Interim Report:

Simulating High Frequency Trading in a Graphics Processing Unit (GPU)

**Abstract – This paper provides an interim project report on the progress of simulating high frequency trading in a GPU. It considers the appropriateness of a GPU for this task, provides an introduction to the OpenCL language used in the project and discusses the evolution of high frequency trading. In addition to this background, the paper sets out implementation and evaluation plans for the project. The evaluation section also includes some analysis of the characteristics of the preliminary model.**

# ****Introduction****

Stock trading has been around for a long time, but in recent years it has taken on new forms with the technological revolution of the age of computing. Trading can now be conducted electronically by humans and by computers employing algorithms. This project aims to simulate a market in which a number of trading algorithms conduct high frequency trading. This will take place in a GPU due to its highly parallel nature. The central objective is to create a market model consisting of an order book and a number of trading algorithms, which participate in the market. It will then be possible to observe the effects of high frequency trading on the market. Other key objectives of the project are to implement correctly the model to run the simulations and to maximise the speed with which the simulations can be run.

# Background

## Graphics Processing Units (GPUs)

GPUs first came about in the 1990s in response to the need to render 3 dimensional objects on 2 dimensional screens.[[1]](#footnote-1) Before GPUs in order to render images to screens the vertices (corners) of any object to be rendered had to be transformed to fit onto a 2D grid of pixels. This meant that the values for each pixel had to be calculated one at time. This is an incredibly slow method for calculating pixels, especially when performing the calculations for complex 3D objects with large numbers of vertices. The answer to this problem was the GPU. GPUs perform multiple operations at the same time thus allowing multiple pixels to be calculated simultaneously. They operate on the SIMD (Single Instruction Multiple Data) computing model. In this model only a single instruction is executed at a time, but there are a large number of processing units which share a single memory interface (where the data is stored). An instruction is executed in all of the processing units at the same time.[[2]](#footnote-2) This can be seen in Figure 1.

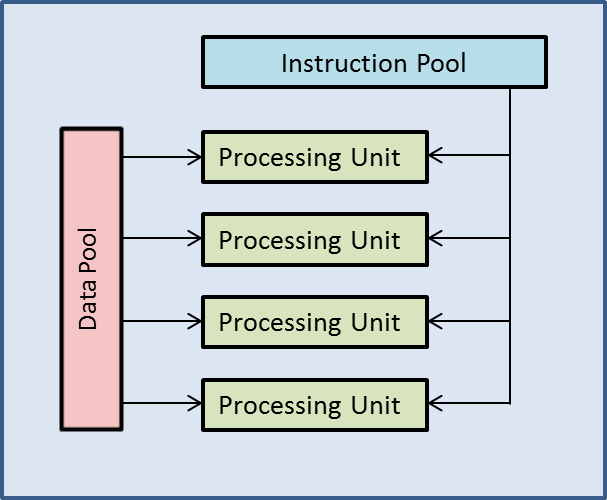


Figure 1

The pipeline for drawing pixels was built into hardware based around the SIMD model. This meant that the new GPUs were capable of many parallel operations. The graphics pipeline consists of several stages, which can be grouped into three main categories: geometry, rasterization and composition. Before the geometry stage begins, instructions and data are received into the GPU pipeline for processing.

The geometry stage begins with the GPU conducting vertex processing. This is where 3D vertices are converted into 2D space. Additionally other graphics effects such as lighting may take place here. Once vertices have been converted to 2D space, clipping takes place. This is where elements of the now 2D image that will not be visible (for example the back of an object) are deleted. Finally the vertices are collected and converted to triangles.[[3]](#footnote-3)

Next in the rasterization stage a number of different tasks are performed. First of all the triangles (or primitives as they are often known) produced by the previous stage are converted into pixels. Next occlusion culling takes place. This removes pixels that are obscured by other objects that are being rendered. Finally pixel shading takes place, which is where pixels’ colours, textures, depths and transparencies are established.

Finally after the rasterization stage all of the data assembled is combined into a composite image, which is then output to the screen.[[4]](#footnote-4) This can be seen in Figure 2.[[5]](#footnote-5)

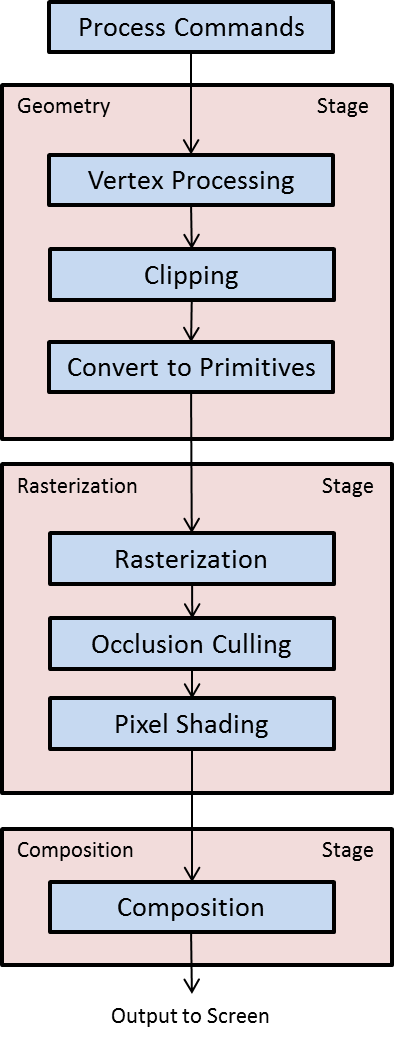


Figure 2

Initially the entire pipeline was implemented in hardware. However this was not to remain the case. One of the key drivers of the development of GPUs was the video game industry, which required ever more processing power. As the industry developed, the designers of video games wanted to have more control over the rendering of their games. As the graphics pipeline was all implemented in hardware, all of the vertex processing (vertex shading) and pixel shading was done in the same way across different games, resulting in them looking similar. In order to combat this, GPUs started to have programmable vertex and pixel shaders which enable programmers to have greater control over the graphics pipeline. This has also allowed GPUs to be used for purposes other than graphics rendering.

Their inherent parallelism is useful for many other scientific and commercial applications. A simple example of the power that a GPU brings to the table is for an N-body problem. This is a problem where there are individual particles which interact with each other within a certain set of laws. An example would be a gas, which is comprised of a large number of randomly moving particles which interact with one another. When done in a CPU the calculations for each particle must be done serially which will take a very long time for large numbers of particles. With a GPU the speed of the process can be increased by at least 50 times[[6]](#footnote-6). This increase was achieved with a GPU that was released in 2006.[[7]](#footnote-7)

Vertex and Pixel shader programming can be accomplished in a number of different languages including HLSL (High Level Shader Language) or OpenGL Shading Language.[[8]](#footnote-8) Additionally other languages for taking advantage of the parallelism of GPUs have been developed. For their GPUs NVIDIA, a major GPU maker, have developed CUDA[[9]](#footnote-9). Working on a larger number of platforms than CUDA is the OpenCL language. It is compatible with GPUs from the three main manufacturers: Intel, NVIDIA and AMD.

## OpenCL

OpenCL allows the use of various pieces of hardware including both CPUs and GPUs for specific tasks. In this project OpenCL code will be implemented for an AMD Radeon 6970[[10]](#footnote-10) GPU. However the way that this parallelism can be implemented in OpenCL is different to parallel programming in another language, for example, C#. In C#, in order to introduce concurrency into a program one must use threads (or similar objects such as tasks). The implementation of the multi-threading can be as simple as declaring a new thread variable, giving it an algorithm to run, setting it off and instructing it to re-join the main thread once complete. This is only a very simple example. Multi-threaded programs can become very complex with issues such as memory access (where more than one thread tries to read/write to the same piece of memory at the same time) or deadlock (where multiple threads are waiting for each other to fulfil a condition before continuing with no possibility of the condition being fulfilled). Problems like these will not be detected by the compiler.

In OpenCL the parallelism is implemented when a kernel is compiled, rather than being accomplished through declared variables. A kernel is piece of code to be run, similar to the main function in C++. It makes use of the grid-like nature of the GPU which, as described above, has a large number of parallel compute units. In OpenCL the GPU can be thought of as a grid with up to 3 dimensions. Instances of the kernel run in work-items which exist in local and global work group(s). An instance of a kernel knows where it is (and consequently what to do) using functions:

* get\_global\_size(index) – this gets the size of the work group in the dimension specified by index, which equals either 0,1 or 2.
* get\_global\_id(index) – this gets the global id of the work-item. This is used to get the coordinates of the work-item on the grid. This is essential for a work-item to know what to do
* get\_local\_size(index) – this gets the size of the local work group in the dimension specified by index, which equals either 0,1 or 2.
* get\_local\_id(index) – this gets the id of the item within a local work group.

This is illustrated in Figure 3. [[11]](#footnote-11)

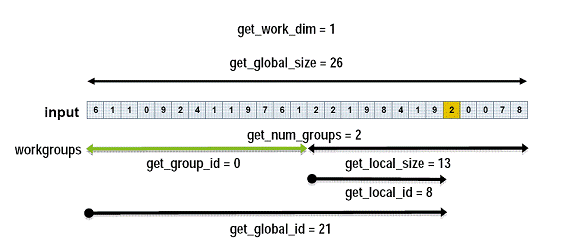


Figure 3

The best way to describe the parallelism within OpenCL on a GPU is with an example. If one were to add together two column vectors, denoted here by and one would have to iterate through the elements of each of the vectors adding them together one at a time to produce the result The code for this in serial would be as follows:

for(int i=0; i<10; i++)

{

C[i] = A[i] + B[i];

}

In OpenCL each of the addition operations would be assigned to a single work-item (assuming the hardware device was initialised with a sufficiently large grid), thus allowing all of the addition operations to occur simultaneously. The OpenCL code for this would be as follows:

\_\_kernel void vector\_add(\_\_global const float\* A, \_\_global const float\* B, \_\_global float\* C, const int num)

{

//num is used to indicate the length of the vector

//Gets the id of the current work item

const int id = get\_global\_id(0);

//Each work-item checks whether its id is in the range of the vector

//If it is, it performs the addition

if (id < num)

C[id] = A[id] + B[id];

}[[12]](#footnote-12)

The use of the \_\_global keyword brings us to the topic of memory in the context of OpenCL. There are four types of memory available for use within the GPU: global (\_\_global), local (\_\_local), private (\_\_private), and constant (\_\_constant). Global memory is available to all work groups and tends to be the slowest type of memory available. Local memory is only available within a local workgroup. Private memory is only available within a single work-item. Finally, constant memory is used to store constant values which cannot be changed. In the GPU used for the project constant memory is a section of global memory that broadcasts its values, which cannot be changed. The first two types of memory (global and local) bring up the issue of memory synchronisation. This is a similar memory access issue to the multi-threading issue described for C#. Global and local memory must be in sync within its global or local work group. Work items within a group must not read/write to the same section of memory at the same time. In order to achieve synchronisation of memory OpenCL has the barrier keyword. This can be used to sync global or local memory as follows:

barrier(CLK\_GLOBAL\_MEM\_FENCE);

barrier(CLK\_LOCAL\_MEM\_FENCE);

When a kernel instance reaches the barrier keyword it will wait for all other kernels in the local (in the case of CLK\_LOCAL\_MEM\_FENCE) work group or all other kernels in the global work group (in the case of CLK\_GLOBAL\_MEM\_FENCE) to reach the keyword. This can introduce the problem of deadlock, for example if for one kernel instance the barrier was within a condition that was never fulfilled, but for another it was not, the second kernel would wait forever for the first.

## High Frequency Trading (HFT)

HFT is a relatively recent development in the world of finance. Although computers have been increasingly used in markets it is only within the last 10-15 years that they have been used for HFT, which is a form of trading that falls within the category of algorithmic trading. This is the automation of trading using algorithms running on computers. It must be noted that whilst HFT is a form of algorithmic trading it is not the only form. In fact, it is a specific subset of algorithmic trading which occurs at high speed. The advances in technology have made it possible for market participants to trade without the need to be on the trading floor of a stock exchange. The system of open outcry or telephone trading has been steadily replaced by electronic ordering. Despite this switch, the principal operation of markets is unchanged in that they operate on a first-come first-served basis. Thus, instead of racing to be the first on the phone or to shout across the floor, today’s market participants need to have the fastest and lowest latency connections in order to be first. Indeed stock exchanges, such as the London Stock Exchange (LSE), now have systems in place which enable trades to be completed in less than a millisecond.[[13]](#footnote-13) This shift to electronic trading has been necessary to enable algorithmic trading. However it is not the only reason that high frequency trading is possible.

In Europe the monopoly of stock exchanges was broken by the elimination of the ‘concentration rule’, which required firms to route their orders only to stock exchanges.[[14]](#footnote-14) In the US intermarket price priority for quotations was introduced. These two reforms greatly increased the competitive environment stock exchanges operated in which resulted in a change in how they charge their customers. The consequence of this was not only lower prices for traders, but also new pricing structures such as the maker-taker structure.[[15]](#footnote-15) This pricing model operates on the principle that passive orders on the order book help to create liquidity and thus these orders, when fulfilled, receive a fee rebate. At the same time active traders are takers of liquidity (as they take orders of the book) and are thus charged a positive fee. Exchanges reward the addition of more liquidity as it makes the exchange a more attractive proposition for prospective clients.[[16]](#footnote-16) These changes in price for market participants have made it more financially feasible to trade smaller volumes at a time and this has enabled high frequency trading to take place. According to a Deutsche Bank paper *“the average trade size is now only 200 shares, down from 1600 shares fifteen years ago; the average value of an order has fallen to USD 6400 from USD 19400 five years ago.”* [[17]](#footnote-17)

As HFT is relatively new it does not yet have a single agreed upon definition. However according to the Technical Committee of the International Organization of Securities Commissions (IOSCO), an association of entities around the world which are responsible for the regulation of securities and futures, there are a number of features and characteristics which can help to define exactly what HFT is:

* *“It involves the use of sophisticated technological tools for pursuing a number of different strategies, ranging from market making to arbitrage.*
* *It is a highly quantitative tool that employs algorithms along the whole investment chain: analysis of market data, deployment of appropriate trading strategies, minimisation of trading costs and execution of trades.*
* *It is characterized by a high daily portfolio turnover and order to trade ratio (i.e. a large number of orders are cancelled in comparison to trades executed).*
* *It usually involves flat or near flat positions at the end of the trading day, meaning that little or no risk is carried overnight, with obvious savings on the cost of capital associated with margined positions. Positions are often held for as little as seconds or even fractions of a second.*
* *It is mostly employed by proprietary trading firms or desks.*
* *It is latency sensitive. The implementation and execution of successful HFT strategies depend crucially on the ability to be faster than competitors and to take advantage of services such as direct electronic access and co-location.”[[18]](#footnote-18)*

There are three main strategies which high frequency trading is used for: liquidity provision, statistical trading and liquidity detection.[[19]](#footnote-19) Strategies that provide liquidity are often based around achieving profitability due to the maker-taker pricing model now present in markets. By increasing the liquidity of the market the algorithm receives rebates, which allows the algorithm to be able to tolerate small trading losses. This strategy is most common in high volume and low volatility stocks.[[20]](#footnote-20) Alternatively the algorithm can make a profit due to the spread between the bid and the offer prices. In addition, a notable difference between a market making algorithm and a traditional market maker is that the algorithm is not formally required to make markets. Statistical strategies are usually focused on achieving arbitrage; this is where there is a small price differential in the market which has not yet corrected itself. Algorithms will detect when this is the case and exploit it to make a profit. Arbitrage is usually conducted across multiple exchanges. Finally liquidity detection strategies focus on price momentum – for example a piece of news may cause a rapid decrease in price causing the algorithm to attempt to sell before the price goes down. Although it is difficult to determine the precise definition of HFT it is possible to know that it is on the rise. For example in 2005 HFT accounted for approximately 21% of US equity trading, but by 2010 this had risen to 56% by 2010.[[21]](#footnote-21)

The impacts of HFT on markets are not yet fully understood due to its relatively new nature. In general HFT is believed to have no consistent negative effect on liquidity, with the exception of the Flash Crash on 6th May 2010. A study on the Flash Crash concluded that HFT had exacerbated the downwards move of prices.[[22]](#footnote-22) Apart from this it is believed that there are benefits to HFT, namely that it helps to reduce volatility as it adds liquidity to both sides of the market. However there is insufficient evidence to state this for certain. Other potential benefits of HFT include narrower bid-offer spreads which results in cheaper trading costs whilst having no impact on long-term investors. Some argue that there are significant downsides to high frequency trading. For instance the investment required to establish the infrastructure necessary to conduct high frequency trading is large, and this may crowd out smaller institutions from the market place.[[23]](#footnote-23)

# Objectives

Due to its recent development the impacts of high frequency trading on a market are not yet fully understood although there are some effects that it is thought to have (see 2.3 High Frequency Trading). Therefore this project aims to simulate high frequency trading in a market in order to examine how HFT affects the market. In order to do this as efficiently and effectively as possible the simulations will be run a GPU due to its highly parallel nature (see 2.1 GPUs). Thus the central objective of this project is to simulate high frequency trading on a GPU.

This can be broken down into three main stages, each of which can be further broken down into separate goals. For each of these three objectives it is indicated whether they are critical to the completion of the project as well as the relevant risks that arise. Additionally there are also stretch objectives which would further enhance the project, but which are not necessary for the completion of the main objective. The final part of this section shows the dependencies between objectives diagrammatically as well as indicating progress on each objective.

## Stage I - Develop A Market Model Where Trading Algorithms Do NOT Affect Prices

The first main objective to be completed is to develop a market model where stocks prices are generated using Geometric Brownian Motion (see **4.1.2 Simple OpenCL and Stock Price Generation** for explanation) and are observed by simple trading algorithms which indicate if they would buy/sell/hold using an output flag. In this model the stock prices are not affected by trading algorithms decisions; there are no bid-ask spreads and volumes are not taken into account. There are several goals built into this objective, the implementation of which will be described in more detail in **4 Implementation**. This objective offers a stepping stone to realising the objective described in **3.2 Develop a Market Model Where Trading Algorithms Do Affect Prices**. It is not critical to the market model that will be simulated by the project’s completion, but does assist with increasing familiarity with OpenCL.

### Gain a Good Understanding Of OpenCL

In order to implement anything to run on the GPU, OpenCL must be understood. The essential elements of OpenCL that must be understood are global and local work-groups and work-items; memory structure; synchronisation across work-groups and the C++ API. All other OpenCL related objectives in this project are dependent on this objective.

### Simple OpenCL Logic and Stock Price Generation

This objective is to implement the following logic in OpenCL and combine them all into a single algorithm:

* Uniformly Distributed Random Number Generation
* Normally Distributed Random Number Generation
* Matrix Multiplication
* Stock Price Generation using Geometric Brownian Motion.

It must also be possible to collect performance data about the finished algorithm such as how long it takes to complete.

### Simple Trading Algorithm Implementation

This objective is to implement a number of very simple trading algorithms that will output simple order data based on the algorithm’s observations of price. It will output a number indicating whether the algorithm is buying, selling, or holding. This is dependent on 3.1.2 Simple OpenCL Logic and Stock Price Generation.

### Analyse Stocks

In order to generate stock prices using Geometric Brownian Motion real stocks will be analysed to determine suitable values for expected returns and the variance of returns. Part of 3.1.2 Simple OpenCL Logic and Stock Price Generation is dependent on this.

### Implement Software Matrix Multiplication

Matrix Multiplication in OpenCL will involve pointers to arrays. This functionality must be duplicated to run on software and must therefore be implemented in C++.

### Implement LU Decomposition

LU Decomposition (see 4.2.1 Implement LU Decomposition (3.1.6)) must be implemented so that randomly generated stocks can have correlation with one another. This will be implemented in C++ so that matrices passed to the GPU are the correct ones for introducing correlation.

## Stage II - Develop a Market Model Where Trading Algorithms Do Affect Prices

The second main objective to be completed is to develop a market model in which stock prices are affected by the trading algorithms. This model will be based around an order book and will consequently be a much better approximation to a real market. The market will include features that were not present in 3.1 Develop a Market Model Where Trading Algorithms Do NOT Affect Prices including volumes; bid-ask spreads; the principle of first-come first-served; and different order types. Trading algorithms will also be more complex and will be designed to pursue realistic strategies. This objective is vital for completing the project as it is to create the most realistic and efficient market model. It can be broken down into several separate goals, the implementation of which is described in more detail in **4 Implementation**.

### Implement an Order Book Based Model in C++

This objective is to implement to market model based around an order-book in C++. Whilst not strictly necessary for the final goal of the project this will assist in implementing the model in OpenCL as well as providing a method to verify the model that will be implemented in OpenCL. This is dependent on 3.1.2 Simple OpenCL Logic and Stock Price Generation.

### Implement More Complex Trading Algorithms

Related to 3.2.1 Implement an Order Book Based Model In C++ in creating the more realistic market model, this objective will involve the implementation of trading algorithms, first in C++ then in OpenCL. These algorithms will be much more complex than those created for 3.1.3 Simple Trading Algorithm Implementation and will be based around strategies described in 2.3 High Frequency Trading.

### Implement an Order Book Based Model in OpenCL

Once 3.2.1 Implement an Order Book Based Model in C++ and 3.2.2 Implement More Complex Trading Algorithms have been completed the model will be translated into OpenCL so that it will run on the GPU.

### Scale Up and Optimise the Model

Finally once the model has been implemented in OpenCL as in 3.2.3 Implement an Order Book Based Model in C++ it must be scaled up to include a larger number of stocks and a larger number of trading algorithms. It must also be optimised to run as fast as possible – ideally close to real-time.

## Stage III - Analysis and Evaluation

This objective is completely dependent on 3.2 Develop a Market Model Where Trading Algorithms Do Affect Prices being completed. This will involve the evaluation of the model implemented as well as analysis of the results generated by simulating the market a number of times. The analysis and evaluation of the project are described more extensively in **5 Evaluation.**

## Stretch Objectives

These objectives are ones which would enhance the project beyond what is absolutely necessary to complete it. They are completely dependent on all other preceding objectives.

### Implement Singular Value Decomposition (SVD)

An alternative to using LU decomposition for determining the matrix used to introduce correlation to stocks is singular value decomposition.

### Millisecond Trading

The model could be scaled to operate on a milliseconds basis, where the time interval between samples represents only a millisecond. This is more applicable to the model described in 3.1 Develop a Market Model Where Trading Algorithms Do NOT Affect Prices**.**

### Multi-GPU

The model could be altered so that it ran on multiple GPUs at the same time. This would allow the model to be scaled up more.

### Full Size Market

The model could include a similar number of stocks to the number of major stock exchanges such as the London Stock Exchange (LSE) or the New York Stock Exchange (NYSE), which in 2002 had 2783 stocks listed.[[24]](#footnote-24)

## Objective Dependencies

Some objectives are dependent on each other. Figure 4 indicates which objectives have dependencies and whether the objectives have been completed or not.

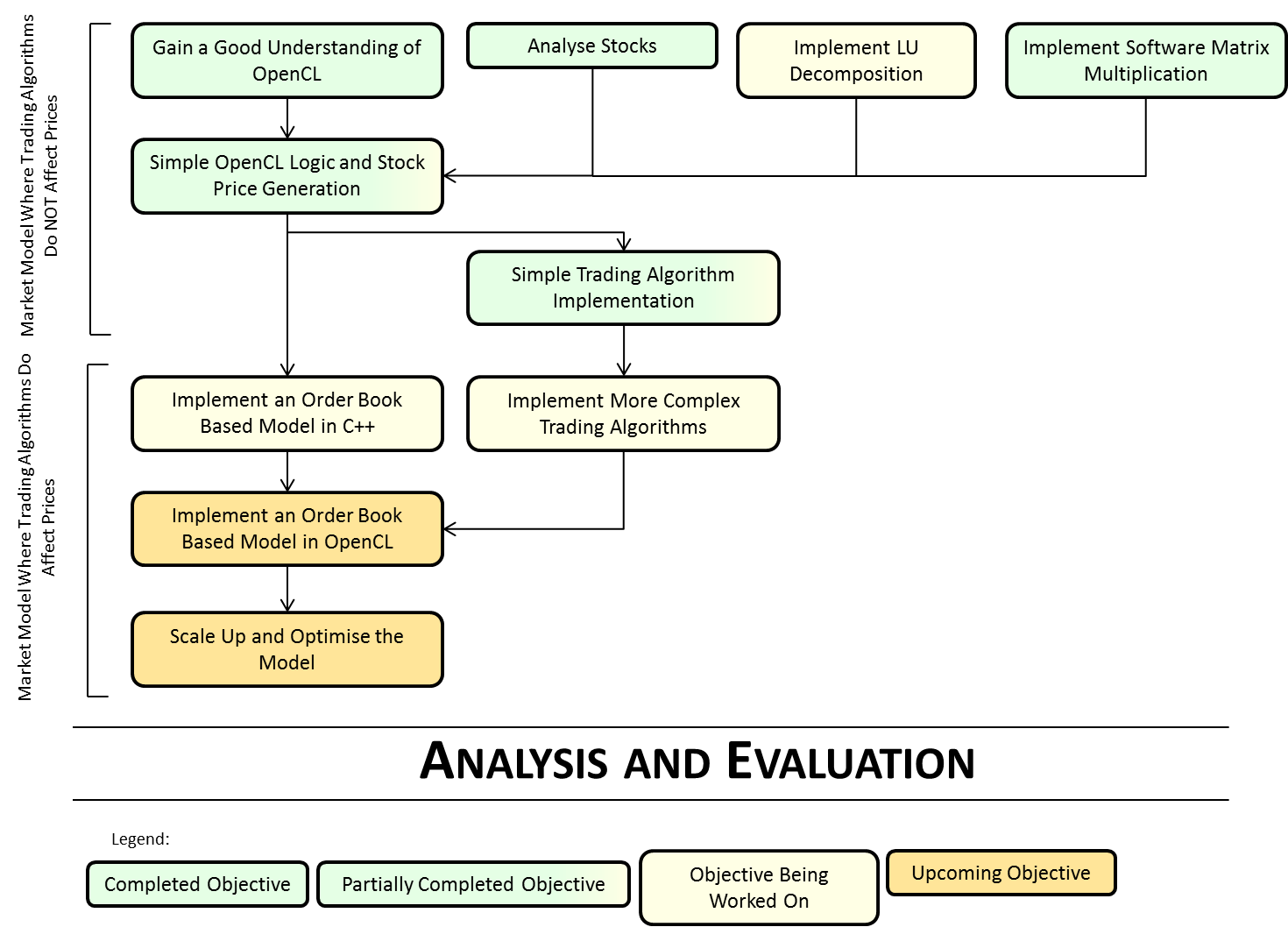


Figure 4

# Implementation

Project implementation has already been begun and will continue throughout the academic year. As described in **3 Objectives** there are numerous objectives which must be completed to achieve the overall goal of simulating high frequency trading. This section sets out an implementation strategy for each objective or describes how an objective has been accomplished. Additionally a timeline of when each objective is to be completed by is provided.

## Objectives Already Achieved

### Gain a good understanding of OpenCL (3.1.1)

I now have a good understanding of work-groups and work-items and understand the memory structure available within the GPU. In addition to this I understand how to synchronise memory within and between work-groups/items and have written a class representing the OpenCL device using the C++ API to keep the code as clear as possible and to maintain a good program structure. A more thorough description of how OpenCL and the GPU work can be found in 2.2 OpenCL. This objective is complete insofar as I now have a sufficiently good understanding of OpenCL to implement kernels for other objectives. However I will continue to become more familiar with OpenCL throughout the project as I gain more experience with it.

### Simple OpenCL Logic and Stock Price Generation (3.1.2)

I am using the mwc64x random number generator written by Dr Thomas which generates uniformly distributed random numbers across the range to (the range of an unsigned int)[[25]](#footnote-25). I have converted this to produce floats in the range to . This is accomplished by dividing each number from the mwc64x generated by .

Using the method just described to generate uniformly distributed random numbers I have implemented a normally distributed random number generator using the Box-Muller transform:

[[26]](#footnote-26)

Where and are normally distributed random numbers and and are uniformly distributed random numbers. Using this method and the mwc64x random number generator the logic to generate a specified number of samples using a specified number of random variables has been implemented. Each set of samples is treated as being generated at a specific point in time (so that the price of a stock can be generated over time). The samples are added to the bottom of a two dimensional array in which each row contains the samples at the time specified by the column.

Matrix multiplication has been implemented whereby an array of numbers is treated as a array and is multiplied by a matrix producing another vector. Thus each row from the aforementioned matrix of random numbers can be multiplied by a covariance matrix.

Geometric Brownian Motion has also been implemented to generate stock prices over time. Geometric Brownian Motion is defined as the solution to the stochastic differential equation (SDE):

Where, is geometric Brownian motion, is the mean (or average rate of return), is the standard deviation (of the stock), and is a normally distributed random variables (RV).

Solving this with Itô’s lemma gives the solution:

Thus it is lognormally distributed.[[27]](#footnote-27)

Finally these pieces of logic have been combined to implement logic that can generate the prices of a specified number of stocks over a given period of time. The structure of the logic is such that a work-item can generate a specified number of stocks. Within each work-item the following occurs:

1. Generate normally distributed random numbers.
2. Multiply the array of numbers by a covariance matrix.
3. Use the Random numbers to generate a array of stock prices. This is appended to the end of a matrix of all the samples generated.

There can be multiple work-items generating stocks at the same time – boosting the number of stocks being generated without having a big impact on the performance (work-items will execute at the same time). The fact that stocks are only related to other stocks within each work-item is to emulate the fact that some stocks in a market will be more strongly correlated with one another, for example banking stocks are usually more strongly correlated with each other than a banking stock would be with a technology company stock. The kernel currently takes as inputs a covariance matrix which each group of stocks in a work-item will be multiplied by; a array of expected returns where is the number of stocks; a array of the variance of the returns and a array of start prices. There are three outputs which are all matrices, where is the number of samples to be generated and is the number of stocks. There is an output matrix in which random numbers are stored before and after covariance matrix multiplication; a test output matrix in which random numbers are stored before covariance matrix multiplication – this is used to verify that the numbers have been generated correctly and that the matrix multiplication is correct; and a matrix of the stock prices which are calculated. The covariance matrix input can be set as an array of covariance matrices (one work each work-unit) or one constant matrix (each work-unit will use the same matrix). The work flow of the algorithm can be seen in Figure 5.

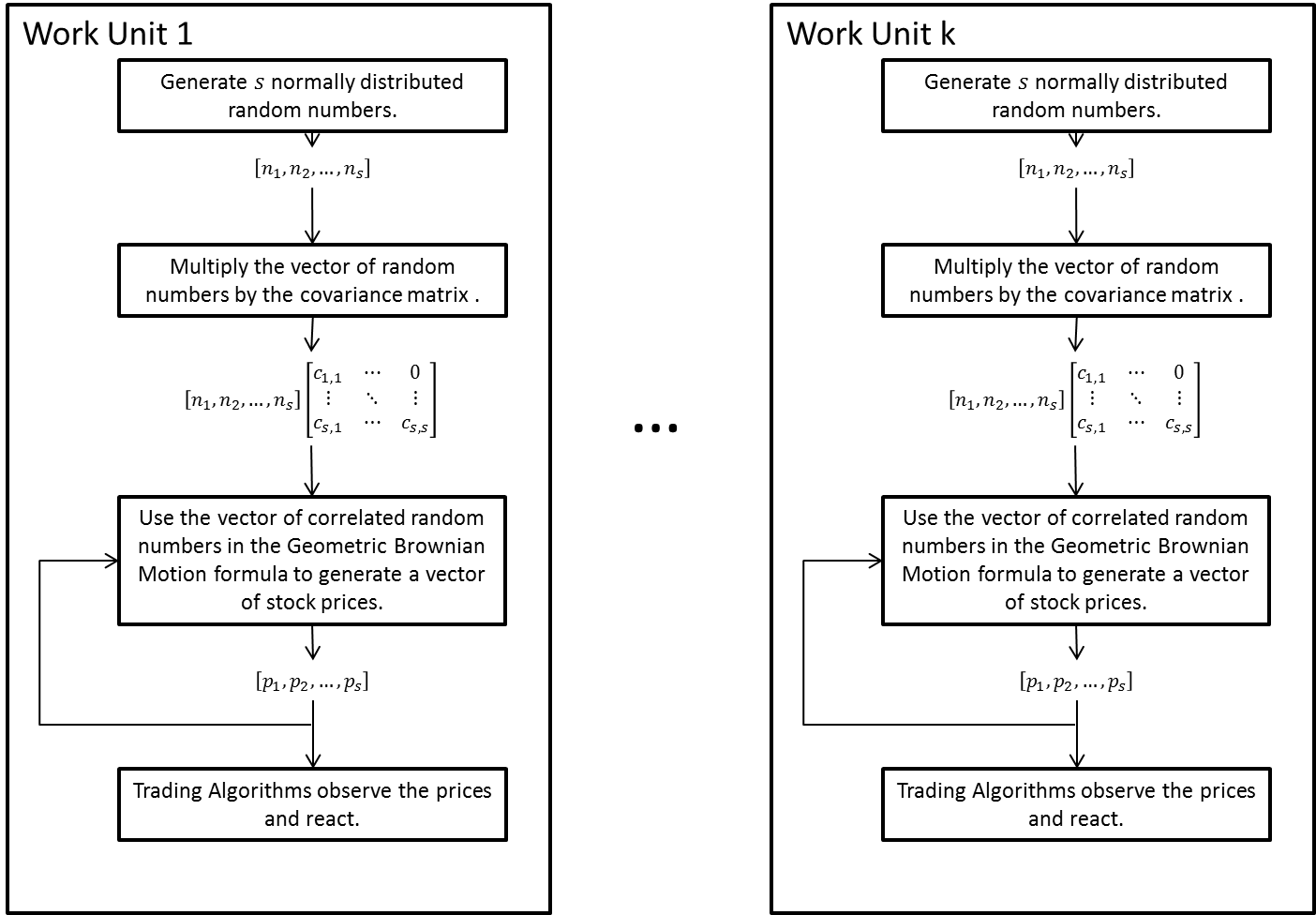


Figure 5

The time taken to complete this algorithm is output by the application.

### Simple Trading Algorithm Implementation (3.1.3)

This objective has been completed. Once stock prices have been generated as described in the previous objective they are looked at by a number of trading algorithms which then set a flag at the appropriate position in a matrix that is determined by the time sample number of the price being observed and the id of the trading algorithm. A ‘0’ is set when the trading algorithm does nothing; a ‘1’ is set when the trading algorithm wants to buy and a ‘2’ is set when it sells.

### Analyse Stocks (3.1.4)

A ‘Stock’ class has been implemented which can read in data from a .csv file containing data. This class calculates the values necessary for Geometric Brownian Motion. These values can be accessed via Get functions. Additionally the class has methods for simulating the stock over a given period of time using Geometric Brownian Motion. Further functionality will be added to this class over time when it is needed.

Daily stock price data can be downloaded for free from websites such as Yahoo Finance.[[28]](#footnote-28) The expected return of a stock is calculated using the formula:

Where is the expected return of the stock over the period of time, is the return of the stock at a given period of time, which is calculated as the close price (or adjusted close price) less the open price.[[29]](#footnote-29)

The standard deviation of returns can be calculated as follows using the typical formula for standard deviation:

[[30]](#footnote-30)

This gives the standard deviation of returns for the time interval. To get the standard deviation over the whole period one multiplies by:

This can be used to convert a standard deviation of returns for a stock from the standard deviation of returns for a day to month or to a second, however when converting to a longer period of time information is lost and it will not be possible to convert back and get exactly the same value. The opposite problem occurs when converting to a smaller time period – the converted value will not be entirely accurate as it will not have been determined with the data for the smaller time period.

### Implement Software Matrix Manipulation (3.1.5)

A ‘VectorMatrix’ class has been written which implements the storage of matrices using a pointer to a section of memory. A matrix is stored in a matrix using the indexing:

This class also implements methods for reading in matrices from text files, operators for manipulating multiple instances of the class (addition etc.) as well as access to rows, columns, value and the pointer to the block of memory. Further methods will be added to this class as necessary.

## Objectives Outstanding for 3.1Develop A Market Model Where Trading Algorithms Do NOT Affect Prices

### Implement LU Decomposition (3.1.6)

In order to generate the correct covariance matrix by which stocks will be multiplied, the correlation matrix must be decomposed. This will be accomplished using LU Decomposition which is the method for decomposing a matrix into the product of a lower triangular matrix and an upper triangular matrix where:

[[31]](#footnote-31)

In the model the covariance matrix will be the product of two other matrices, where:

This indicates that is a positive definite matrix, which means that Cholesky decomposition can be used in order to determine A. The Cholesky algorithm has two parts. Firstly the formula for determining the diagonal elements of the matrix:

Secondly the formula for determining the elements below the diagonal:

Where is the i by jth element of the matrix which is the lower triangular matrix being determined, and is the i by jth element of the matrix which is the correlation matrix. These two formulae will be used to calculate the matrix which will be passed into the OpenCL kernel as the matrix by which each set of random numbers will be multiplied.[[32]](#footnote-32)

Alternatively SVD may be used. This will alter the dimensions of the matrix and so could be used to boost the performance of the kernel.

## Objectives Outstanding for 3.2 Develop A Market Model Where Trading Algorithms Do Affect Prices

### Implement an order book based model in C++ (3.2.1)

Actual markets are based around an order book. The model used for simulating the trading must incorporate an order book. This will introduce the volume of the stocks being traded as well as allowing trading algorithms to place different types of orders and thus affect the price of the stock. This model will first be implemented in software to help facilitate testing and evaluation. In this model trading algorithms place orders of a specific type, volume, direction and price which are added to an order book. Orders of opposite directions (buy and sell) are matched up on a first-come first-served basis. This can be seen in Figure 6 - an example order book from the SIX Swiss Exchange.[[33]](#footnote-33)

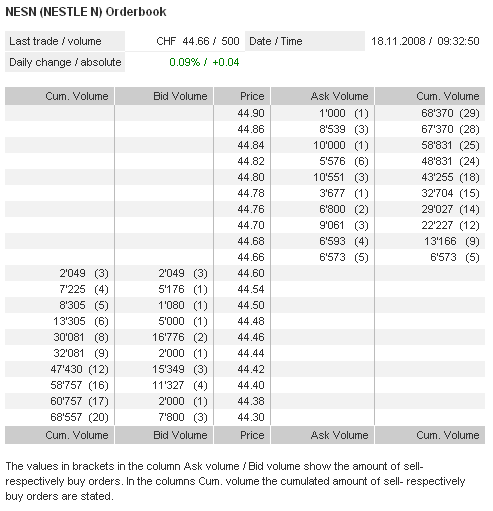


Figure 6

### Implement more Complex Trading Algorithms (3.2.2)

In reality trading algorithms are designed to achieve a certain objective, for instance, an algorithm may be designed to remain as close to flat as possible. More complex algorithms that are based on objectives that are held in real markets will be implemented. These will be written in software before being implemented in OpenCL to make it easier to test their ‘success’. The quantification of ‘success’ will be described in the Evaluation section. Further description of the different strategies employed by trading algorithms can be found in 2.3 High Frequency Trading.

### Implement an Order Book Based Model in OpenCL (3.2.3)

Once 3.2.1 Implement an Order Book Based Model In C++ has been competed the order book model will be converted to OpenCL. This will be a matter of translating the C++ code already written into OpenCL. The complex part of this implementation step will be to convert the model to take advantage of the parallelism of the GPU. Stocks and algorithms will be split up so that a specified number of stocks or algorithms will execute on a specified work-unit. It must then be possible to run multiple work-units simultaneously in a work-group. There can be one work-group containing units executing algorithms and another group providing the matching engine of the market.

### Scale Up and Optimise the Model (3.2.4)

The model must be scaled up to a larger number of stocks and/or to produce samples at a smaller time interval. The inputs and outputs of the model must also be altered to use less memory as reading and writing memory is the bottleneck in GPU operations. It will not be necessary to have all of the inputs and outputs as described in the 4.1.2 Simple OpenCL Logic and Stock Price Generation (3.1.2).

## Stretch Objectives (3.4)

### Implement Singular Value Decomposition (SVD) (3.4.1)

SVD is an alternative method to decomposing a matrix to LU Decomposition. It also provides the opportunity to have a non-square matrix which can be used to increase the speed of matrix multiplication (by decreasing the dimensions of the multiplication).

### Millisecond Trading (3.4.2)

This could be accomplished by setting the time interval to milliseconds in the model. It would only be possible assuming that the model has been optimised and with a limited number of stocks and/or trading algorithms.

### Multi-GPU (3.4.3)

This could be accomplished by executing multiple kernels on multiple GPUs using the OpenCL C++ API.

### Full Scale Market (3.4.4)

This could be accomplished simply by increasing the number of different stocks and the number of different trading algorithms. Similarly to 4.4.2 Millisecond Trading (3.4.2) this may require a larger time interval to be present in the model or it may not run close to or at real-time.

## Objectives Timeline

The timeline in which objectives will be completed can be seen in Figure 7.

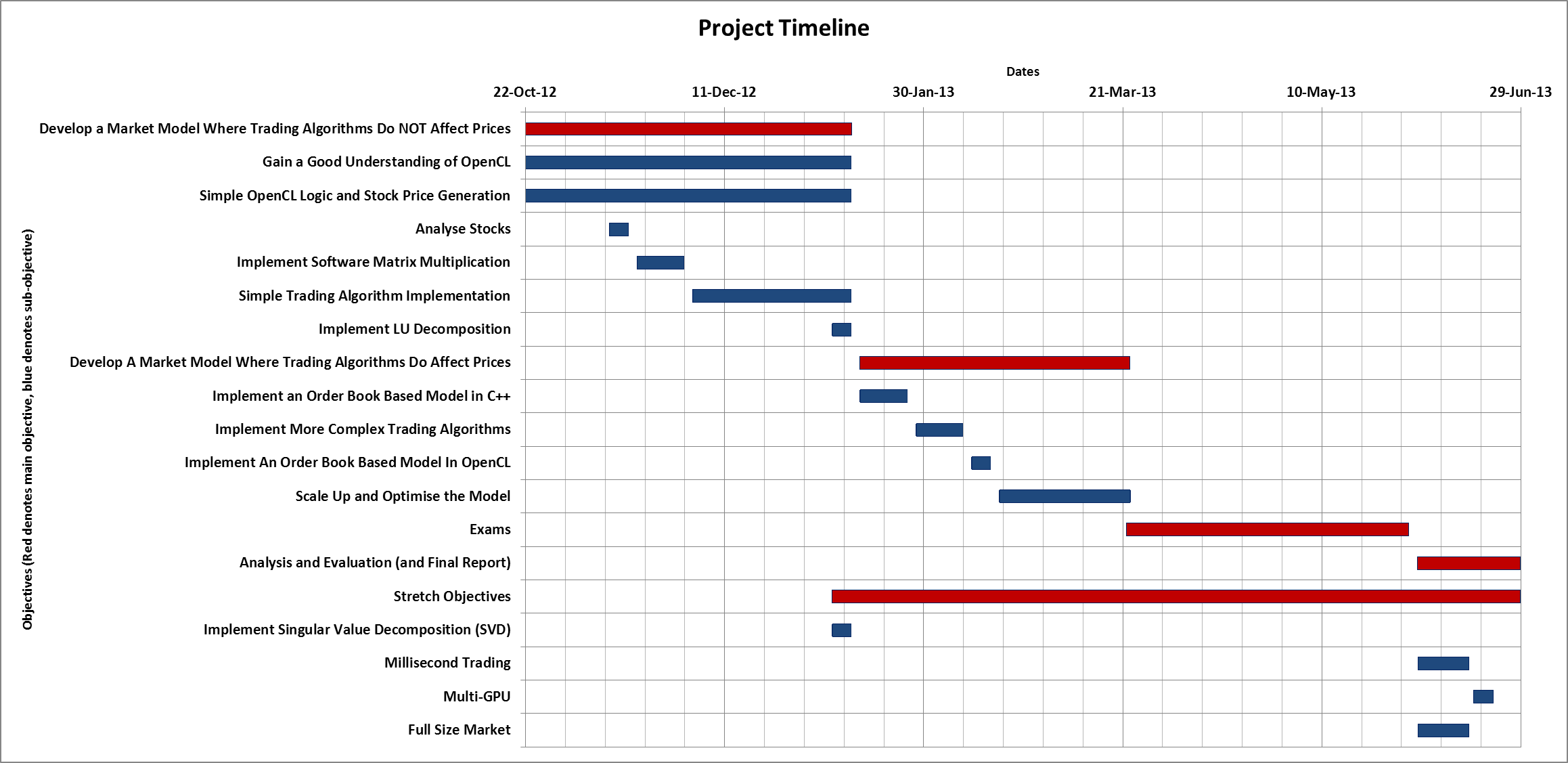


Figure 7

# Evaluation

The primary focus for much of the project will be the evaluation of the market model implemented in OpenCL and in software. This can be split into two separate parts: the initial random stock prices generation model and the order book based model. The features of these models which will be evaluated are the performance, the quality of the trading algorithms and the realism of the model. Additionally the performance of the initial model will be evaluated below.

### Performance

For the initial random price generation model the main measurements of performance are:

* The time in which the kernel executes (in seconds).
* The total number of stocks which is determined by the number of work-units and the number of stocks per unit.
* The total number of trading algorithms.
* The total number of samples generated.

The measurements of the performance of the order book model will be slightly different:

* How close to real time does the simulation run? This will be measured in terms of how long the simulation takes to simulate one second of real time. Ideally the simulation will run in at least real time (i.e. one second of simulation produces results equivalent to one second of real time). Thus it will be a ratio of simulation seconds to real seconds.
* How many trading algorithms there are.
* How fast is each trading algorithm – this will affect the first measurement of how close to real time the simulation is.

### Quality of Trading Algorithms

There will be a number of different types of trading algorithm, each of which will be employing a different strategy (see 2.3 High Frequency Trading). The overall measure of the quality of each trading algorithm will be its profitability; however it will also be assessed to see whether it fulfils the goals of its strategy. This will be tested first in software before implementing the algorithms in OpenCL.

### Realism of the Model

The realism of the model can be assessed in terms of how close the model is to replicating a real market. In the initial stock price generation model this will include the use of appropriate values for the expected return and the variance of returns for each stock. For both models the covariance matrix or matrices must be realistic. This will be assessed against the correlation between real stocks. In the order book model, the volumes of the stocks must also be realistic i.e. of the same order of magnitude of real stock volumes.

## Initial Model Evaluation

The initial model was assessed in terms of how long it took to execute the kernel in seconds in relation to the total number of stocks, the total number of stocks per work unit and the total number of samples.

The number of samples has a linear relationship with the amount of time taken to complete the kernel. This can be seen in Figure 8.

Figure 8

This is expected since increasing the number of samples will linearly increase the number of operations that have to be completed.

The time taken compared to the number of stocks per work unit was found to have the relationship seen in Figure 9.

Figure 9

The time taken has a squared relationship with the number of stocks per work unit. This is expected as the most costly part of the algorithm in terms of number of operations is the multiplication of the vector of normally distributed random numbers by the covariance matrix. This multiplication takes operations, thus increasing produces an order of magnitude increase in time taken.

The advantages of using the parallelism of the GPU to implement multiple work-units each generating multiple stocks is made apparent in Figure 10.

Figure 10

It can be seen that by doubling the number of work units that are processing the same number of stocks each; the total number of stocks can be doubled with only a very small increase in the time taken for the simulation to be completed.

# Conclusion

This interim report provides a look at progress towards the realisation of the Simulating High Frequency Trading on a GPU project to date. GPUs have developed into extremely powerful parallel platforms making them ideal for simulations which require a large number of tasks to be completed simultaneously, such as simulating high frequency trading in a stock market. The report goes on to examine the functionality of OpenCL in relation to the project with particular regard to its usefulness in implementing a model that will run on a GPU and that takes advantage of its parallelism. It gives essential context regarding the history and nature of high frequency trading. The objectives section sets out exactly which objectives must be completed in order to finish the project as well as setting out some stretch objectives which would further enhance the project. The implementation section which follows describes when each objective will be completed (or if it has been already) as well as setting out a method to implement the objective. It also describes the implementation of the objectives which have already been realised including: 3.1.1 Gain A Good Understanding of OpenCL, 3.1.2 Simple OpenCL Logic and Stock Price Generation, 3.1.3 Simple Trading Algorithm Implementation, 3.1.4 Analyse Stocks, and 3.1.5 Implement Software Matrix Multiplication. Across the outstanding objectives there just remains 3.1.6 Implement LU Decomposition, to complete 3.1 Develop a Market Model Where Trading Algorithms Do NOT Affect Prices, and then work can begin on 3.2 Develop A Market Model Where Trading Algorithms Do Affect Prices and all of its sub-objectives. Additionally the report sets out the strategies that will be used to evaluate the project in areas such as Model Performance (5.1), Model Quality (5.2) and Model Realism (5.3). It also assesses the features of the implementation work already completed and confirms the suitability of using a GPU for the task.

1. Mohamed Zahran, ‘Lecture 2: History of GPU Computing’, New York University, url: http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture2.pdf, accessed 21/12/2012, slide 4. [↑](#footnote-ref-1)
2. Stanislav Stanković, Dušan Gajić, Radomir S. Stanković, ‘GPU Architecture and the Programming Environment’, in: Jaakko Astola, Michitaka Kameyama, Martin Lukac and Radomir S. Stanković (eds.), *GPU Computing with Applications in Digital Logic*, (Tampere: 2012), url: http://ticsp.cs.tut.fi/reports/ticsp-report-62-November-19-2012.pdf#page=15, accessed: 21/12/2012, page 10. [↑](#footnote-ref-2)
3. John Owens, Presentation: ‘GPU Architecture Overview’, University of California Davis, url: http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf, accessed: 21/12/12, slide 8. [↑](#footnote-ref-3)
4. ‘Programming Guide for Direct3D 11’, Graphics Pipeline, Windows Dev Center – Desktop, Microsoft, url: http://msdn.microsoft.com/en-gb/library/windows/desktop/ff476882(v=vs.85).aspx, accessed: 21/12/12. [↑](#footnote-ref-4)
5. Diagram appearing in: ‘Definition of Graphics Pipeline’, PCMAG.com Encyclopedia, url: http://www.pcmag.com/encyclopedia\_term/0,1237,t=graphics+pipeline&i=43933,00.asp, accessed: 21/12/12. [↑](#footnote-ref-5)
6. Lars Nyland, Mark Harris, Jan Prins, ‘Fast N-Body Simulation with CUDA’, in: *GPU Gems* 3, Addison-Wesley Professional: 2007, url: ftp://sleepgate.dyndns.org/library/CUDA/nbody\_gems3\_ch31.pdf, accessed: 21/12/2012, page 678. [↑](#footnote-ref-6)
7. ‘nVidia GeForce 8800 GTX’, GPUReview.com, url: http://www.gpureview.com/GeForce-8800-GTX-card-474.html, accessed: 21/12/12. [↑](#footnote-ref-7)
8. ‘OpenGL Shading Language’, OpenGl.org, url: http://www.opengl.org/documentation/glsl/, accessed: 21/12/12. [↑](#footnote-ref-8)
9. ‘Introducing CUDA 5’, NVIDIA.com, url: http://www.nvidia.com/object/cuda\_home\_new.html, accessed: 21/12/12. [↑](#footnote-ref-9)
10. ‘AMD Radeon HD 6970 Graphics’, AMD.com, url: http://www.amd.com/uk/products/desktop/graphics/amd-radeon-hd-6000/hd-6970/Pages/amd-radeon-hd-6970-overview.aspx#2, accessed: 21/12/12. [↑](#footnote-ref-10)
11. ‘Part 3: Work-Groups and Synchronization’, CodeProject.com, url: http://www.codeproject.com/Articles/143395/Part-3-Work-Groups-and-Synchronization, accessed: 21/10/2012, and, Tim Mattson, Ian Buck, Michael Houston, Ben Gaster, Presentation: ‘OpenCL: A Standard Platform for programming Heterogeneous parallel computers’, url: http://software.intel.com/sites/default/files/m/e/1/8/4/5/24571-OpenCL-tut-sc09.pdf, accessed: 21/12/12, slide 15. [↑](#footnote-ref-11)
12. ‘OpenCL Tutorials 1 – Quickstart’, CodePlex.com, url: http://opencl.codeplex.com/wikipage?title=OpenCL%20Tutorials%20-%201, accessed: 21/10/2012. [↑](#footnote-ref-12)
13. ‘Colt to offer sub-millisecond connectivity to London Stock Exchange’, 10January 2011, ComputerWeekly.com, url: http://www.computerweekly.com/news/1280094775/Colt-to-offer-sub-millisecond-connectivity-to-London-Stock-Exchange, accessed: 01/11/12. [↑](#footnote-ref-13)
14. ‘MiFID the Markets in Financial Instruments Directive’, Price Waterhouse Coopers, url: http://www.pwc.lu/en\_LU/lu/mifid/docs/pwc-mifid230506.pdf, accessed: 20/12/2012, page 2. [↑](#footnote-ref-14)
15. Michael Chlistalla, ‘High-frequency trading: Better than its reputation?’,7 February 2011, Deutsche Bank Research, url: http://www.dbresearch.de/PROD/DBR\_INTERNET\_EN-PROD/PROD0000000000269468.PDF, accessed: 20/12/2012, page 2. [↑](#footnote-ref-15)
16. Ryan Riordan and Andreas Park, ‘Maker/taker Pricing and High Frequency Trading’, 14 February 2012, Economic Impact Assessment EIA12, Foresight, Government Office for Science, url: http://www.bis.gov.uk/assets/foresight/docs/computer-trading/12-1073-eia12-maker-taker-pricing-and-high-frequency-trading.pdf, accessed: 20/12/2012, page 3. [↑](#footnote-ref-16)
17. Michael Chlistalla, ‘High-frequency trading: Better than its reputation?’,7 February 2011, Deutsche Bank Research, url: http://www.dbresearch.de/PROD/DBR\_INTERNET\_EN-PROD/PROD0000000000269468.PDF, accessed: 20/12/2012, page 2. [↑](#footnote-ref-17)
18. ‘Regulatory Issues Raised by the Impact of Technological Changes on Market Integrity and Efficiency, Consultation Report’, OICV-IOSCO, Technical Committee of the International Organization of Securities Commissions, CR02/11, July 2011, url: https://docs.google.com/viewer?url=http%3A%2F%2Fwww.iosco.org%2Flibrary%2Fpubdocs%2Fpdf%2FIOSCOPD354.pdf, accessed 01/11/12, page 21. [↑](#footnote-ref-18)
19. Michael Chlistalla, ‘High-frequency trading: Better than its reputation?’,7 February 2011, Deutsche Bank Research, url: http://www.dbresearch.de/PROD/DBR\_INTERNET\_EN-PROD/PROD0000000000269468.PDF, accessed: 20/12/2012, page 3. [↑](#footnote-ref-19)
20. Diagram: ‘The World of High Frequency Trading: The Six Primary Strategies’, TechniciansCorner, fibozachi.com, url: https://www.fibozachi.com/images/stories/TechniciansCorner/WorldofHFT/t3live%20the%20world%20of%20hft.png, accessed: 30/10/12. [↑](#footnote-ref-20)
21. ‘Regulatory Issues Raised by the Impact of Technological Changes on Market Integrity and Efficiency, Consultation Report’, OICV-IOSCO, Technical Committee of the International Organization of Securities Commissions, CR02/11, July 2011, url: https://docs.google.com/viewer?url=http%3A%2F%2Fwww.iosco.org%2Flibrary%2Fpubdocs%2Fpdf%2FIOSCOPD354.pdf, accessed 01/11/12, page 22. [↑](#footnote-ref-21)
22. Ibid, page 26, footnote 36, and, Andrei Kirilenko, Albert S. Kyle, Mehrdad Samadi and Tugkan Tuzun, ‘The Flash Crash: The Impact of High Frequency Trading on an Electronic Market’, 12 January 2011, Social Science Research Network, url: http://papers.ssrn.com/sol3/papers.cfm?abstract\_id=1686004, accessed: 20/12/12. [↑](#footnote-ref-22)
23. Bruno Biais, Thierry Foucault and Sophie Moinas, ‘Equilibrium High Frequency Trading’, 14 October 2011, Social Science Research Network, url: http://papers.ssrn.com/sol3/papers.cfm?abstract\_id=2024360, accessed: 2/11/12. [↑](#footnote-ref-23)
24. ‘Number of NYSE listed stocks (1924-2002), New York Stock Exchange Data, url: http://www.nyxdata.com/nysedata/asp/factbook/viewer\_edition.asp?mode=table&key=76&category=4, accessed: 21/12/12. [↑](#footnote-ref-24)
25. David B. Thomas, ‘The MWC64X Random Number Generator’, Circuits and Systems Research Group, Imperial College London, url: http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html, accessed: 23/10/12. [↑](#footnote-ref-25)
26. Eric W. Weisstein, ‘Box-Muller Transformation’ in MathWorld – A Wolfram Web Resource, url: http://mathworld.wolfram.com/Box-MullerTransformation.html, accessed: 3/11/12. [↑](#footnote-ref-26)
27. Claudio Pacati, ‘Brownian Motion and Geometric Brownian Motion’, Dipartimento di Economia Politica E Statistica (Department of Economics and Statistics), url: http://www.econ-pol.unisi.it/fm10/BMandGBMdoc.pdf, accessed: 5/11/12, and, Karl Sigman, ‘Geometric Brownian Motion’, 2006, Columbia University, url: http://www.columbia.edu/~ks20/FE-Notes/4700-07-Notes-GBM.pdf, accessed: 5/11/12. [↑](#footnote-ref-27)
28. ‘Yahoo Finance’, Yahoo, url: http://uk.finance.yahoo.com/, accessed: 10/12/12. [↑](#footnote-ref-28)
29. ‘Risk and Return: Expected Return’, PreMBA Finance, Columbia University, url: http://ci.columbia.edu/ci/premba\_test/c0332/s6/s6\_3.html, accessed: 20/12/12. [↑](#footnote-ref-29)
30. ‘Standard Deviation of Returns’, FundManagerSoftware.com, url: http://fundmanagersoftware.com/help/def\_standard\_deviation\_returns.html, accessed: 15/11/12. [↑](#footnote-ref-30)
31. Eric W. Weisstein, ‘LU Decomposition’, in MathWorld – A Wolfram Web Resource, url: http://mathworld.wolfram.com/LUDecomposition.html, accessed: 19/12/12. [↑](#footnote-ref-31)
32. Thijs van den Berg, ‘Generating Correlated Random Numbers’, sitmo.com, url: http://www.sitmo.com/article/generating-correlated-random-numbers/, accessed: 19/12/12, and, ‘Cholesky Decomposition’, rosettacode.org, url: http://rosettacode.org/wiki/Cholesky\_decomposition#C, accessed: 19/12/12. [↑](#footnote-ref-32)
33. Diagram: ‘NESN (NESTLE N) Orderbook’ in ‘Trading at the SIX Swiss Exchange’, SIX Swiss Exchange, url: http://www.six-swiss-exchange.com/knowhow/exchange/trading\_en.html, accessed: 04/01/2013. [↑](#footnote-ref-33)