Final Report

Introduction:

In this report I will discuss the results I obtained from implementing a variety of algorithms using OpenCL and CUDA and the conclusions I drew from comparing the runtime of both implementations of the algorithms. 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 transistors decrease in size 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%. Despite the eventual death of Moore's law, demand for faster computers has not seized.

To meet this growing demand, engineers have taken an alternative approach by adding more cores to a processor. Multi-core processors have proven to greatly reduce the time taken to perform a number of tasks, in addition, manufacturers of Graphics Processing Units (GPUs) have made it possible for programmers to offload general tasks to the GPU. This is commonly referred to as General Purpose computing on Graphics Processing Units (GPGPU). However, 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.

One of the software models 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 C/C++ language that allow the programmer to achieve parallelism.

Another software model that aims at standardising software development on GPUs is Open Computing Language (better known as, OpenCL). 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).

While OpenCL boasts of being able to run on GPUs from any vendor (For example: Nvidia, AMD), this project 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 usability and online support.

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 needs to provide reasonable performance benefits for the cost of implementation (in terms of time). Therefore, the best way to assess its usefulness is to compare it with a widely used but 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 discover whether it is worth the effort.

Approach to problem:

In order to ensure a fair comparison of both OpenCL and CUDA, a number of algorithms were implemented, namely, the matrix multiplication algorithm, the Laplace equation, the image blurring algorithm and the molecular dynamics algorithm.

To ensure that the algorithms were implemented correctly the results produced by the parallel version were compared with those produced by the sequential version to ensure correctness. The time measured was the time taken for the kernel to execute. For the sake of consistency a simple C method was used to measure the time taken for the Kernel to execute as opposed to using OpenCL and CUDA's timing mechanisms provided. 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 the name of 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 has the following specifications:

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 some of the code was run is called pomegranate. The graphics card in this machine has the following specifications:

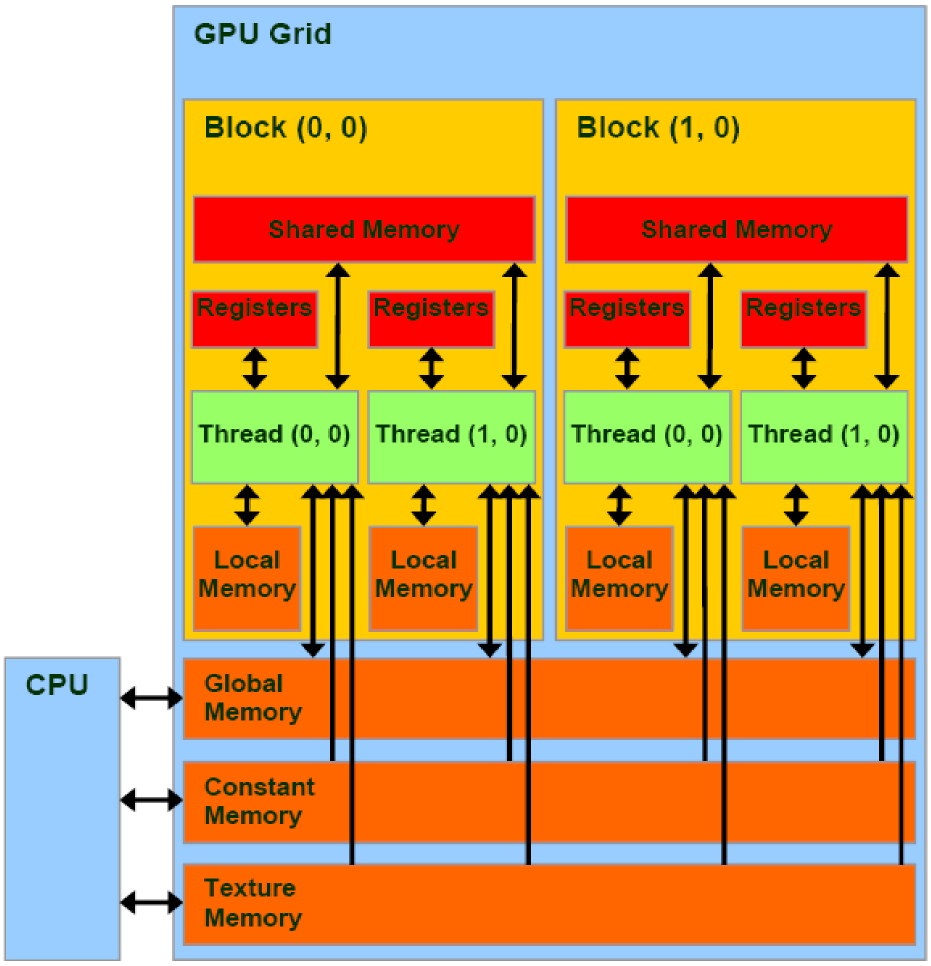
Major revision number: 3

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.

CUDA:

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:



Texture and Constant memory are Read-only memory. Constant memory is fixed at 64kB and since the compiler knows it is constant it is cached and therefore can be beneficial to use 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 one thread is spatially near memory accessed by other threads and most importantly, 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.

Global memory is the largest and slowest memory and is shared by all blocks of threads within a grid and lives from allocation to deallocation. Shared memory is shared by all threads in a block. One block cannot access the local 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. 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 local memory which is equivalent in speed of access to global memory but only lasts as long as the thread is alive.

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.

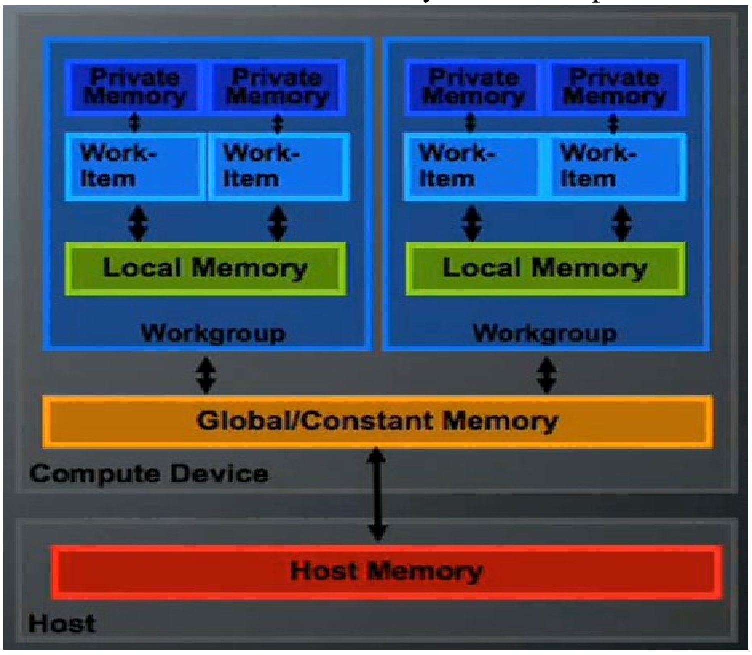
BlockIdx.x: Gives the local ID in the x dimension of the block within the grid. '.y' or 'z' can be used to find the block's y and z values.

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 dimensions1.

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.

OpenCL:

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:



There are a few differences to notice in OpenCL's memory model. Firstly, shared memory is called Local Memory and secondly that there is no texture memory (as it cannot assume that a device has texture memory). 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

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 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 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.

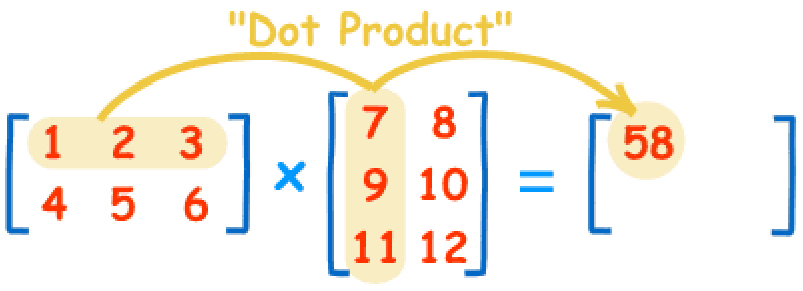
get_local_id(0): Gives the Workitem (or thread) number of the Workitem within the x dimension of the Workgroup that it resides in. If "1" or "2" is passed as an argument instead, then get_local_id will return the number of the Workitem within the y and z dimension of the Workgroup in which it resides.

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.

Matrix Multiplication:

Problem Description:

The matrix multiplication algorithm produces the 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 of 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.



Code Description:

The Pseudocode for the sequential algorithm is shown below:

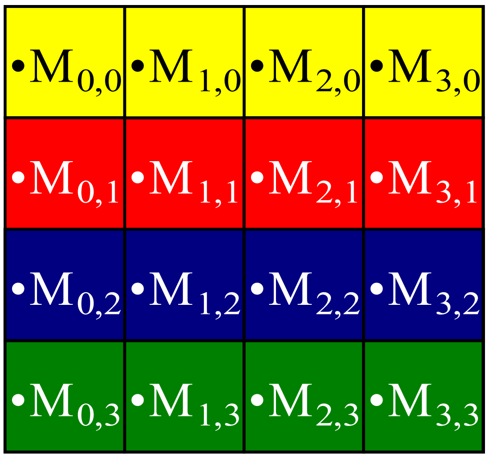
```
MatrixMultiplication(A[m][n], B[n][p]):
    Input: Two matrices A and B
    Output: The matrix from multiplying the two matrices A and B

    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
```

Due to the fact that it has a triple nested for loop, if we assume that square matrices of dimension n are used, its complexity becomes O(n^3). Therefore as the matrices get extremely big, the time taken to perform this operation gets unreasonable.

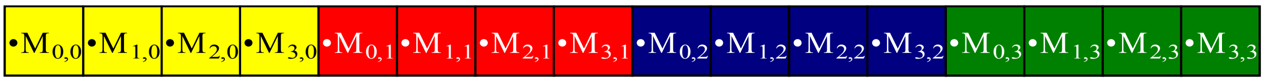
The first step to parallelising an algorithm such as this is to convert the matrices into one dimensional matrices since CUDA and OpenCL do not support two dimensional matrices. The new sequential algorithm will now look more like this:

```
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:



$\cdot M_{0,0}$ $\cdot M_{1,0}$ $\cdot M_{2,0}$ $\cdot M_{3,0}$
$\cdot M_{0,1}$ $\cdot M_{1,1}$ $\cdot M_{2,1}$ $\cdot M_{3,1}$
$\cdot M_{0,2}$ $\cdot M_{1,2}$ $\cdot M_{2,2}$ $\cdot M_{3,2}$
$\cdot M_{0,3}$ $\cdot M_{1,3}$ $\cdot M_{2,3}$ $\cdot M_{3,3}$



$\cdot M_{0,0}$ $\cdot M_{1,0}$ $\cdot M_{2,0}$ $\cdot M_{3,0}$ $\cdot M_{0,1}$ $\cdot M_{1,1}$ $\cdot M_{2,1}$ $\cdot M_{3,1}$ $\cdot M_{0,2}$ $\cdot M_{1,2}$ $\cdot M_{2,2}$ $\cdot M_{3,2}$ $\cdot M_{0,3}$ $\cdot M_{1,3}$ $\cdot M_{2,3}$ $\cdot M_{3,3}$

A simple way to parallelise this algorithm is to let each thread compute one element of the result array. In order to make the algorithm more understandable, the threads can be launched as a two dimensional set of blocks. 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 (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 this method is executed by every thread that is launched. Therefore, for this to work, there have to be 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 which would cause a segmentation fault as the extra threads would be writing to memory locations that haven't been allocated to the resulting matrix. This implementation of the Matrix Multiplication algorithm 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 is exactly divisible by TILE_WIDTH. 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 completed performing their computation before replacing the values in shared memory.

Experiments performed:

The matrix multiplication both with and without the use of shared memory was 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 each matrix size was measured ten time after which the average was taken and eventually plotted on a graph so that the results could be visualised.

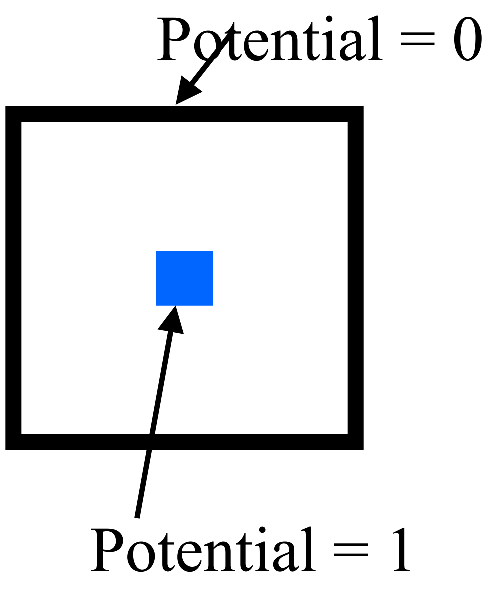
Results from experiments:

Figure 1 in Appendix A shows the graph obtained from the experiments performed. As can be seen, the shared memory algorithm is much faster than the algorithm simply using 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 in this case is that of finding the electrical potential around an 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 the border which indicates that those values should not be changed. 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 uses quite a few variables most of which are pointers, it seemed that it would be more palatable to just show the code as opposed to the Pseudocode for this algorithm. The entire program does quite a bit, it even outputs an image after executing. For the sake of simplicity 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. xp and xm represent the element 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 within a kernel, therefore, the only way to ensure that all threads have finished using the old_phi array before it is updated is to update it in a separate kernel.

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 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.

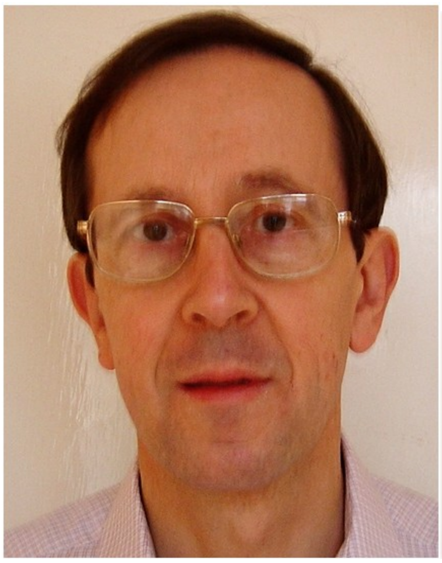
Results Obtained:

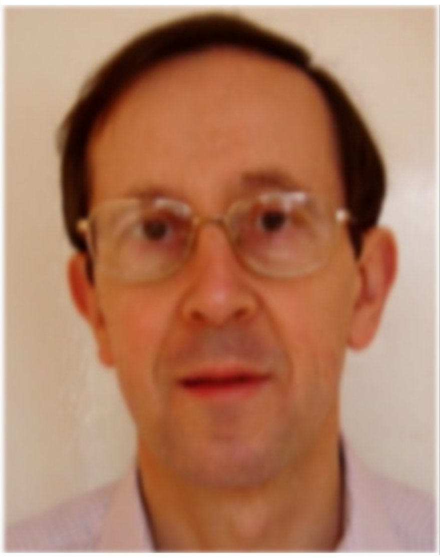
Figure 2 of Appendix A shows the graph obtained by plotting the time taken on the y axis against the size of one dimension of the array on the x axis. As is evident, there is very little between the OpenCL and CUDA implementations of the Laplace equation. This makes sense since the amount of computation 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 make it any more than slightly faster than the OpenCL code.

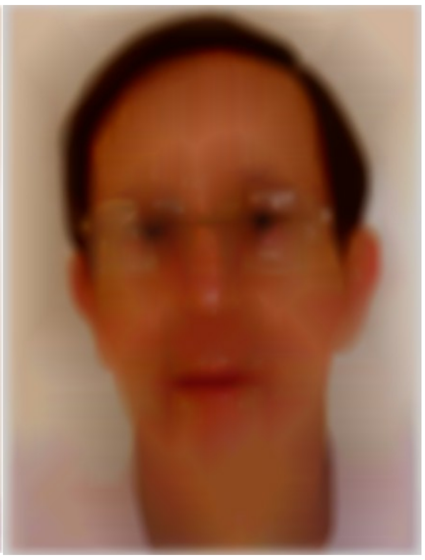
Image Blurring Algorithm:

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 an large amount of effort to make it work with other formats as well which is not only beyond the scope of this project but a 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. Below I've shown the effect of running the code on a randomly chosen picture


Original Picture


Blurred ten times


Blurred hundred times

Code Description

In order to process png images, the freely available lodepng class is used. The reason I chose to use this was because it was lightweight (you just need to include lodepng.h and compile lodepng.c) and portable, not mention easier to use than any of the other libraries available. 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 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. Unfortunately the large 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.

Again, the difference here for the most part is rather minute. However, given that there are a larger number of conditionals and calculations, it isn't too surprising that the difference in performance is more significant than what was observed for the Laplace equation as, when possible, the CUDA code will be optimised much more by the compiler than the OpenCL code.

Molecular Dynamics:

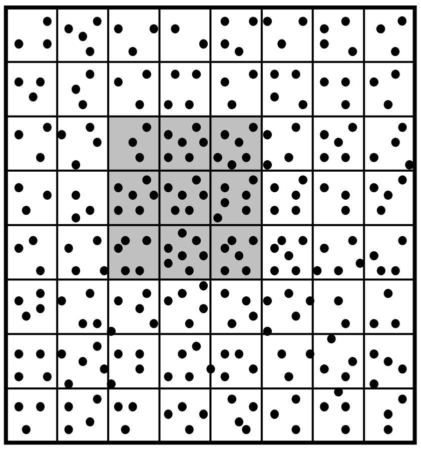
Problem Description:

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

1) If two particles and 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 use the third rule mentioned above and only consider particles within an r_0 distance 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 expressed in the image below:



The shaded area represents the distance r_0. This problem has been implemented in two different ways. The first way takes 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.

Experiments Performed:

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

Results Obtained:

Figure 4 in Appendix A shows the results of the timing experiments performed. 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 the particle based molecular dynamics algorithm and it will not be evenly distributed among the threads since the particles move in a random manner. 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%). In addition, while each thread in the cell based approach loads each 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), 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 gather particle information from will be responsibility of another thread and hence it will have to access global memory for that cell's data 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 thereby reducing schedulers ability to optimise the work done per unit time (since it is limited in how much it can schedule in parallel).

Programming with OpenCL and CUDA:

Before starting this project I was familiar with the basics of CUDA and the of High Performance Computing and this helped greatly 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 because since 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).

Personally I found some of OpenCL's methods didn't work as documented. To provide an example, for the molecular dynamics algorithm since the arguments passed to the kernel changed every time the kernel was called, I used a method to write to the buffer corresponding to the 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 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.

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 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, CUDA Driver API and CUDA Runtime API. The CUDA Runtime API is built on top of the CUDA Driver API and is easier to use and 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, however, it provides much more control. The algorithms on 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.

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 impressively in-depth analysis of all the reasons why a performance difference occurs when it does. They started by comparing the peak performance with OpenCL and CUDA. They started with the peak bandwidth of device memory and found that OpenCL proved to be between 2.4% - 8.5% faster than CUDA (depending on the GPU used). When it came to peak Floating Point Operations per Second (FLOPS), they noticed that OpenCL produced a similar performance, in fact, they even claimed that OpenCL was slightly better suggesting that it has the potential to use the Hardware as efficiently CUDA. The authors then compared the performance against a large number of real-world algorithms. Through doing this they came 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 CUDA provides access to Texture memory which allows to gain an edge over OpenCL in performance. In addition, to discover compiler differences, the authors analysed the code produced by the compiler and were able to conclude that the CUDA compiler had been optimised much more heavily. This becomes obvious by analysing the table below which shows the number of instructions each compiler generated for different classes of operations:

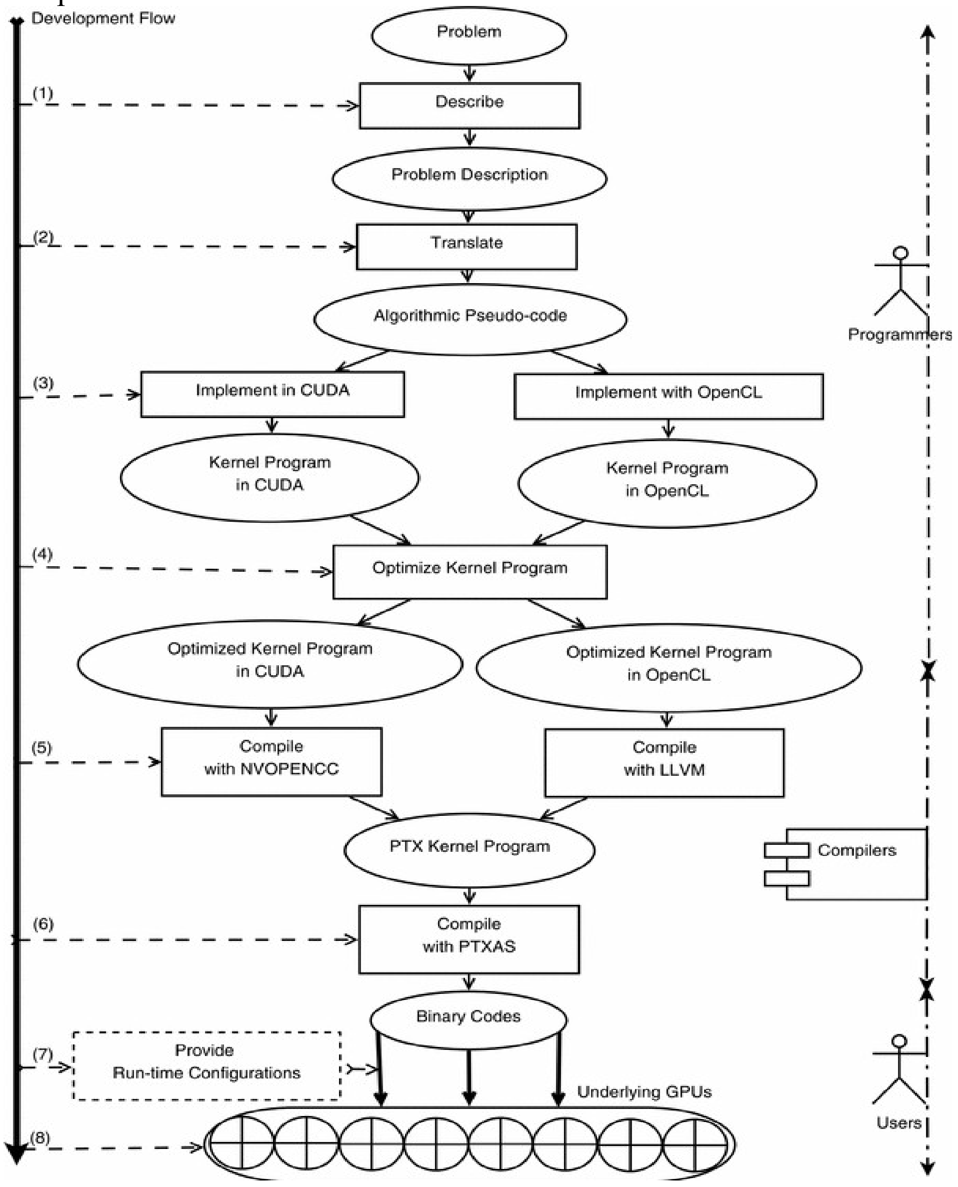
Class of Instruction	Number of CUDA instructions	Number of OpenCL instructions
Arithmetic	220	521
Logic Shift	4	163

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 weren't very costly.

Additionally, the authors notice that the OpenCL kernel takes longer to launch than a CUDA kernel which could explain why OpenCL seems to take slightly longer to execute a kernel than CUDA.

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. If these eight points are adhered to, the authors claimed that OpenCL will show similar performance to CUDA. From the eight points, four were the responsibility of the programmer writing the code, two were in the compiler's control and two were in the user's hands. A diagram giving more detail concerning each step in a fair comparison is shown below:



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

Komatsu et al. compares the performance of OpenCL and CUDA on a few algorithms and using different compilers and Graphics Cards. From the five algorithms that were tested in OpenCL and CUDA, the CUDA implementation 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. Given that the authors faithfully translated the CUDA algorithms into their exact representation in OpenCL, they looked in the PTX code (Parallel Thread Execution; the compiled code) for 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 while those for the OpenCL version had not been. They also noticed that for CUDA common sub-expression elimination (whereby a compiler evaluates the result for an expression once despite it 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. By manually applying all of these techniques to the OpenCL code, the developers noticed that the OpenCL version of the algorithm took the same amount of time as the CUDA implementation. They also 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). In addition, they concluded by performing their tests on a variety of GPUs and using different block widths. 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).

Future Considerations:

The algorithms that have been implemented only use private memory, shared memory and global memory (at the most). There is potential for further increase 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 meaning 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.

Another opportunity for performance improvement in the cell based molecular dynamics algorithm relates to the way 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.

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 be more specific, 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.

The Laplace equation was implemented in haste, and, in hindsight, it could probably benefit from the use of shared memory. It is hard to say for certain and it would require careful thought rather than employing a bull in a china shop approach but there is a possibility that it could benefit from loading the old_phi array into shared memory.

The image blurring algorithm is very restrictive. Currently 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 popular formats such as jpeg, tif and gif. In addition, 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 take the average of to replace their pixel value with thereby removing the need for any if statements. Unfortunately, though, while this method produced no visible difference in the final picture, when the pixel values were analysed programmatically and compared with a similar picture produced by the original method, the border values in the picture were found to differ slightly.

Limitations:

Initially, I had proposed to run one of my algorithms on mobile phone, however, I was unable to implement the 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 (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 very little 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.

Conclusion:

Parallel programming presents incredible potential for performance improvement, however, it comes with a price. Parallel programming is not a programming model that most programmers are familiar with it. In addition it comes with its own challenges that must be addressed to truly reap the benefits from Parallelism. These challenges include deadlock where two threads cannot proceed without another resource each held by the other thread and non-determinacy where the outcome of execution cannot be predicted as depends on the order in which threads are executed. In addition, there are more low level challenges that most programmers would never have to spare a thought for such as 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) which is something most programmers will struggle with initially as with other languages programmers tend to be very generous with their use of variables. 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 and it can prove quite challenging to come up with an elegant solution that makes efficient use of shared memory.

Despite the challenges it presents, programmers need to be familiarising themselves with parallel programming concepts and languages as the closer Moore's law gets to its death, the greater the need for software that exploits multiple cores and devices in a computer. This move towards parallel programming has been much slower than it should be and 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. Some paradigms are emerging that offer such functionality such as OpenCL (to utilise multiple cores of a CPU) and OpenACC (to utilise a GPU for general purpose computing) that allow you to simply add directives to your code that will essentially give the compiler hints about how to parallelise the code. While 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.

When it comes to programming languages available, though CUDA is probably the most popular at the present time, it is limited to Nvidia cards. Since Nvidia cards also happen to be amongst the most popular this would seem to be the best choice of languages to learn. However, it must be remembered that Nvidia's cards are not used in phones and it's the mobile market that is on the rise as opposed to the PC market which is in decline (further highlighted by the fact that the mighty Sony have sold their PC business). This is where OpenCL comes in. While it is still not heavily used (especially on the mobile market), given that Apple invented it and are trying to promote it, there is a good chance that it will 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. Learning OpenCL also has the benefit that one is not tied down to a particular manufacturer if the worst should happen in this volatile market. What's more is that knowing OpenCL is likely to have a positive effect on one's CUDA programming skills since they become more aware of what is happening under the hood.

The results from my experiments show that when it comes to performance the only real difference between the CUDA and OpenCL code is due to the compiler. Given that OpenCL is relatively new, its compilers haven't reached the stage that have been reached by Nvidia's CUDA compilers. It also isn't as easy to for compilers to optimise OpenCL since it isn't as restrictive as CUDA and unlike CUDA cannot always assume that the code will be run on a GPU. However, that isn't to say that OpenCL simply cannot reach the speed that CUDA is capable of as it is evident from the simpler of my algorithms that OpenCL performs just as well as CUDA. Furthermore, research carried out in previous papers all say that if the optimisations carried out by the CUDA compiler are manually implemented in the OpenCL algorithm, it reaches a similar (if not better) speed to CUDA. Although this isn't exactly practical (especially since it required the authors to methodically analyse PTX code), 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 most of the papers I read had said that there was a one-to-one mapping between OpenCL and CUDA. However, that could not have been further away from the truth. One possible reason for this discrepancy is that the authors would probably have been fairly familiar with OpenCL when writing their papers and 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 this is that once the steep learning curve of OpenCL is finally scaled, programming with it is about as subconscious as breathing. However, despite my venom towards OpenCL, I am extremely glad I did go through the trouble learning as it gave me a much better understanding of what is happening under the hood in CUDA and of Parallel programming in general that I would have had had I not chosen to learn OpenCL.

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 intended solely for people who were familiar with the field of computer graphics (which I was not) which made 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. Though, my original statement still stands, in that, a bit more documentation would have saved me a lot of time.