a variety of sizes. The advantage with vectors is that if a mathematical operation is performed on a vector (or between two vectors), the operation is applied to each element of the vector at the same time. Making use of this functionality could lead to a performance increase in the OpenCL implementations. To provide an actual example, vectors could be used in the matrix multiplication example where instead of multiplying one element from one matrix with that of another, four or eight elements could be multiplied together simultaneously to save time.

Another interesting possibility that wasn't fully explored would be to test the algorithms on mobile devices. From a performance perspective, it would be interesting to see how well they run on mobile devices and how their performance compares with that seen when the algorithms were run on computers. It would also be intriguing to see how the parallel mobile applications compare to their sequential equivalent and whether the speed up produced is equivalent (in ratio) to that seen on a computer. It would probably also be beneficial to report on just how easy it is to actually write OpenCL or CUDA enabled applications on mobile devices since there is very little documentation on that front.

In terms of portability, this project didn't assess OpenCL's ability to run on devices from multiple vendors. In addition, despite the fact that OpenCL's performance was evaluated,  this was only done for one type of device, namely, Nvidia's GPUs. OpenCL may show very different performance on GPUs from other vendors. Furthermore, it would be quite interesting to see how well OpenCL adapts to the other devices it is supported on such as CPUs and FPGAs etc.

Despite the variety of Parallel programming architectures available, this project focused on comparing OpenCL and CUDA. Currently there are other parallel languages available such as OpenACC and DirectCompute. One of these may potentially be the answer to the rising demand for a parallel architecture to exploit the power available in a co-processor. Furthermore, when it comes to other devices such as CPUs, it would be intriguing to see how OpenCL compares with languages such as OpenMP.

This project focused solely on using OpenCL and CUDA with the C programming language. Both CUDA and OpenCL are supported by Python and Java. It would be interesting to see how the Python and Java versions of the languages compared in difficulty and performance to that of the C-variety of OpenCL and CUDA.

# Conclusion

Parallel programming presents incredible potential for performance improvement provided the algorithm allows for it. However, even if the algorithm is a suitable candidate for parallelism, it still comes with a price attached to it. Parallel programming is not a programming model that most programmers are familiar with. In addition, it brings its own challenges with it that must be addressed to truly reap the benefits from Parallelism. These challenges include deadlock, where two threads cannot proceed without acquiring another resource each held by the other thread, and, non-determinacy, where the outcome of execution cannot be predicted as it depends on the order in which threads are executed.

To add to the difficulty, there are a considerable amount of low level challenges that most programmers would normally never have to (or care to) spare a thought for with most sequential programming models. One example concerns the use of memory. If too much memory is used, the algorithm will not reap all the benefits that there are to gain from parallelisation. The size of memory is very limited and programmers even have to be careful when declaring variables in the kernel as each variable will take up a register or two (of which there are very few). This is something many programmers will struggle with initially, as, with other languages programmers tend to be very generous with their use of variables, in fact, programmers are strongly reprimanded if they re-use a variable for two completely different purposes in sequential programming. However if that allows the programmer to declare one less variable, it is often encouraged in parallel programming. Therefore, parallel programming will seem extremely archaic to modern programmers who are used to having 12 to 14 GB of memory and 1 Terabyte of storage.

In addition to memory sizes, programmers need to be aware of the speed differences between the various memory types and, when possible, they need to use shared memory which is significantly faster than global memory. However the use of shared memory will usually require a significant overhaul of the original algorithm which can prove to be quite challenging for most programmers. Additionally, programmers need to come up with an elegant solution that makes efficient use of shared memory, otherwise it can be quite damaging to performance.

Nonetheless, despite the challenges it presents, programmers need to be familiarising themselves with parallel programming concepts and languages if they want to build applications that are utilising the full potential of the machine they are run on. This becomes especially true as Moore's law gets even closer to its inevitable death. The move towards parallel programming has been much slower than it should have been. This is probably because programmers are waiting for a solution that doesn't force them to drop the original frameworks and methodologies that they are familiar with and pick up something completely alien to them. While it is true that some paradigms are emerging that provide the ability to do that such as  OpenMP (to utilise multiple cores of a CPU) and OpenACC (to utilise a GPU for general purpose computing) by only requiring the developer to add directives to their source files that give the compiler hints about how to parallelise their code, programmers should embrace these frameworks with a generous pinch of salt and not rely solely on them for improved performance. As though this may be sufficient for more standard software where performance improvement is nice to have but not essential, critical applications (such as those used by banks) will be reluctant to rely on the compiler to provide them with the necessary speed up and would probably prefer to have it explicitly stated by the programmer so that they can have an assured quality of service.

When it comes to the programming models currently available, though CUDA is probably the most popular at the present time, it is limited to Nvidia cards. Since Nvidia's line of cards also happen to be amongst the most popular, this would seem to be the best choice of languages to learn since there isn't much demand for programming to GPUs from other providers. Despite that, it must be remembered that Nvidia's cards are only used in 1.4% of the mobile market (as off 2013) and it's the mobile market that is on the rise as opposed to the PC market (further highlighted by the fact that the mighty Sony have sold their PC business). To address this growing void, OpenCL has been introduced. While it is still not heavily used, that should not discourage one from learning it especially since Apple invented it and are trying to promote it. Knowing that, it would be foolish to ignore its potential for growth as it may soon start finding its way into mobile applications. Once this starts happening, anyone not employing the coprocessor to improve performance of their applications will struggle to match the competition. Mobile phones aside, given the volatility and speed with which the technology business is growing, it would be naïve to assume that Nvidia (and CUDA) will always be leaders in the area of Graphics. If they were to be ousted from their position in the Graphics world, programmers familiar with OpenCL could still sleep well knowing that it is not tied to a particular manufacturer. [35][36]

Nevertheless, even if OpenCL were never to take off, knowing OpenCL is likely to have a positive effect on one's CUDA programming skills (and other parallel architectures) since it makes you more aware of what is happening under the hood. Therefore, knowing a parallel programming model as generic and low-level as OpenCL is bound to reduce the barrier to entry to any other programming models that may spring up in the future.

On to the topic of the experiments I carried out in my project, while it is clear that OpenCL only occasionally reached the performance of that reached by the CUDA code, it wasn't far off on the other occasions and it was still much faster than the sequential version (as seen in the molecular dynamics experiments). Though OpenCL frustrated me more than CUDA with its indecipherable error messages (that's not to say that CUDA doesn't bless us with indecipherable error messages) and its relatively lengthy set up processes (in terms of lines of code), it did get easier with time and is undeniably worth the effort required to master it due to the benefits it provides in terms of portability and the understanding that one gains of GPU programming. It is common knowledge that the best programmers have the best understanding of what is happening to their code at a lower level, therefore, why should this be any different for parallel programming. To draw another parallel, while OpenCL is hard to learn, it is much easier to learn if one has already learnt CUDA just like C is much easier to learn if you already know Java. Additionally, I discovered that with experience I got even better at parallelising applications and this is partly due to the fact that many applications parallelised in exactly the same way. As a result, after having done it a couple of times, it becomes second nature since many of the changes that need to be made to different sequential algorithms are very similar. Therefore,  I would say that with experience, OpenCL is almost as easy as any other language.

Analysis of related work showed that when it comes to performance, the difference between CUDA and OpenCL is likely to be due to the compiler. In fact, some of the authors went so far as to manually add all the optimisations to their OpenCL code that the compiler automatically adds to the CUDA code in order to prove their hypothesis and as a result of doing that, they have subsequently observed identical performance from the OpenCL and CUDA implementations. While that probably isn't practical for most programmers in the real-world (since it required detailed analysis of the PTX code), it is worth remembering that OpenCL is still relatively new and therefore its compilers haven't matured quite as much as Nvidia's CUDA compilers. That being said, there may always be a slight performance difference since OpenCL isn't as easy for compilers to optimise given that it isn't as restrictive as CUDA and, unlike CUDA,  the compiler cannot always be told to assume that the code will be run on a GPU (since OpenCL supports multiple devices). Nevertheless, despite the performance limitations of OpenCL, it is still encouraging to know that it can reach the speed of CUDA as it presents hope for the future.

# Reflection

Given that I was quite comfortable with CUDA, I didn't expect to encounter too many problems trying to learn OpenCL especially since all of the papers I had read said that there was a one-to-one mapping between OpenCL and CUDA. Unfortunately, that was not entirely true. One possible reason for this discrepancy is that the authors would probably have been fairly familiar with OpenCL when writing their papers and I know that had I not kept a record of my initial struggles trying to understand OpenCL, I too probably would have been singing the same tune as the authors of the papers I read. The reason for my initial blindness to the difficulties I faced when trying to learn OpenCL is that once the steep learning curve of OpenCL's is finally scaled, programming with it is about as subconscious as breathing. On the other hand, I should point out that I don't disagree entirely with the authors. In fact, I agree that when it comes to programming the kernels there is almost a one-to-one mapping between OpenCL and CUDA. Despite my venom towards OpenCL's steep learning curve, I am extremely glad I did go through the trouble learning it as it gave me a much better understanding of what is happening under the hood in CUDA. Additionally it solidified some of the concepts of Parallel programming in general that I hadn't fully understood.

After learning how to use OpenCL, the next significant setback that I suffered was when I tried to write the image blurring algorithm to work with any png image. The library I had chosen to use (lodepng) was very poorly documented. The only documentation available were the comments in its header file and these were intended solely for people who were familiar with the field of computer graphics (which I was not) making it far from user friendly. What made things harder was that there was very little online support for lodepng meaning that I had to deal with most of the errors I encountered on my own. Despite this, I do not regret choosing to use lodepng to load and save png images as it did provide the huge benefit of being lightweight and not requiring installation. In addition, once I knew which methods I needed to use, and was able to get one of the methods to work, it was relatively straightforward to figure out how to use the rest of the methods. Nonetheless, my original statement still stands, in that, a bit more documentation would have saved me a significant amount of time.

As you may have guessed, this project taught me a huge amount about performance. This is mainly because I kept challenging myself to make my algorithms more efficient. I achieved this by constantly trying to replace if statements or reduce accesses to global memory in an attempt to cut off a millisecond. As a matter of fact, it started becoming an unhealthy obsession as I found myself even playing about with simple Java applications that I wrote (for other modules) to ensure that they were as optimal as could be. I even started researching Java threads in a desperate bid to reduce the runtime of my algorithms beyond what I originally thought technologically possible. The point I'm getting at is that I'm extremely grateful to this project for helping me think more about performance and give more importance to it when developing applications (whatever they may be for) as before this, I never even gave performance a thought.

In addition to performance, this project forced me to write code in a memory efficient manner. Before starting this project, I would quite happily conjure up variables that I may or may not use later. I was also content with bringing a sledgehammer to a nut with most problems by employing a wide variety of complex data structures to store all sorts of data that would have been quite happy in a slightly more lightweight storage facility. However I quickly discovered that if I took that approach for my project, I was severely punished for it. Consequently, to avoid punishment, I was soon recycling variables like cardboard in an attempt to keep the number of variables declared in a kernel to a minimum due to the profound effect on performance. Looking back, it is quite safe to say that if we were to measure the memory usage of my programs before I started on this project and compared it with my usage after (and during) this project, we would find that it had reduced by a considerable amount.

Technical skills aside, this project tested my self-motivation and self-discipline skills to their limits. This is because besides this project, I only had one other module and therefore only four hours of lectures per day (and sometimes less). Added to that, there weren't any hard deadlines that had to be met (other than the deadline for this report and the intial report). Due to this, it was sometimes easy to become complacent and lull yourself into a false sense of security about the time you had to complete the project. However, this is where the weekly meetings with my supervisor proved to be beneficial as I had to report my progress to him on a weekly basis. Furthermore, what helped the most in this department was that I was doing a project in a topic that I was truly interested in and hungry to learn more about. As a result, it never really felt like work as I enjoyed myself so much. As a result, in hindsight, I'm quite pleased the project's were left to be done only during the second semester of term as it has been good preparation for the real world where you may not necessarily be pushed to work. As a matter of fact, you may be trusted to get on with the work yourself. Therefore, in the absence of any hard deadlines or bosses, you would need to motivate yourself to work efficiently.

Time management is an essential skill to have in all walks of life and this project was no exception. Throughout the duration of this project, I was constantly consulting the schedule that I drew up in the first week. This was not only because my memory is as leaky as a sieve, but because I wanted to be sure that I was on track. When I managed to stick to my schedule, I didn't have much to do in the way of time management. However, when I met with hurdles during the implementation of the image blurring and molecular dynamics algorithms, I was forced to reorganise my priorities in my initial schedule and go through the heartbreaking process of taking some objectives out of my schedule because it simply wasn't feasible to do all that I had initially set out to. This process was educational in that I realised that despite my best efforts I always needed to be prepared to completely throw away my initial schedule for a new one. I also had to find a balance between marks gained/lost and time spent on a particular problem (or sub-problem of a problem) as it was easy to get lost trying to fix a small problem or shave off a couple nanoseconds at the expense of a large amount of time with very little (if any) gain in marks.

When it comes to theoretical knowledge, my understanding of how programming languages work in general and my awareness of their limitations has grown far beyond I thought it ever would. To provide an example, when implementing the molecular dynamics algorithm, I was completely stumped by the fact that while my program was working and producing output, the output it was producing was incorrect (in comparison to that produced by the sequential algorithm) despite the fact that it was doing all the mathematical operations that were required of it. When I had exhausted all debugging methods known to me (and, possibly, mankind), I pleaded with my supervisor to see if anything jumped out at him. After a considerable amount of analysis on both sides (mine and my supervisor's), we managed to deduce that the errors was due to the way the C language represented floats (to put it in the simplest sense possible). At first glance, this would seem like an unfortunate waste of time, but I am secretly thankful that I was the unfortunate victim of this obscure error due to the fact that I tried to learn about the inner workings of the C language as well as a large number of debugging tools (such as cuda-gdb and cuda-memcheck) in an attempt to find and fix this error. Bizarre as it sounds, I am indebted to this error for all that now know about C and CUDA as I would never have bothered to learn all that I did had it not been for this error.

In terms of this project as a whole, I feel the results are not a fair representation of the amount of work I had to put in to get them. To be more precise, I worked every day including late evenings, weekends, bank holidays and (take a deep breath) early mornings on this project. However, if one were to study my code and experiments, one would come to the conclusion that a project such as this would not require much effort. This is because much of my time was consumed debugging the code since code written to run on GPUs is excruciatingly difficult to debug (as mentioned previously). In fact, in relation to the amount of time spent debugging, the experiments almost took no time whatsoever. For this reason I would argue that, although this project may not have a great amount to show when it comes to quantity (in relation to code and documentation), it still deserves the respect and recognition of one that would since it would required loyal devotion from someone else to repeat what I have done (in my humble opinion).

# References

1. Kadhim Shubber. October 23, 2013. *Moore's Law is dead. The future of Computing* [Online]. The Connectivist. Available at: http://www.theconnectivist.com/2013/10/moores-law-is-dead-the-future-of-computing/ [Accessed: ]

2. Encyclopedia – Britannica. *Moore's Law* [Online]. Availble at: http://www.britannica.com/EBchecked/topic/705881/Moores-law

3. Investopedia. *Definition of 'Moore's Law'* [Online]. Available at: http://www.investopedia.com/terms/m/mooreslaw.asp

4. Bill Dally. April 29, 2010. *Life after Moore's law* [Online]. Forbes. Available at: http://www.forbes.com/2010/04/29/moores-law-computing-processing-opinions-contributors-bill-dally.html

5. Brad Chacos. April 11, 2013. *Breaking Moore's Law: How Chipmakers are pushing PCs to blistering new levels* [Online]. PC World. Available at: http://www.pcworld.com/article/2033671/breaking-moores-law-how-chipmakers-are-pushing-pcs-to-blistering-new-levels.html

6. *Moore's Law OR how overall processing power for computers will double every two years* [Online]. Available at: http://www.mooreslaw.org/

7. Nvidia. *What is CUDA* [Online]. Available at: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

8. Oak Ridge Leadership Computing Facility. *History of CUDA, OpenCL and GPGPU* [Online]. Available at: https://www.olcf.ornl.gov/kb_articles/history-of-the-gpgpu/

9. Joel Hruska. February 1, 2012. *The death of CPU scaling: From one core to many  - and why we're still stuck* [Online]. Available at: http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck

10. Khronos Group. *The open standard for parallel programming of heterogeneous systems* [Online]. Available at: https://www.khronos.org/opencl/

11. Stream Computing Performance Engineers. *What is OpenCL* [Online]. Available at: http://streamcomputing.eu/knowledge/what-is/opencl/

12. AMD. *OpenCL: The Future Of Accelerated Application Performance Is Now* [Online]. Available at: http://www.amd.com/Documents/FirePro_OpenCL_Whitepaper.pdf

13. mathsisfun.com. *Matrix Multiply* [Online]. Available at: http://www.mathsisfun.com/algebra/images/matrix-multiply-a.gif

14. Ching-Lung Su; Po-Yu Chen; Chun-Chieh Lan; Long-Sheng Huang; Kuo-Hsuan Wu, "*Overview and comparison of OpenCL and CUDA technology for GPGPU,*" *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on*, vol., no., pp.448,451, 2-5 Dec. 2012

15. Nitin Gupta. *What is CUDA Driver API and CUDA Runtime API and Difference In Between?* [Online]. Available at: http://cuda-programming.blogspot.co.uk/2013/01/what-is-cuda-driver-api-and-cuda.html

16. Jianbin Fang; Varbanescu, A.L.; Sips, H., "A Comprehensive Performance Comparison of CUDA and OpenCL," *Parallel Processing (ICPP), 2011 International Conference on*, vol., no., pp.216,225, 13-16 Sept. 2011
doi: 10.1109/ICPP.2011.45

17. *Evaluating Performance and Portability of OpenCL Programs.* Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa and Hiroaki Kobayashi1;

18. *Common Subexpression Elimination*: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, & Tools. Addison Wesley, 2nd edition, 2007.

19. Rob Farber. 27 October 2010. *Part 2: OpenCL – Memory Spaces* [Online]. Code Project. Available at: http://www.codeproject.com/Articles/122405/Part-OpenCL-Memory-Spaces

20. Jeremiah Van Oosten. November 25, 2011. *CUDA Memory Model* [Online]. 3D Game Engine Programming. Available at: http://3dgep.com/?p=2012

21. Nitin Gupta. *Texture Memory In CUDA | What is Texture Memory in CUDA Programming?* [Online]. Available at: http://cuda-programming.blogspot.co.uk/2013/02/texture-memory-in-cuda-what-is-texture.html

22. Nvidia. 2008. *CUDA Programming Model Overview* [Online]. Available at: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

23. Nvidia. *CUDA Toolkit Documentation*. Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#formatted-output

24. Khronos Group. *OpenCL Reference Pages*. Available at: https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/

25. Stephen Shankland. November 18, 2013. *Top500 supercomputer rankings await speed test shakeup* [Online]. CNET News. Available at: http://www.cnet.com/uk/news/top500-supercomputer-rankings-await-speed-test-shakeup/

26. John Morris. November 10, 2013. *SC13: Top 500 list the world's fastest computers*. ZDNet. Available at: http://www.zdnet.com/sc13-top500-lists-the-worlds-fastest-computers-7000023347/

27. Alexis Writing. *Simple Uses for Matrices* [Online]. Available at: http://www.ehow.com/info_12019829_simple-uses-matrices.html

28. Kirk B. David et al. December 28, 2012. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann

29. Matthew Scarpino. 2012. *OpenCL in Action*. Manning Publications

30. Karimi, Kamran et al. May 16, 2011. *A Performance Comparison of OpenCL and CUDA*.

31. Sanden, Jarno van der. August 11, 2011. *Evaluating the Performance and Portability of OpenCL*. Electronic Systems Group. Faculty of Electrical Engineering. Eindhoven University of Technology.

32. Sutter, H. March 30, 2005. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency In Software*. C/C++ Users Journal. Available at: http://www.gotw.ca/publications/concurrency-ddj.htm

33. Gillespie, C. S. January 25, 2011. *CPU And GPU Trends Over Time*. Available at: http://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/

34. Severance, C. *Loop Optimizations – Basic Loop Unrolling*. OpenStax CNX. Available at: **http://cnx.org/content/m33732/latest/**

35. BBC News. February 6, 2014. *Sony to sell PC unit and cut jobs*. Available at: **http://www.bbc.co.uk/news/business-26062084**

36. Jon Peddie. September 25, 2013. *Qualcomm Single Largest Proprietary GPU Supplier, Imagination Technologies the Leader in GPU IP, ARM and Vivante Growing Rapidly, According to Latest Report From Jon Peddie Research*. Marketwatch. Available at: **http://www.marketwatch.com/story/qualcomm-single-largest-proprietary-gpu-supplier-imagination-technologies-the-leader-in-gpu-ip-arm-and-vivante-growing-rapidly-according-to-latest-report-from-jon-peddie-research-2013-09-25**