Simulating High Frequency Trading Using A GPU

Christopher Latawski

**Abstract:**

**Acknowledgements:**

**Table of Contents:**

Contents

[1 Introduction 3](#_Toc357702593)

[2 Background 3](#_Toc357702594)

[2.1 Graphics Processing Units (GPUs) 3](#_Toc357702595)

[2.2 OpenCL 6](#_Toc357702596)

[2.3 High Frequency Trading Background 8](#_Toc357702597)

[3 Design 10](#_Toc357702598)

[3.1 Order Book 11](#_Toc357702599)

[3.2 Trader Manager 12](#_Toc357702600)

[3.3 Traders 13](#_Toc357702601)

[3.3.1 Random Trader (RT) 13](#_Toc357702602)

[3.3.2 Large Random Trader (LRT) 14](#_Toc357702603)

[3.3.3 Position Trader (PT) 14](#_Toc357702604)

[3.3.4 Momentum Trader (MT) 14](#_Toc357702605)

[4 Implementation of the Model 14](#_Toc357702606)

[4.1 Command Line and Main Loop 15](#_Toc357702607)

[4.2 OrderBook and Classes 17](#_Toc357702608)

[4.2.1 Stock Class, PastPrice Structur 17](#_Toc357702609)

[4.2.2 Order and OrderType 18](#_Toc357702610)

[4.2.3 OrderRequest and OrderQueue 19](#_Toc357702611)

[4.2.4 Rules, IRule and RuleManager 20](#_Toc357702612)

[4.2.5 Trade 22](#_Toc357702613)

[4.2.6 OrderBook 22](#_Toc357702614)

[4.3 TraderManager and Classes 25](#_Toc357702615)

[4.3.1 Trader and TraderType 25](#_Toc357702616)

[4.3.2 Structures for OpenCL 28](#_Toc357702617)

[4.3.3 OpenCLController and OpenCLDevice 29](#_Toc357702618)

[4.3.4 Traders OpenCL 33](#_Toc357702619)

[4.3.5 TraderManager 38](#_Toc357702620)

[4.4 Helper Classes 40](#_Toc357702621)

[4.4.1 Seed 41](#_Toc357702622)

[4.4.2 Utils 41](#_Toc357702623)

[4.4.3 Logger 41](#_Toc357702624)

[4.4.4 TBBlog 42](#_Toc357702625)

[4.4.5 WallTimer 42](#_Toc357702626)

[4.4.6 Tests 42](#_Toc357702627)

[5 Analysis 42](#_Toc357702628)

[5.1 Verification of Motivation 42](#_Toc357702629)

[5.1.1 Varying Total Number of Traders 42](#_Toc357702630)

[5.1.2 Varying the Average Trader T 46](#_Toc357702631)

[5.1.3 Varying the Duration of the Simulation 50](#_Toc357702632)

[5.1.4 Varying the Number of Traders With the Same Trader T 50](#_Toc357702633)

[5.2 Verification of Determinism 50](#_Toc357702634)

[5.3 Model 1 53](#_Toc357702635)

[5.3.1 Performance Analysis 53](#_Toc357702636)

[5.3.2 Financial Analysis 70](#_Toc357702637)

[5.4 Model 2 94](#_Toc357702638)

[5.4.1 Performance Analysis 94](#_Toc357702639)

[5.4.2 Financial Analysis 94](#_Toc357702640)

[5.5 Conclusions of Analysis 94](#_Toc357702641)

[5.5.1 Performance 94](#_Toc357702642)

[5.5.2 Financial 94](#_Toc357702643)

[6 Future Work 94](#_Toc357702644)

[7 Conclusion 94](#_Toc357702645)

[8 Bibliography 94](#_Toc357702646)

[9 Appendix (Stored in a separate file currently) 94](#_Toc357702647)

# Introduction

# 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.

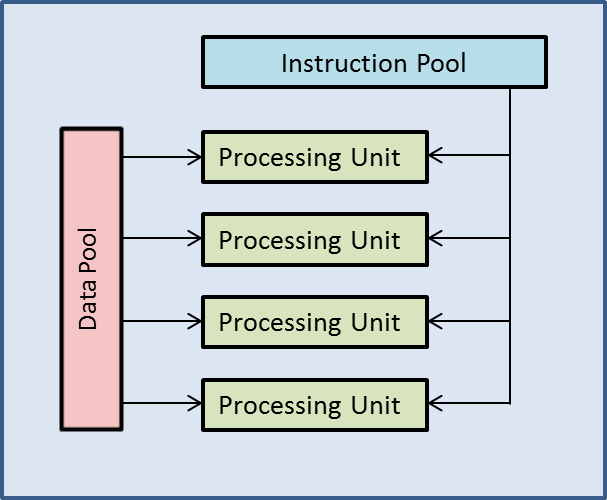


Figure 1

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)

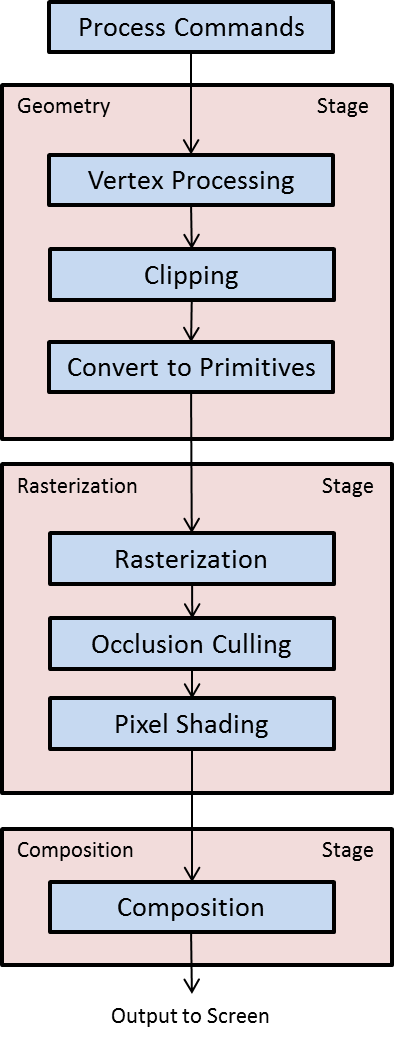


Figure X

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)

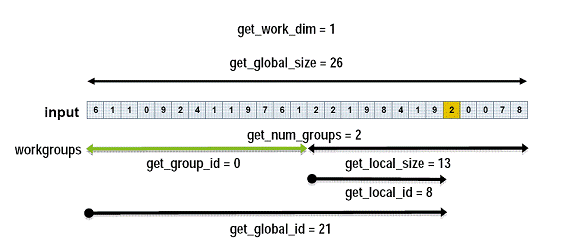


Figure 3 Remove or better explain this figure

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 Background

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)

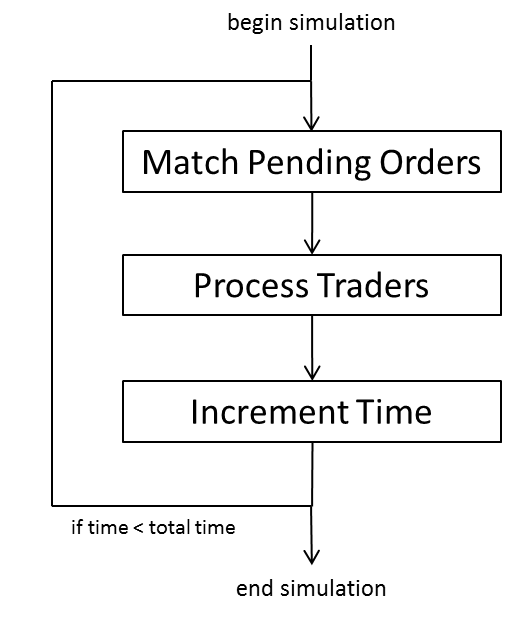
# Design

The model was designed with C++ and OpenCL in mind.

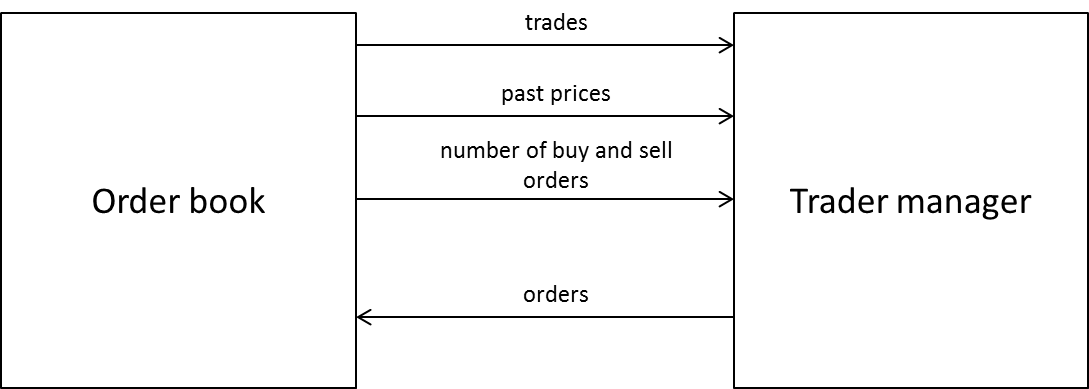
The overall model consisted of a main loop which runs over a set number of discrete time steps representing one millisecond of real time. Within each iteration of the loop the following was to occur:

1. Match pending orders in the order book to form trades (where possible).
2. Process the trading algorithms using the data available for the current time unit.
3. Increment the time.
4. If the time is equal to the end time then quit, otherwise return to 1.

This can be seen in Figure X, which is a flowchart showing the highest level process flow of the model.



The model was designed to consist of two main components: the order book, and the traders. These two main components would be contained and managed from two classes: one representing the order book and the other representing the traders – called a trader manager. The separation between these was to be maintained so that the functionality of the order book could remain unchanged whilst a number of different variations of trades could occur. These could include using different trading algorithms, executing the traders differently (e.g. OpenCL on a GPU, on a CPU, or a multi-threaded setup), as well as allowing the number of traders to be varied independently of the order book. The trader manager and the order book were to be the only classes that communicate with one another, relaying information such as past prices, the number of buy or sell orders, trades and orders. This can be seen in Figure X.



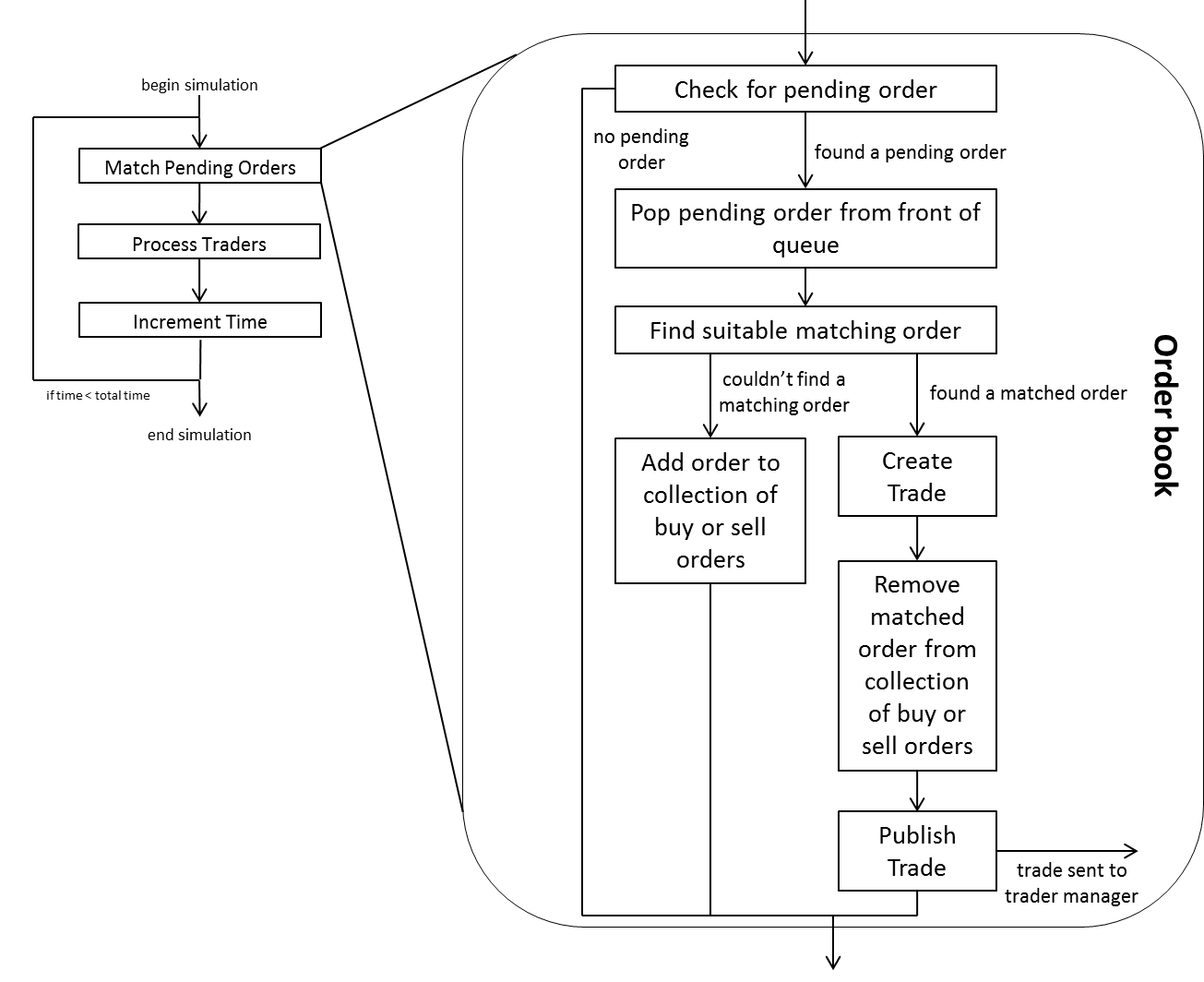
The order book would take care of Step 1. Of the main loop discussed above, and the trader manager would take care of Step 2. These will now be discussed further.

## Order Book

The order book was designed to include the following:

* A collection of buy orders. This was to keep track of all buy orders which have been placed, but not yet matched with a sell order.
* A collection of sell orders. This was to keep track of all sell orders which have been placed, but not yet matched with a buy order.
* A queue of pending orders, which can have orders added to the end of the queue and removed from the front. This was to keep track of any orders which have been placed, but have not been attempted to be matched yet. It is also necessary to maintain time-priority of orders, i.e. first-come-first-served.
* The ability to match an order with another order of opposite direction to produce a trade. This was to be called the Matching Engine. This was necessary so that trades could be created. It also had to ensure time-priority and price-priority (lowest sell price and highest buy price have highest priority).
* A collection of trades. This was to keep track of all trades that were created.
* The ability to publish a trade (sending it to the trader manager), as well as the ability to send price data and the number of buy or sell orders to the trader manager. This was vital to ensure that traders could know when they had executed a trade, as well as giving them access to data such as past prices, the use of which will be discussed subsequently in section 3.3.

All of these were necessary to produce the operation flow of the order book, which needed the ability to add/remove orders and match them together into a trade. The operation flow of the order book can be seen in Figure X, which is a flowchart shown in the context of the main loop of the model.

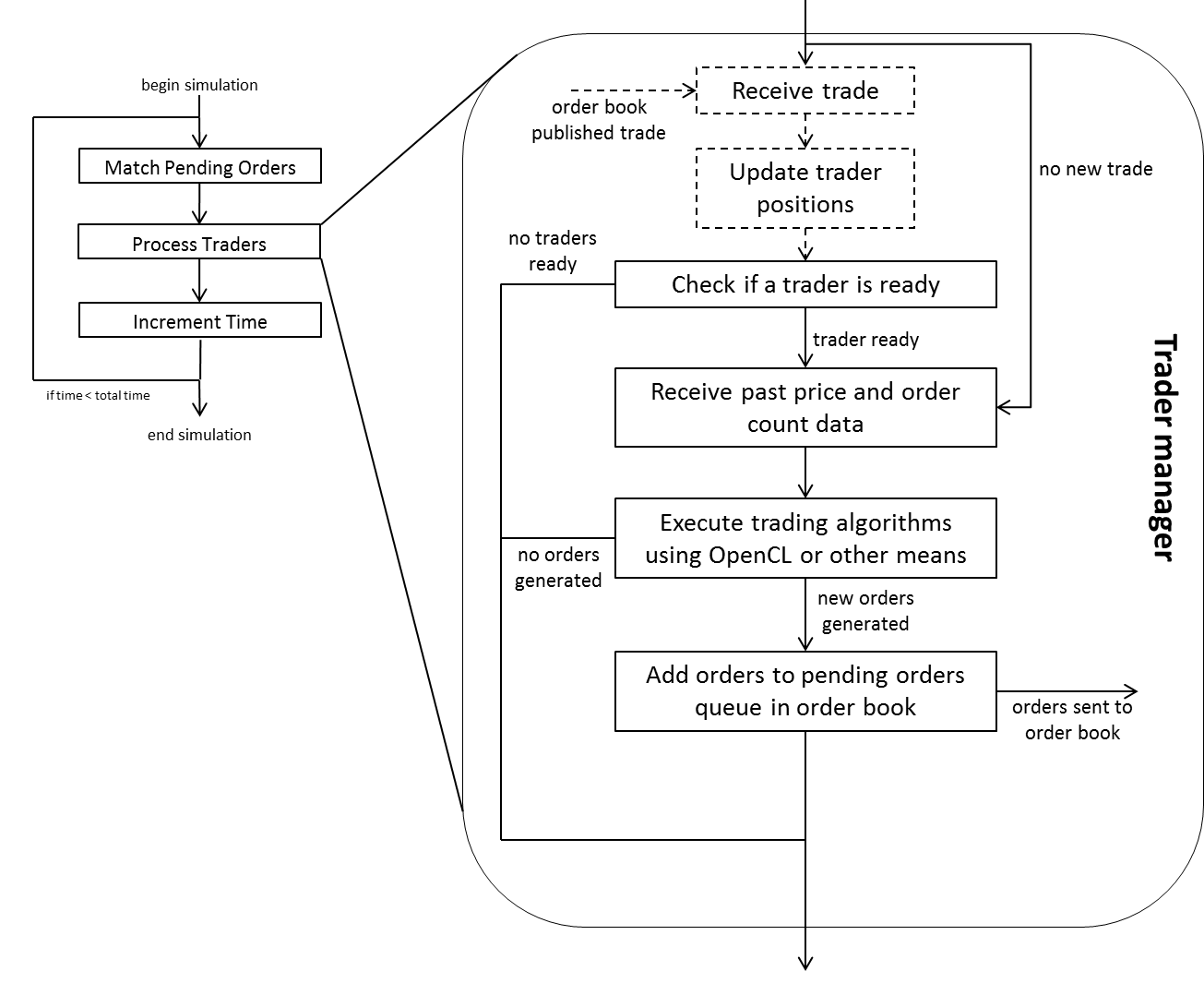


## Trader Manager

The trader manager was designed to include the following:

* A collection of each type of traders. The types of traders will be discussed in the next section (3.3). This was necessary so that the trader manager could maintain the ability to act as a central point from which all traders can be accessed or controlled.
* The ability to execute traders when they are due to run. This was necessary so that traders could have a time in simulation milliseconds which would determine how frequently they would place orders.
* An OpenCL object which could be used to execute trader algorithms written in OpenCL on a GPU (or a CPU). This was necessary to contain all of the OpenCL API calls necessary for executing OpenCL kernels.
* The ability to send orders to the order book. Self-explanatory.
* The ability to receive published trades from the order book and notify the corresponding traders that they have executed a trade.
* The ability to receive price data and other data from the order book and pass it to the traders that make use of it.

In each tick of the simulation the trader manager was designed to receive past price data and the number of buy or sell orders in the order book, and then execute each trader’s trading algorithm. Once these had been completed then any orders generated by the algorithms would be added to the order book. This can be seen in Figure X, which shows the operation of the trader manager in the context of the main loop.



## Traders

A total of four different types of traders were designed for the model. Two of which were designed to behave randomly to simulate some of the price movement of the market. The other two were designed to pursue a specific financial goal. The two random based traders were the Random Trader (RT) and Large Random Trader (LRT), and the two strategy-oriented traders were the Position Trader (PT) and the Momentum Trader (MT). Each type of trader would keep account of its current amount of cash and its current holdings, as well as the effects of its placed orders on its amount of cash and current holdings.

### Random Trader (RT)

The RT was designed to behave completely randomly. It will generate orders in which the following features were all random to some degree:

* Type. Orders would have a random chance of being a market or limit order.
* Direction. Orders would have a random chance of being buy or sell.
* Price. If the order was a limit order then the price would be randomly generated close to the last reported price from the order book.
* Volume. The quantity of the order would be randomly generated.

### Large Random Trader (LRT)

The LRT was designed to behave similarly to the RT (see section 3.3.1), however with some notable differences. First of all the volume of the orders would be at least an order of magnitude larger than that of those generated by RTs. Secondly the direction of the order would be dependent on the number of buy or sell orders outstanding in the order book. For example if there were considerably more buy orders than sell orders on the book, then the LRT would place a sell order.

This behaviour was designed to simulate the fact that sometimes in real markets large orders are placed which clear or nearly clear the opposite side of the order book. It was also designed with performance in mind; helping to keep the collections of buy or sell orders small would help the time taken to match orders as the matching engine would have to traverse the list of orders in the opposite direction to the current order, which depending on the data structure used could take up to O(n) operations.

### Position Trader (PT)

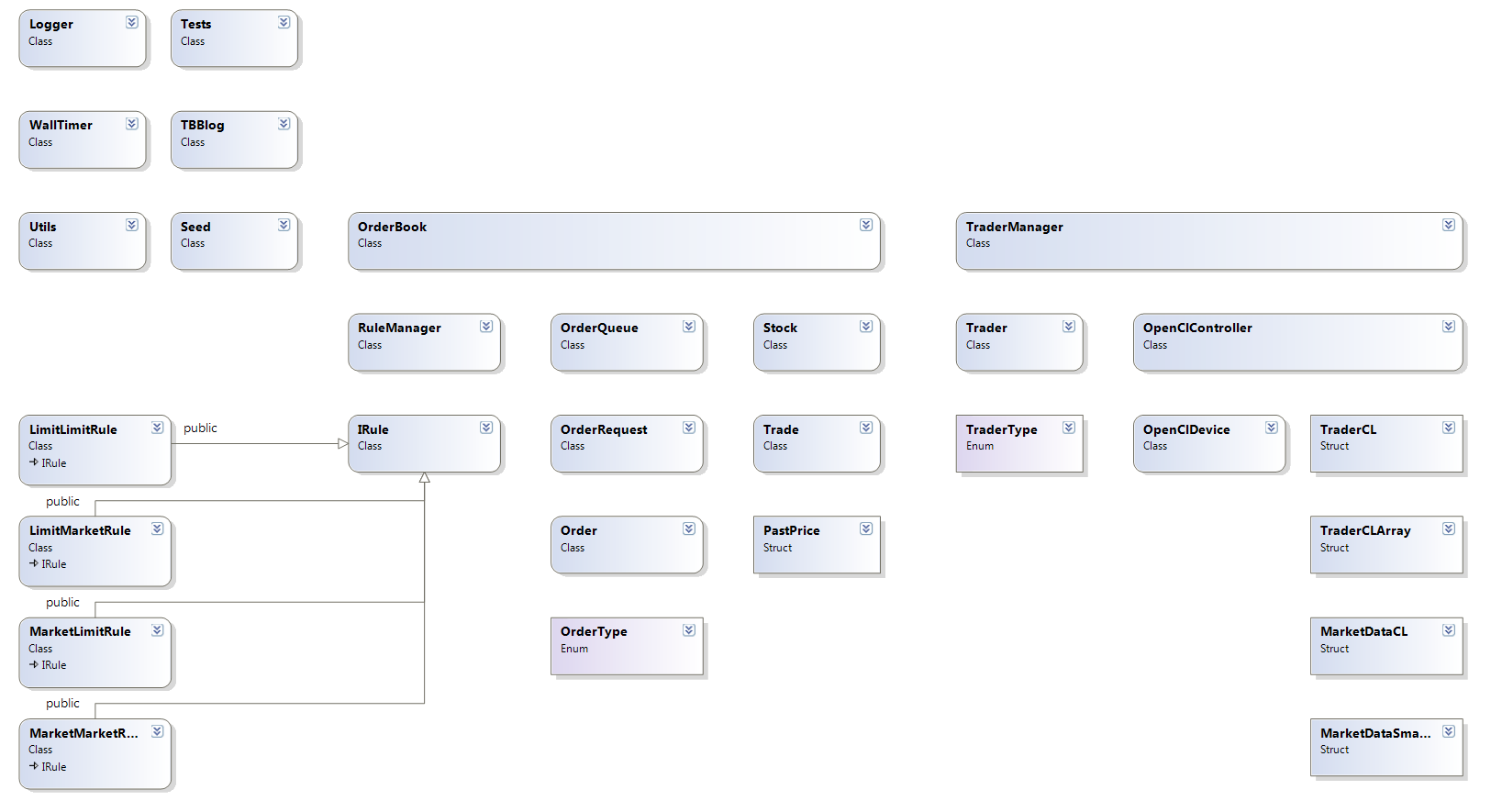
The PT was designed to approximate market making activity (see Background section x for explanation of market making). It would thus try and maintain an amount of cash and volume of holdings within bounds of its starting cash and holdings. In order to sustain this balance it would place orders to try and keep its positions neutral. For example if the trader had less cash that it had started with it would place an order to sell the amount of its current holdings at the last price necessary to get an amount of cash that would bring it back in line with its starting cash. In this way PTs would generate liquidity without having much effect on price (and therefore volatility).

### Momentum Trader (MT)

Unlike the PT which was designed to pursue a market-making strategy, the MT was designed to pursue a profit-making strategy by observing the price momentum of the market and placing orders correspondingly. It would have two windows for observing prices: short term and long term. If the MT observed that the prices had been increasing in the long term (i.e. prices had increased a specified number of times corresponding to long term in the simulation) then it would place buy orders with a view to sell once the price reached its peak. Similarly for decreasing prices the MT would place sell orders. When the MT observed prices beginning to fall (after a rise) or vice versa then it would attempt to sell (or buy) in order to profit from the price movement. This trader had potential to drive price movements. For example rising prices would lead to buy orders, which would lead to rising prices etc.

# Implementation of the Model

The model was implemented using primarily C++ as well as OpenCL. C++ was used as the best OpenCL APIs are available for C/C++ and it also provides object-oriented programming functionality such as classes. It was implemented as a console application using the top level design set out in Section 3, with a class called OrderBook representing an order book and a class called TraderManager acting as the trader manager. The overall class hierarchy that was implemented can be seen in Figure X.



In the figure, arrows indicate inheritance, and classes used in other classes have been placed below the class that uses them where possible. This section will explore the command line arguments and main loop, the implementation of the OrderBook and all of the relevant classes (seen below the OrderBook class in Figure X), the implementation of the TraderManager and all of the relevant classes (seen below the TraderManager class in Figure X), the implementation of the six helper classes seen in the upper left corner of Figure X, and the implemented process flow.

## Command Line and Main Loop

As the model with built as a console application this allowed parameters to be supplied to the application on the command line. This allowed different features of each simulation as well as some more general options to be set in the command line and also allowed the chaining of multiple simulations together by writing a batch file containing multiple commands to run the application. There were a total of 18 command line parameters possible which were as follows:

1. Runs. This parameter sets the number of times to run the simulation. Running the simulation a large number of times such as 500 or 1,000 times decreases the error margins in the results. It must be an integer.
2. Duration. This parameter sets the total number of ticks (one iteration through the main loop) that the simulation will run for. Each tick represents one millisecond of real-time. It must be an integer.
3. Filename. This parameter sets the name of the file to output data (something about this about data?) to, as well as setting the name of the folder that will be created containing additional output data (price, number of buy orders and sell orders at each tick each run). It must be a string.
4. Output per x. This parameter sets how often to update the progress of each simulation in the console window. For example for a simulation lasting 60,000 ticks, if this value was set at 6,000 the console window would be updated every 10% of progress through the simulation. It must be an integer.
5. Logging Level. This parameter sets how much information will be logged during the simulation. This was implemented for debugging purposes. The Logging capabilities of the model will be discussed in Section X. It must be an integer of value 0,1,2 or 3.
6. Number of Random Traders with Time Range 1. This parameter sets the number of RTs that will exist in the simulation with an execution time range as specified in the two values of time range 1 (see below). Must be an integer.
7. Number of Large Random Traders with Time Range 1. This parameter sets the number of LRTs similarly to parameter 6. Must be an integer.
8. Number of Position Traders with Time Range 1. This parameter sets the number of PTs similarly to parameter 6. Must be an integer.
9. Number of Momentum Traders with Time Range 1. This parameter sets the number of MTs similarly to parameter 6. Must be an integer.
10. Time Range 1 Start. This sets the minimum number of ticks that must occur before a trader using Time Range 1 can place another order. Must be an integer.
11. Time Range 1 End. This sets the maximum number of ticks that must occur before a trader using Time Range 1 can place another order. Must be an integer.
12. Wait At End. This parameter sets whether once all runs of the simulation have completed the program will wait for the user to press Enter to continue. Must be an integer.
13. Number of Random Traders with Time Range 2. This parameter sets the number of RTs that use Time Range 2. Must be an integer.
14. Number of Large Random Traders with Time Range 2. This parameter sets the number of LRTs that use Time Range 2. Must be an integer.
15. Number of Position Traders with Time Range 2. This parameter sets the number of PTs that use Time Range 2. Must be an integer.
16. Number of Momentum Trader with Time Range 2. This parameter sets the number of MTs that use Time Range 2. Must be an integer.
17. Time Range 2 Start. This sets the minimum number of ticks that must occur before a trader using Time Range 2 can place another order. Must be an integer.
18. Time Range 2 End. This sets the maximum number of ticks that must occur before a trader using Time Range 2 can place another order. Must be an integer.

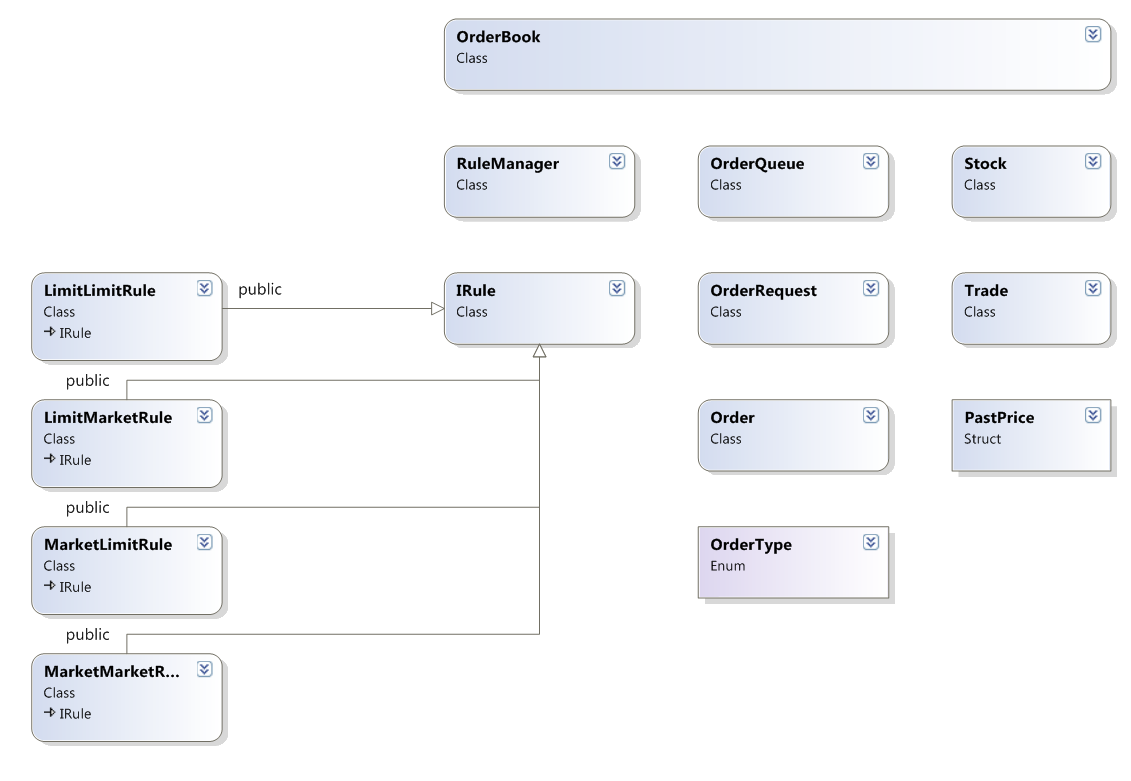
The main loop was implemented as described in Section 3, and can be seen in the Figure X.



The pBook variable is a pointer to an OrderBook object, whilst time is the current tick of the simulation. It is also worth noting that the command to process traders is given by the OrderBook object. This is because the TraderManager was implemented to be registered with the OrderBook. This will be seen later on in this section. The only addition to the design for the main loop set out in Section 3 is the inclusion of output to the console. This was to see how quickly a simulation was progressing. The duration variable is set by command line argument 2. This main loop is contained within a larger main loop which includes which allows the execution of multiple runs of a simulation, the number of which are controlled by command line argument 1. The complete main loop can be seen in Appendix X.

## OrderBook and Classes

This section will explore the implementation of the OrderBook class as well as all of the classes necessary for it to operate. The OrderBook and these classes were the first classes to be implemented as it is the necessary foundation for any trading to occur. The classes heavily involved in the OrderBook can be seen in Figure X.



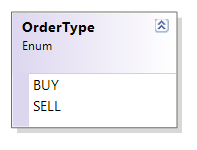
Many of the classes in this section were implemented with a view of having multiple stocks and therefore multiple order books present in a simulation connected to a single set of traders via a single TraderManager. Unfortunately due to time constraints and complexity of some of the TraderManager components multiple stocks and order books have not been tested and thread-safety code has not been written for allowing an OrderBook (or more) per thread. This is future work and further discussion of it can be found in Section X Future Work.

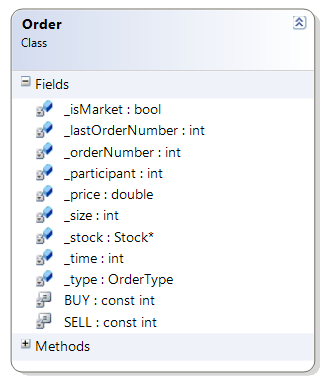
### Stock Class, PastPrice Structur

The stock class was amongst the first implemented and is one of the most simple. It was designed with the existence of multiple stocks in mind and is essentially a container for identifying a particular stock from its symbol (e.g. GOOG for Google [Reference]) or an integer id assigned. It stores the last price of the stock. Its code can be found in Appendix X.

The PastPrice struct is a very simple structure designed to hold a price and the tick time at the point when that price was the current price. Its code can be found in Appendix X.

### Order and OrderType

The OrderType structure is a simple enum representing the direction of an order, which is either buy or sell. This can be seen in Figure X, and the complete code can be found in Appendix X. The Order class uses the OrderType enum as one of its private members to store the direction of the order. All of the private members of the Order class can be seen in Figure X, and are explained as follows:

* \_isMarket is a Boolean storing whether the order is a market (true) or limit (false) order.
* \_lastOrderNumber is a static integer (meaning it is not specific to a single instance of an Order) holding the value of the last order number.
* \_orderNumber is an integer, which holds the id number of this order.
* \_participant is an integer, which holds the id integer of the trader who created and placed this order. (see Section something)
* \_price is a double storing the price of this order. It does not matter what value this hold if \_isMarket is true and the order is a market order.
* \_size is an integer holding the volume of the order.
* \_time is the time at which the order was placed.
* \_type is an OrderType which holds the value either BUY or SELL seen in Figure X. NB that the BUY and SELL seen in Figure X+1 are simply constant static integers which were used in identifying the order direction before the implementation of OrderType.

The most important functions within the Order class are compareBuys(), compareSells() and getNextOrderNumber().

The getNextOrderNumber() function is necessary to ensure that every order has a unique id number as well as to ensure that each time an id is assigned the next one will have an id incremented by one. This makes it possible to tell if an order was placed earlier than another order by looking at their order numbers. The code for this function can be seen in Figure X. Note that whilst it is designed for orders to be constructed in multiple places it is not thread-safe as the static keyword does not ensure thread-safety. This can be seen in Order.h in Appendix X. It is also a protected function meaning that only classes derived from Order (of which there are none) can call this function. It cannot be called from an object outside the Order class. This ensures that the last order number is only incremented from with the Order class when a new instance of an Order is created. The static integer \_lastOrderNumber is initialised to a value of -1 so that the first Order created will have an id of 0.



The two compare functions are necessary for sorting a list of orders and are designed to be passed as the function to determine whether an order should be ranked higher than another order. They match the required signature that a comparison function to be passed into std::list::sort() must take [Reference]. The code for compareBuy() can be seen in Figure X. If both orders are market orders or both orders have the same price then they have equal price priority and thus must be ranked by time. This is done by checking which order has a lower order number (as lower orders have a lower id number). The order with the lower number is ranked higher than the other by the function returning true. This ensures time priority as older order will have been placed first and thus must be completed first. If the two orders are not both market orders or do not have the same price, then if one of them is a market order it will be higher ranked and if one of them has a higher price than the other then it will be higher ranked. Note that the opposite is true for sell orders – the order with the lowest price will be higher ranked.



### OrderRequest and OrderQueue

The OrderRequest class was implemented as an object which represented messages sent from traders to the OrderBook. It acted as a wrapped around an Order, including the Order itself as well as a Boolean representing whether this OrderRequest is a request to add or remove the contained Order from the OrderBook. The functionality to remove (or cancel) Orders on the OrderBook was implemented, however due to the difficulties involved with allowing trading algorithms executing in OpenCL on the GPU complete access to lists of all of the orders they have placed as well as full view of the current Orders on the OrderBook they do not have the functionality to cancel their orders. This will be discussed further in Section 4.3.1.

The OrderQueue class is a queue class specifically designed for OrderRequests. It has the ability to store OrderRequests in vector to which new items can only be added to the back and elements can only be removed from the front of the queue. This is used to ensure time priority of Orders as they arrive at the OrderBook.

### Rules, IRule and RuleManager

In order to match different types of trade (limit and market) four different rules had to be written for dealing with the four cases of the types of trades being matched. These cases were:

* Market-Market. The case when both the buy and sell order are market orders.
* Market-Limit. The case when the buy order is a market order and the sell order is a limit order.
* Limit-Market. The case when the buy order is a limit order and the sell order is a market order.
* Limit-Limit. The case when both orders are limit order.

These cases are all different as the method of determining the price of the resultant trade is different in each of the four cases.

* For the Market-Market case the price is determined as the last price, the value of which is retrieved from the OrderBook.
* For the Market-Limit case the price is determined as the lowest value that the sell limit order will accept.
* For the Limit-Market case the price is determined as the highest value that the buy limit order will accept.
* For the Limit-Limit case the price is determined to be the lowest value from the sell order or the highest value of the buy order depending on the price overlap between the two orders.

Thus four different rule classes were created, the code for which can be seen in Appendix X:

* MarketMarketRule
* MarketLimitRule
* LimitMarketRule
* LimitLimitRule

Each of these classes implemented an interface called IRule, which has three pure abstract methods:

* fitsCritera(). This method checks whether there are any Orders in the OrderBook which match the criteria to match the current Order being matched, i.e. for the LimitLimitRule it checks for a limit Order on the OrderBook of suitable price to be matched with the passed in limit Order.
* processRule(). This method matches the passed in Order to the appropriate Order in the OrderBook and produces a Trade (see Section X).
* ToString(). A method for writing to a string the name of the rule.

Having each of the rule classes implementing IRule allows for simpler control over exercising the rules. The RuleManager class stores pointers to each of the rules. Rules can be added or remove from the RuleManager, which acts a central place to manage rules for the OrderBook. The only other method besides adding and removing rules is the applyRules() method. This method can be seen in Figure X.



This method takes pointers to the OrderBook and the current Order to be matched. It then goes through the list of rules that it holds checking whether an Order can be found on the OrderBook which fulfils the criteria of the current rule. If a match is found then a trade is produced and the Orders are removed from the OrderBook (or have their volume modified if their volumes are not equal). Matches are continually checked for and trades produced using the current rule until no more matches are found, at which point the algorithm moves onto the next rule. The use of the IRule interface allows the RuleManager to apply rules without being aware of which type of rule is being applied. This reduces the amount of code in RuleManager, however it is important to note that the order in which rules are added to the RuleManager is important, as the Market-Market rule has higher priority than the Market-Limit or Limit-Market rules, which equally are more important than the Limit-Limit rule. This is because market orders have higher priority than limit orders. The OrderBook stores a pointer to the RuleManager class allowing rules to be added at the beginning of a simulation in the correct order as seen in Figure X.



Together the RuleManager, IRule and the four types of Rule implement the matching engine.

### Trade

The Trade class acts as a wrapper for two Orders which have been matched together to form a trade. It provides an easy method of keeping the price, size, time and participating party ids of the Trade as well as containing the two Order objects which were matched to create the Trade. The method described in Section 4.2.4 and seen in Figure X generates a Trade when Orders are matched.

### OrderBook

The OrderBook class uses all of the classes described in this section. It implements all of the functionality that is necessary within an order book (see Section 3.1). The private members of the order related to the functionality of the order book can be seen in Figure X, and their purpose is described below.



* \_stock. This is a pointer to the Stock that the OrderBook represents the order book for.
* \_buyOrders and \_sellOrders are template lists from the stl library using Order as the type of the elements of the list. Lists were chosen for this purpose due to the need to sort the buy and sell orders each tick in terms of which has highest priority (price and time are the key determinants of priority). A std::list<T> has a sort function which takes as an argument the comparison function with the signature: bool compare(T first, T second), which returns true if the first argument should go before the second argument. The compare functions for buy and sell orders can be seen in the code in Appendix X, and the compare function for buy orders was described in detail in Section X. Despite having the advantage of ease of sorting lists also have the disadvantage of having an index or find time of O(n) [Reference HPCfE]. This causes the search time in the matching engine to be O(n) in the worst case as the engine searches through the entire list to find a matching Order.
* \_trades is a vector of all the trades that are completed. It acts as central repository for all Trades which have been executed. It is primarily for debugging purposes, allowing problems to be traced back to a specific Trade and therefore a specific Order.
* queue is an OrderQueue of OrderRequests (see Section X). Every time an Order is to be added to the OrderBook it is placed in an OrderRequest and added to the back of the OrderQueue. Each tick the front of the queue is popped and is attempted to be matched.
* \_prices and \_allPrices are vectors of prices. \_prices stores only when the price change and is thus a vector of PastPrices which store a price and the time of that price. \_prices is thus a more efficient way of storing the price of OrderBook across time as it only hold values for when the price changes and the price is assumed to be constant the rest of the time. Using less memory in this manner is useful for communicating these past prices to Traders and to the GPU for use in an OpenCL kernel and the communication to GPU memory is the slowest part of executing code on a GPU. \_allPrices stores the price at each tick and is used for debugging as well as outputting the price at each tick to a csv file for analysis.
* \_openPrice is the starting price and is set either from the Stock’s last price at OrderBook construction or it can be passed as a parameter to the constructor of the OrderBook.
* \_threshold is the minimum difference between two prices that can occur in the OrderBook. It is set at 0.01 representing $0.01.
* \_ruleManager is a pointer to the RuleManager object which is used as the matching engine.
* \_traderManager is a pointer to the TraderManager. The OrderBook stores this as a pointer so that if multiple or different TraderManager were being used then the OrderBook can be connected to the correct one with just a pointer. This is consistent with the design set out in Section X.
* \_time is the current tick.

The OrderBook also stores a number of variables used for debugging and storing data to be output for analysis of the model. These can be seen in Figure X.



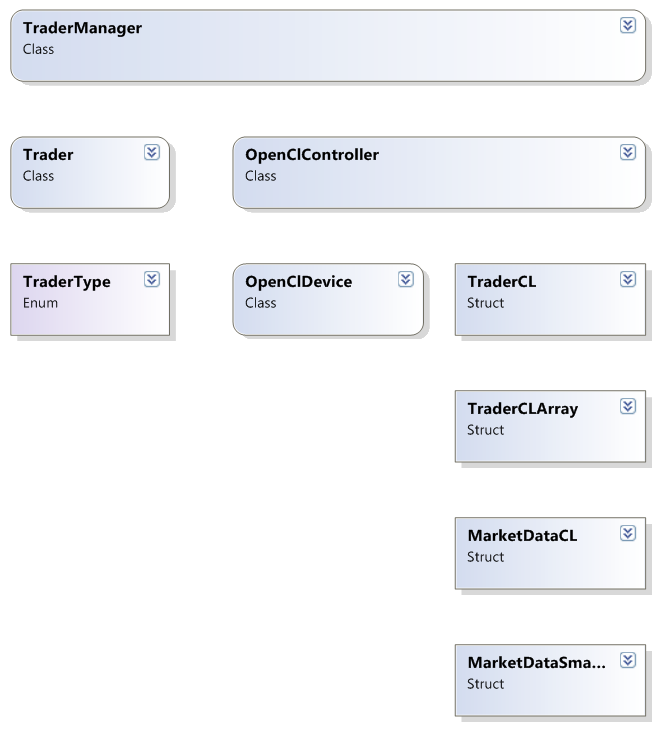
* \_performanceAnalytics is used to toggle whether timings will be taken of the system
* \_timer is a pointer to a WallTimer object which is described further in Section X.
* \_matchTime and \_traderProcTime are used to store the maximum amount of time taken to match orders and to process traders respectively. They are only set when \_performanceAnalytics is true.
* \_matchTimes, \_traderProcTimes and\_oclProcTimes are vectors which store the matching time, trader processing time, and OpenCL kernel execution time at every tick. These are used for calculating the average of each of these times at the end of the simulation, by summing all of the values and dividing by the total number of values.
* \_tradersSize and \_ordersSize hold the size in bytes of \_trades and \_buyOrders and \_sellOrders variables described above.
* \_pastReturns stores the return (the difference between the current price and previous price) at each tick and is used for calculating results for analysis.
* \_spreads stores the difference between the highest priced buy order and lowest price sell order on the book at every tick. This is used to calculate the average spread over a simulation. Average spread is a useful measure of liquidity.

The key section of the code in the OrderBook implements the loop described as necessary for the order book in Section Design. It can be seen in Figure X.

This is the function seen in the main loop in Section 4.1, Figure X. Firstly the queue is checked for any OrderRequests, if there are none then this step of the main loop is complete. If there are pending OrderRequests then the front OrderRequest is removed from the front of the queue. It is then checked whether it is a request to add or remove and order. The functionality to remove orders has been implemented in the OrderBook part of the system, but not TraderManager for reasons described in Section X. The order is then extracted from the OrderRequest before being added to the appropriate list in the OrderBook. It is then matched by the RuleManager to see if it is possible to produce a Trade. The RuleManager itself is responsible for passing any Trades generated to the OrderBook’s public publishTrade() function, which adds the Trade to the OrderBook’s vector of trades as well as instructing the OrderBook to pass the Trade onto the TraderManager. It is also worth noting that try, catch statements are used throughout the code in every class to assist in debugging. In particular their usage throughout allows the easy tracking of an error to a specific place in a specific function, especially as errors are logged. This makes it possible to debug an error that may occur in the middle of a simulation in a large number of runs of the simulation.

## TraderManager and Classes

This section will explore the implementation of the TraderManager class and its associated classes, which can be seen in Figure X. It will also examine the implementation of the trading algorithms in OpenCL.



The implementation of traders and their trading algorithms went through many stages and there is always the possibility of further optimisation of how they are implemented. This will be discussed further in Section X.

### Trader and TraderType

The TraderType is a simple enum that contains the different types of traders that are presently implemented. There are currently four types of trader implemented and these are included in the enum along with a null trader case. This is to deal with the possibility of an error occurring or an additional trader being implemented but not added to the enum. The values of the enum are as follows:

* NULL\_TRADER
* RANDOM\_TRADER
* LARGE\_RANDOM\_TRADER
* POSITION\_TRADER
* MOMENTUM\_TRADER

The four types of trader in the enum are used to determine which type of trader a Trader object represents as well as telling it which trading algorithm to execute in the OpenCL Kernel. The four types of trader are implemented as designed (see Section X). The code for this enum can be found in Appendix X.

A single class was implemented to represent a trader (called Trader). Since the trading algorithm is written in OpenCL and is written to be executed in a kernel it was not deemed necessary to have multiple different classes representing each type of trader as there would be large amounts of duplicate code. The TraderType enum is used to enable a Trader object to identify which of the four types of traders it represents and enables it to execute the correct function within the OpenCL kernel. The TraderType is held within a private variable within the Trader class. In addition to this, the Trader class has a number of other private variables, which can be seen in Figure X, and are explained below:



* \_id stores the id of the trader. This allows us to keep track of which trader places which Orders and executes which Trades.
* \_lastId is a static variable which is used to store the id of the last created Trader object. Every time a new trader is created this value is incremented and the new Trader receives its value as its \_id. This is the same as the method used to ensure that every Order has a unique id value (see Section X).
* \_type is the TraderType as described above.
* \_currentT is the current tick of the simulation. It is updated every tick.
* \_processT hold the value of how often the Trader is allowed to execute its trading algorithm (and thus potentially place orders).
* \_lastCompleteT stores the tick at which the Trader last executed their trading algorithm. It is used to check whether the Trader is allowed to execute their trading algorithm again. This can be seen in the code in Figure X.
* \_startCash and \_startVolume are used to store the amount of cash in dollars that the trader starts with as well as the volume of the stock that the trader holds at the start of a simulation.
* \_cashPosition is used to store the trader’s current cash position i.e. how much cash the trader currently has
* \_cashPosWOrders is used to store the trader’s cash position taking into account all orders that are currently placed but not fulfilled.
* \_pendingOrders, \_completedOrdersV and \_tradesV are used to store a collection of the current instance of Trader’s outstanding orders which have not yet been matched; a collection of orders that have been matched; and a collection of trades that this Trader instance has participated in.
* \_stockPositions and \_stockPosWOrders are used to store the current position in volume of a specific stock. \_stockPositions stores the current held volume of a stock similar to \_cashPosition does with cash, whilst \_stockPosWOrders stores the position taking into account if all outstanding orders were fulfilled. They are maps as they were designed to allow a trader to participate in trading multiple different Stocks (and therefore multiple different OrderBooks).

This function is used to check whether a Trader is ready to execute their algorithm and is called by the TraderManager to which the Trader is registered. It simply checks if enough time has passed since the Trader last executed their algorithm for them to execute it again.

Traders use the structures described in the next section (Section 4.3.2) to transfer pertinent data to and from the GPU. When a Trader’s algorithm has been executed the TraderManager calls a Trader’s processTraderCL() function which translate the data from the structure received from the GPU into whether or not to place an order. This can be seen in Figure X. Note that when any Trader is ready the TraderManager executes a kernel which executes every Trader’s algorithm simultaneously, thus the function show in Figure X includes a check to see whether the Trader is allowed to place an Order.



### Structures for OpenCL

There are a number of different structures which are declared both in OpenCL (see Appendix X) and in C++ (see Appendix Y). They are declared identically in both places as otherwise the struct would not be usuable in either place. They are used to pass a larger number of parameters into the OpenCL kernel than would be permitted if each variable were passed as a separate parameter. The structures are:

* TraderCL
* TraderCLArray
* PastPrice
* MarketDataSmallCL
* MarketDataCL

The PastPrice structure was already discussed in Section X, however each of these other structures will now be explored in more detail.

#### TraderCL and TraderCLArray

The TraderCL struct was used to act as a container for all information that a Trader’s trading algorithm would need to know about the current state of the Trader object. It therefore acts as a simplified version of the Trader class, but with no methods, and only containing data. The information that it holds can be seen in the code in Figure X and is explained below:



* cashPos and cashPosWO hold the current cash position and cash position taking into account unfulfilled orders.
* volPos and volPosWO do similarly for the current volume position.
* startCash and startVol hold the starting cash and volumes.
* id and type store the id and type of the trader, with the type being expressed as an integer. The mappings from semantic trader types to integer values can be seen in the code in Appendix X.
* isMarket, price and volume are used to store information about any type of order that the trading algorithm generates. They store the type of order (market or limit), price and volume respectively.

The TraderCLArray struct acts a container for an array of TraderCL structs. It is used to store a TraderCL struct for each Trader registered with a TraderManager. It can thus be used as a buffer between the CPU part of the system and the OpenCL kernel. It also stores the total number of traders as well as the number of each trader type. This is used to help access specific TraderCL instances within the array.

#### MarketDataSmallCL and MarketDataCL

These two structures are used to store information about the current state of the OrderBook. The information stored includes:

* The current total volume on the buy side of the OrderBook.
* The current total volume on the sell side of the OrderBook.
* The last price in the case of MarketDataSmallCL, or an array of PastPrices in the case of MarketDataCL.
* The number of last prices.

### OpenCLController and OpenCLDevice

The implementation of the ability to execute OpenCL kernels was separated into two separate classes: OpenCLController and OpenCLDevice. This was done to separate some of the lower level OpenCL API calls out from the higher level functionality which should be exposed to the TraderManager. Thus the OpenCLDevice class manages the lower level OpenCL API calls, whilst the OpenCLController manages higher level functions. The OpenCLDevice has methods for the following:

* Setting up build options for the kernel. This includes setting any #defines that need to be present in the OpenCL kernel.
* Building the kernel.
* Setting up the buffers for relaying data between the rest of the system and the GPU.
* Setting up the arguments for the kernel
* Running Read and Write commands to read/write the contents of the buffers to the GPU.

The signatures for these methods can be seen in Figure X.



These methods are meant to be called in the order above. The first two methods only need to be called once as once the kernel is compiled it does not need to be recompiled whilst the application is running. The other four methods are for updating the data passed in to the kernel and writing and reading the data at the beginning and end of the kernel execution. All OpenCL API calls within these functions are blocking to ensure that the application cannot progress any further until the function has completed. This is particularly important for the enqueueReadBuffer and enqueueWriteBuffer calls as if the TraderManager attempted to read from the buffer before it had been updated to the latest values produced by the kernel, the wrong data could be passed to a Trader. The data and buffers passed to the OpenCL kernel will be discussed in the next section (Section X). All OpenCL API calls are also within try catch statements to ensure that any errors are caught. This is possible since the OpenCL API now includes a C++ API, which generates exceptions specific to OpenCL. These exceptions can be caught using a try and catch statement and passed up the call stack. Since try catch statements are used consistently throughout all code with potential to throw an exception, informative exceptions can be generated for debugging including exactly where an error occurred and what the error was. To enable OpenCL exceptions the following define must be used:



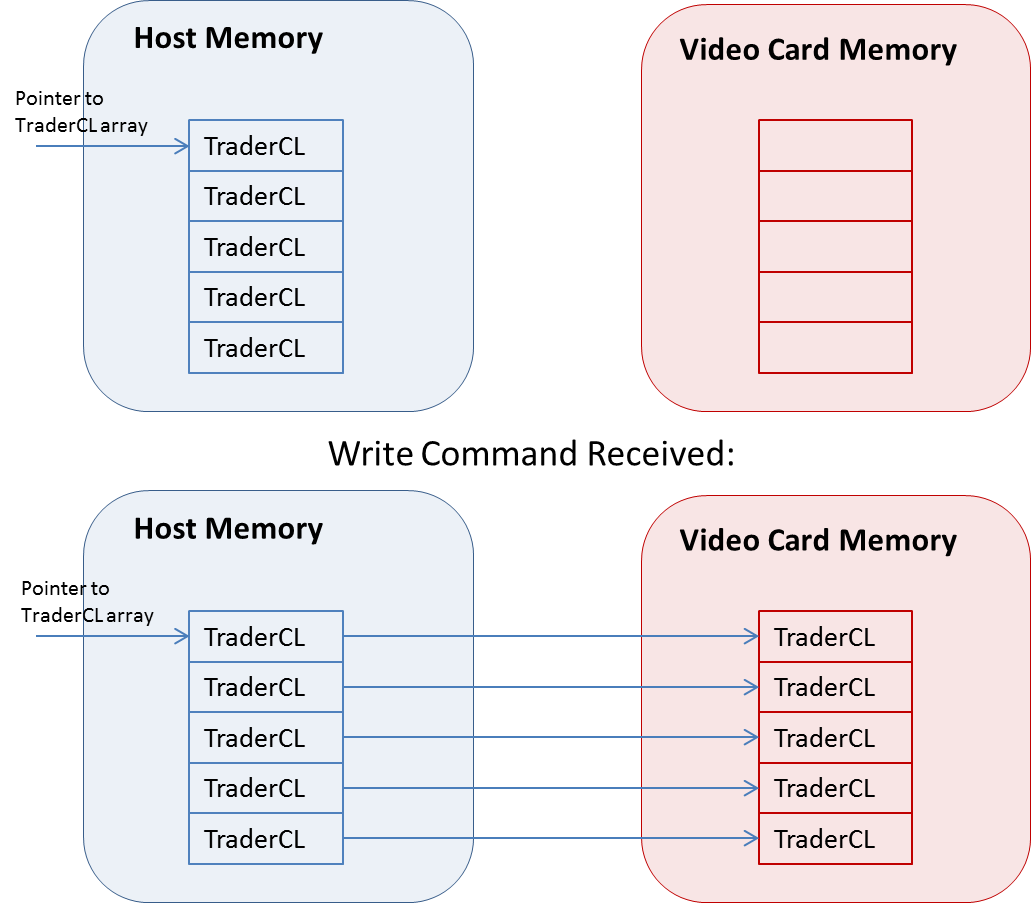
The blocking and exception catching can be seen in Figure X, which shows the code for the OpenClDevice::EnqueueRead() function. This function also includes the ability to check how long the kernel took to execute which is used in the analysis of the performance of the system.



Catches exceptions if the command with the try block fails.

Sets that the call is blocking.

Data is relayed to and from the GPU using the enqueueReadBuffer() and enqueueWriteBuffer() commands. The commands are given the pointer on the host to the data of the buffer (e.g. the array of TraderCL structures). These commands explicitly create a map between a section of memory on the host and the video memory. On the write command the contents of the memory on the host are explicitly copied into the video memory and vice versa for the read command. This can be seen in Figure X. This explicit memory is potentially the slowest part of the program involving OpenCL.



Unlike the OpenClDevice class the OpenClController class is designed to be a singleton. This enables us to keep only one instance of the OpenClController allowing us to make calls within an OpenClDevice such as building a kernel just once. It exposes higher level methods for the execution of the kernel including those seen in Figure X:



* The RefreshBuffers() function is used to update the array of TraderCL which store the current state of each Trader that is registered with the TraderManager, as well as update the MarketData that is passed into the kernel.
* The SetText() function is used once to set the text of the .cl file to be compiled as a kernel.
* The SetupFirstTime() function is only called before a simulation is begun. It sets whether the cl::Device within the OpenClDevice class has profiling enabled. It also calls methods to setup build options and to compile the kernel within the OpenClDevice. This is done before a simulation is run as it takes several seconds (usually around 3) to build the kernel, which can distort any timing data collected for analysis.
* The UpdateBuffersAndArgs() is to be called after RefreshBuffers() and calls the functions necessary to update the cl::Buffers within OpenClDevice with the new data from RefreshBuffers()
* The Run() function simply executes the kernel (and calls the necessary read and write buffer calls within OpenClDevice), it returns the amount of time in milliseconds that the kernel took to execute if profiling is enabled.

In order to execute the kernel the TraderManager class only has to call the RefreshBuffers(), UpdateBuffersAndArgs() and Run() functions thus the OpenClController exposes more high level intuitive functionality to the TraderManager.

It is also noted that the OpenClController is a singleton class meaning that only one instance can exist at once. This was accomplished by making the destructor and constructor private within the OpenClController class meaning that they could only be called within the class itself. The only method exposed for gaining access to an OpenClController object was thus a method which returns a static pointer to an OpenClController object. The code for this can be seen in Figure X. Note that the \_instance variable is a private static pointer to an OpenClController object, and the \_instanceFlag is a private static Boolean which stores whether an instance of the OpenClController has been created. Thus the only way to get an instance of the OpenClController is by calling the GetInstance() method which creates an instance if no existence already exists or returns the only existing instance if already created.



### Traders OpenCL

The OpenCL code consisted of a kernel and several separate functions representing the trading algorithms of different trader types. It was implemented so that to add an additional trader type one could just write a new algorithm and include it in the kernel. The code for the kernel can be seen in Figure X.



The arguments to the kernel are explained as follows:

* offset is the seed of the current simulation. It is used in setting up the stream of pseudo-random numbers generated by the MWC64X random number generator.
* traders is a pointer to the start of an array of TraderCl structures which are stored in global memory as it is necessary to both read and write to the structures and is necessary for the section of memory to be copied to and from system memory.
* prices is a pointer to the start of an array of PastPrices which are stored in the constant section of global memory so that they can be accessed by any work item and cannot be altered.
* data is a MarketDataSmallCL structure used to store the rest of the information about the OrderBook (see section X for the contents of this structure).

The kernel first executes the functions for determining whether the prices have been rising in the long and short term. This is used to set two booleans which will be used by MT’s trading algorithm. It then executes each trader’s algorithm in a separate work unit. There are four trading algorithm functions:

* RandomTrader
* LargeRandomTrader
* PositionTrader
* MomentumTrader

These four functions will be explored in more detail in the following sections.

#### Random Trader

The code for this trader can be seen in Figure X.



First of all a random number is generated (roll) and is then used to determine whether the algorithm should generate a buy or sell order. It is also used in determining whether the order generated should be market or limit, and is used in setting the price and volume. All of these determinations are made by comparing the roll value to a number of pre-set thresholds which are defined when the kernel is built.

#### LargeRandomTrader

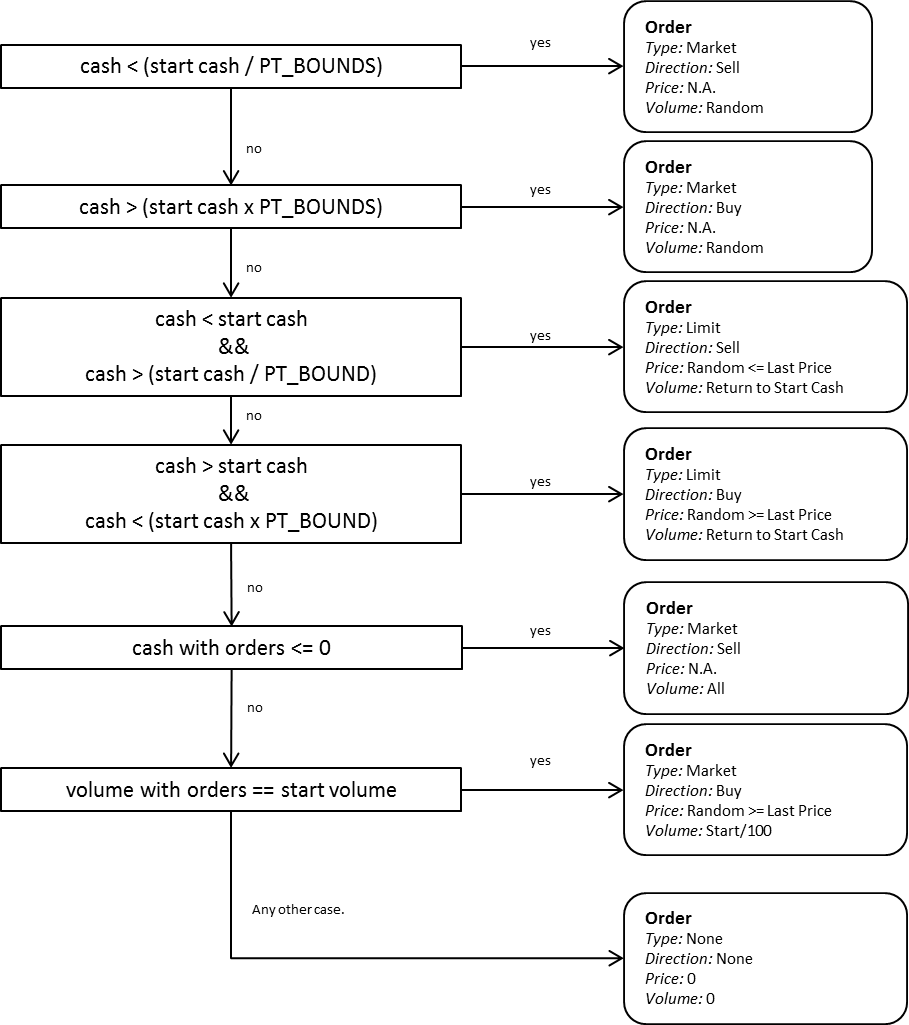
The code for this trader can be seen in Figure X. The only difference between this algorithm and that of the RT seen in Section 4.3.4.1 is that the method for determining whether an order should be a buy or sell is different. Instead of using a random number to determine the direction of the order, the LargeRandomTrader algorithm checks whether the total volume of buy orders is larger than the total volume of sell orders on the order book using data contained within the passed MarketDataSmallCl variable. If the volume of buy orders is larger than the volume of sell orders then this algorithm generates a sell order in order to help balance the order book better. Likewise it does vice versa if the volume of sell orders is larger.



#### PositionTrader

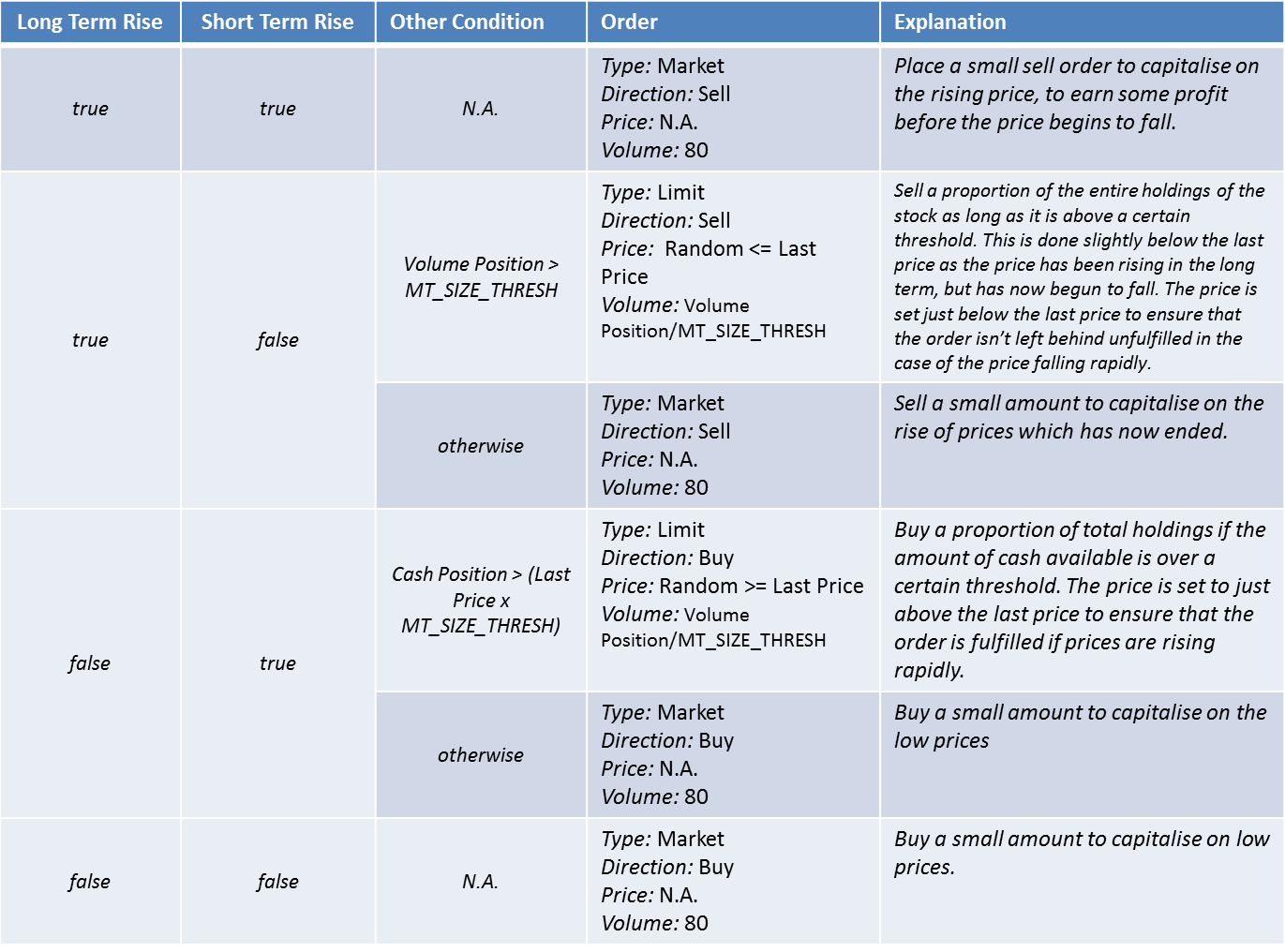
The code for this algorithm can be seen in Appendix X. It is separated into dealing with a number of different cases regarding the state of the trader. These can be seen in the if and else if statements. The checks and the corresponding actions they produce are as follows, these can be seen diagrammatically in Figure X:

1. Is the cash position less than starting cash divided by a fixed amount (PT\_BOUNDS)?
   * If so a market order to sell a random amount is generated in order to increase the amount of cash that the trader has.
2. Is the cash position greater than the starting cash multiplied by a fixed amount (PT\_BOUNDS)?
   * If so then a market buy order is generated to buy a random amount to decrease the amount of cash and increase its position in volume.
3. Is the cash position less that the starting cash, but greater than in 1.?
   * If so then a limit order is generated to sell the amount approximately necessary to generate enough cash to return the trader to their starting cash position. The price is set randomly and slightly below the last price.
4. Is the cash position greater than the starting cash, but less than in 2.?
   * If so then a limit order is generated to buy the amount approximately necessary to generate enough cash to return the trader to their starting cash position. The price is set randomly and slightly above the last price.
5. Is the cash position taking into account unfulfilled orders negative?
   * If so then sell the entire current holdings using a market order.
6. Is the volume position taking into account unfulfilled orders the same as the starting volume position?
   * If so then buy an hundredth of the starting volume at a price slightly above the last price in a limit order. This used to set this trader off by selling a small amount of its holdings at a price which it would not normally do so. Otherwise the trader would be permanently stuck in the case 7 as it doesn’t fulfil any of the criteria for cases 1-6.
7. Any other case.
   * Do nothing.

****

#### MomentumTrader

The code for this trading algorithm can be found in the Appendix X. It takes in as parameters the Boolean values indicating whether the price has been rising for a long time and whether the price has been rising in the short term. It uses these values to determine what type of order to generate. The order generated based on the situation can be seen in the third column, and the explanation of why this order is generated is in the fourth column. The value of MT\_SIZE\_THRESH is set as a build option during the compilation of the kernel.



### TraderManager

The TraderManager class represents and implements the design described in Section X. It is responsible for manager traders and executing their trading algorithms, as well as notifying them of any trades and relaying their orders to the OrderBook. The private members of this class can be seen in Figure X and are described below.



* \_randomTraders, \_largeRandomTraders, \_positionTraders, \_momentumTraders and \_allTraders are all vectors of pointers to Trader objects (which store their cash positions etc.) The vector \_allTraders contains the same pointers as are contained within the other four vectors. Vectors are used as their indexing time is O(1) [reference]. It is thus fast to iterate through the vectors. In order to process traders the TraderManager can simply go through the vector \_allTraders getting TraderCL data and passing it back once the algorithm has been executed using the OpenClController and kernel.
* \_tradersBuffer is a TraderCLArray struct (see Section X). It is used as a buffer for the data for each trader that is passed to and from the GPU. The pointer to the start of the array is used as the host pointer which is passed into the cl::Buffer in OpenClController and OpenClDevice.
* ocl is a pointer to the instance of the OpenClController.
* \_totalTraders stores the total number of Traders that are registered with the TraderManager
* \_firstTime is a bool which indicates whether the process with the OpenClController has been run before. On the first time the kernel is compiled and the buffers are set. Once this has occurred this variable is set to false so that there are no unnecessary repeat of processes.
* \_procTime holds the value of the time in milliseconds that it took to execute the kernel – the value returned by the Run() function in OpenClController.
* \_minTraderProcT holds the minimum number of ticks that a trader takes between placing orders.
* \_currentT and \_lastT store the current tick number and the last time the kernel was executed.

The function which is called from the main loop (see Section X) can be seen below in Figure X. It corresponds to ‘ProcessTraders’ step of the design (see Section X). First of all the \_tradersBuffer is updated with the latest values from each Trader in the WriteBuffers() function. Next a MarketDataCL structure is populated using data from the order book which is accessible via a pointer which is passed in as a parameter to this function. Next these two data structures are sent to the OpenClController to update them for use on the GPU. Finally the kernel is executed with a global size equal to the number of traders (resulting in one work unit per trading algorithm), which updates the values in the \_tradersBuffer. Finally the \_tradersBuffer is iterated through extracting any new values and sending them to the correct Trader in the ReadBuffers() function.



## Helper Classes

These final six classes are used for a variety of different supporting roles.

### Seed

The Seed class is a singleton implemented in the same manner as the OpenClController singleton (see Section X). It is used to act as a central place to store the seed of a simulation. The seed is accessed by any function that needs it by calling the GetSeed() function, whilst the seed can be refreshed using the Update() function. This simply sets the internal seed value to be the current system time (retrieved using the time(0) command). This command is guaranteed to return a different value as the system time is always incrementing, as long as it is not called within too quick a succession.

### Utils

This class contains a collection of useful methods which are static to enable them to be called from anywhere that includes the Utils header file. The methods implemented include:

* Find the minimum value in a vector of doubles.
* Find the maximum value in a vector of doubles
* Find the arithmetic mean value in a vector of doubles.
* Find the standard deviation of a vector of doubles.
* Merge two strings or constant character arrays together
* Convert an integer to a string.
* Convert a float or double to a string.
* Convert a string to a wstring (wide string), which is used in the Logger class for accessing folders in the file system.
* Merge two pieces of exception information into one string.
* Convert an OpenCL exception into a string for use in an error message.
* Resurrect an exception which catches all possible types of exception (an OpenCL exception, an std exception, any other type) and turns the error into a string.

### Logger

The Logger class is a singleton implemented in a similar manner to Seed or OpenClController. It is used to log varying levels of output data. It is a singleton so that only one instance of the Logger exists at once. This is essential as it opens numerous file streams, and it is preferable to have only a single file stream for a single file at once. The Logger has four levels of logging each represented by an integer:

* Debug – 0. This level logs every single consequential action that the system takes whilst running a simulation. It includes all information that would be logged at the following log levels.
* Info – 1. This level records most actions that occur within the system, e.g. creation of an Order or a Trade. It does not log almost every function call unlike the debug level.
* Warn – 2. This level only records information when a function produces a result which may not be acceptable.
* Error – 3. This level only records when an error occurs and is written to within every catch statement in the application.

The Logger outputs information depending on the log level set into an html log file, in which the above log levels are colour coded to make it easier to identify when there are problems. The Logger was essential in identifying a number of very specific errors which would have been extremely difficult to identify otherwise. An example of these errors is that a rule in the RuleManager would return true for the fitsCriteria() function, however the processRule() function would incorrectly match the order to the first Order in the list of orders, when the Order that caused fitsCriteria() to return was the second Order.

The Logger is also used to output the price, the number of buy and sell orders, trades and orders all to separate files for use in debugging and observation.

### TBBlog

This is a function object used in the calculation of volatility per minute data. Intel’s Threaded Building Blocks (TBB) is used to accelerate this calculation from taking several thousand milliseconds to a few milliseconds. It is called as the function object inside a parallel\_for loop in OrderBook which is useful for executing a data-parallel operation on a number of pieces of data [Refernce]. The TBBLog functor takes the natural logarithm of the return at each tick of a simulation.

### WallTimer

The WallTimer class is used to measure the time that passes whilst another function executes. It gets the current CPU frequency using the QueryPerformanceFrequency command. It then gets the current value of Window’s inbuilt high-resolution performance counter using the QueryPerformanceCounter. This counter is accurate to microseconds [reference HPCfE lecture notes]. In order to determine how much time has passed since the timer was started the GetCounter() function gets the current count with QueryPerformanceCounter and calculates the time that has passed in milliseconds by taking the difference in counter values and dividing them by the CPU frequency. It is heavily based on code found at: <http://stackoverflow.com/questions/1739259/how-to-use-queryperformancecounter>

### Tests

The Tests class contains a number of unit tests for the Stock and Order classes.

# Analysis

## Verification of Motivation

In order to verify whether the implemented system is faster using the GPU or using a CPU, three different tests were run. The first varied the total number of each type of trader whilst keeping all other parameters constant. The second varied the average trader T (the average minimum time required between orders for a trader) whilst keeping all other parameters constant. The third and final test involved increasing the duration of a simulation whilst keeping all other factors constant.

### Varying Total Number of Traders

The total number of traders was varied. An equal number of each type of trader was used and the numbers of each trader type used were: 0,1,3,6,12,18,24,30,36,42,48,60,72,84,96 resulting in total numbers of traders ranging from 0 to 384. The model parameters were thus:

* Number of RTs: 0->96.
* Number of LRTs: 0->96.
* Number of PTs: 0->96.
* Number of MTs: 0->96.
* Trader Range 1: 2,000-4,000.
* Duration: 60,000.
* Runs: 1000 for GPU, 50 for CPU.

The most encompassing measure of performance is the total time taken to run a simulation. It is faster to run a simulation on the CPU rather than the GPU in all cases of the model whilst varying the number of traders. This can be seen in Figure X.

The speedup of using the CPU over the GPU beings at one as for zero traders there are no traders with OpenCL kernels to execute, thus the only part of the system which can use either the GPU or CPU is never run. Thus the entire program is run on the CPU and results in the speedup being one and the amount of time taken being equal across CPU and GPU runs. As the number of traders is increased the time taken between running on the CPU and GPU begins to diverge, with the GPU time increasing faster than the CPU time. This is because as more and more instances of the kernel are executed on the GPU, the data has to be transferred back and forth from the GPU resulting in a longer time. This is not the case with the CPU as the data is already cached on the host meaning that to execute the OpenCL kernel on the CPU no explicit copy of memory has to occur and the data is already stored on the host. (See section X for explanation of copying memory using a host pointer in OpenCL). The speedup of the CPU over the GPU initially rapidly increases but then appears to tend towards a value around 2.5x. We would expect that the speedup of the CPU over the GPU should decrease with a sufficiently large number of traders as the probability of multiple traders being executed simultaneously is higher. This will be discussed in more detail in the next section (Section 5.1.2).

The other key metrics to observe to compare the performance on the CPU vs the GPU are the average and maximum kernel execution times and the average trader time. These can be seen in Figure X.

The average and maximum kernel times on the CPU have a linear relationship with the number of traders, whilst the times on the GPU have a second order polynomial relationship with the number of traders. It is possible that the CPU ones have a second order polynomial with a shallow gradient otherwise I have no idea how to explain this. The maximum time taken on the CPU diverges considerably more with the average time on CPU than the two lines do for the GPU. At points the maximum time taken on the CPU is larger than the average time taken on the GPU. Another important measure of the performance between the CPU and the GPU is the average trader process time. This includes the kernel time as well as any other operations (such as memory copies) that must be performed to process traders. The average trader time for the GPU and CPU vs the number of traders can be seen in Figure X. Both curves are once again a second order polynomial, indicating the bottleneck within the algorithm for processing traders is not the kernel execution which in the above figure had an O(n) relationship with the number of traders. This being the case, the GPU is once again considerably slower than the CPU in processing traders. This can be explained by its longer kernel execution times as well as the need to explicitly copy memory around in the GPU case. If the number of traders that were running simultaneously were increased then we would expect to see the GPU outperform the CPU.

The speedups for the average kernel and trader process time speedups for running on the CPU over the GPU can be seen in Figure X. Both graphs tend towards the same value indicating that the boost in trader process time is attributable to the booth in kernel execution time on the CPU. The average trader time speedup curve begins at a value of one as with little to no traders the speedup should be one as the kernel is not being executed on the GPU or CPU often if at all (for no traders). The average kernel speedup begins extremely large and decreases rapidly to hold approximately the same value as the average trader process time speedup. This is because REASON.

### Varying the Average Trader T

The average trader T can be varied by altering the possible range of time that will be used as bounds for generating trader Ts. As discussed in Section X the rand() function produces a uniformly distributed random number, which means that by halving the bounds of the distribution we can expect to roughly halve the average value. The bounds used and the average values that they produced, as well as the standard deviations produced can be seen in Table X. The other parameters besides the changing range were:

* Number of RTs: 12.
* Number of LRTs: 12.
* Number of PTs: 12.
* Number of MTs: 12.
* Trader Range 1: Varying as in Table X.
* Duration: 60,000.
* Runs: 1000 for GPU, 50 for CPU.



The total time taken to complete the simulation for both the CPU and the GPU can be seen in Figure X. The time taken on the CPU for decreasing average trader T is always less than the time taken on the GPU, although at the lowest value on the graph the error bars overlap. It is worth noting that attempts with ranges of 10-25 and 25-50 were successful and these time taken values can be seen in Section X, however it was not possible to run the simulation on the CPU with this ranges as the application crashes. The speed up of the CPU vs the GPU can be seen in Figure X. As the average trader T tends to infinity the section of the system which can be executed on either the GPU or CPU is executed less and less frequently. At the limit of infinity that section will never be executed and thus the entire system is running on the CPU. This explains why the speedup of the CPU vs the GPU tends to one as the average trader T tends to infinity.

The speedup begins to decrease again below an average trader T of approximately 150ms, although it is difficult to say how much due to the size of the error bar for the value at ~74ms. This may due to the fact that as the average trader T decreases, and the size of the range of possible values for trader T decreases, the probability of more than two traders having the same trader T increases. It would be expected that the CPU would be faster for small numbers of traders with the same trader T as the clock speed of the CPU used is considerably faster than the clock speed of the GPU. In fact the CPU used (an Intel Q9550) was clocked at 2.83GHz [reference System Information] whilst the Radeon 6970 GPU used was clocked at 880MHz [Catalyst control center]. We would thus expect that maximum speedup to be 2.83GHz/0.88GHz = 3.22x which is just about within the error margin of the largest speedup on the graph in Figure X. The number of traders that would have to have the same trader T before the GPU began to improve on the CPU’s performance advantage would thus be approximately 12 as the CPU used was a quad core: Max Speed up 3.2x \* 4 cores = 12.something. The CPU may also gain a performance boost as to execute the kernel on the GPU data has to be moved into the video card memory, whilst when executing on the CPU it is already in the correct place for execution.

Given that the CPU offers some speed up over the GPU in this situation we would expect the CPU to have both lower average kernel execution times and also lower average trader process times. These, along with the max times can be seen in Figures X and Y. There is a consistent separation between both the average times and the max times. The max trader times have been omitted from the graph due to large error values which are of the same order of magnitude as the data points. These can be seen in Appendix X. The consistent separation between the times can be seen in better in Figure Z, which shows the speedup of the CPU over the GPU for both average times.

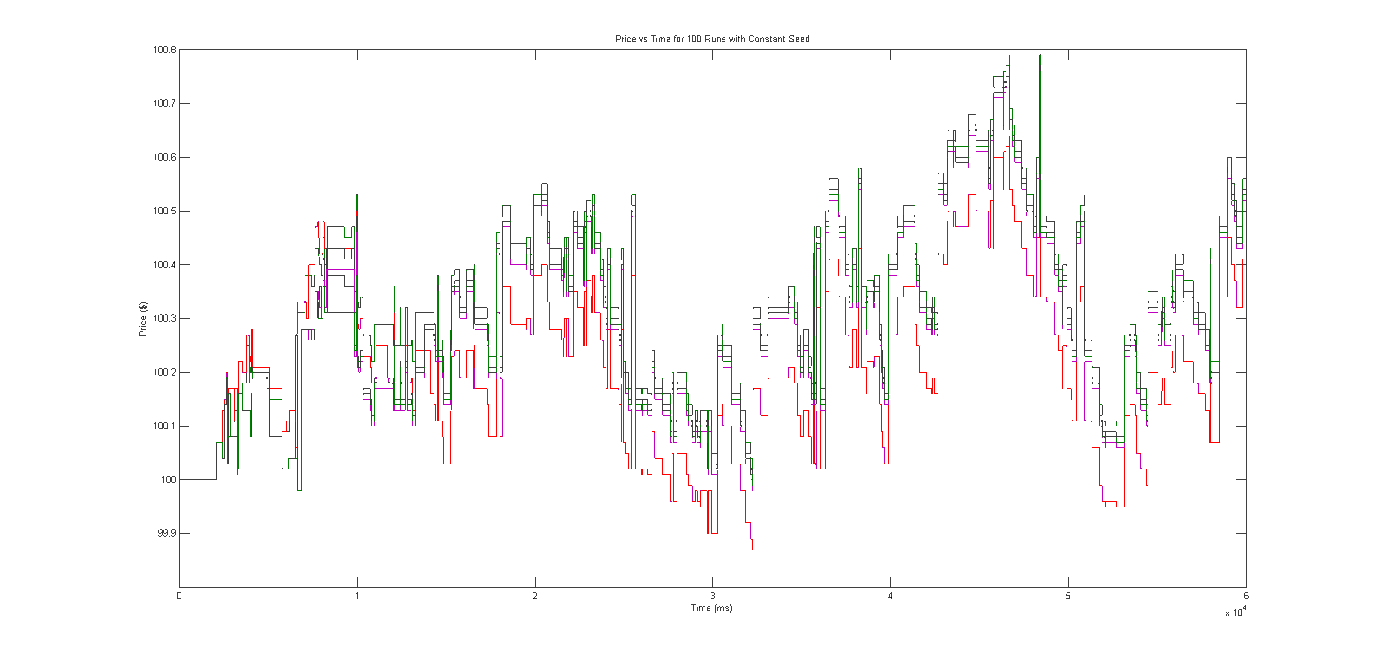
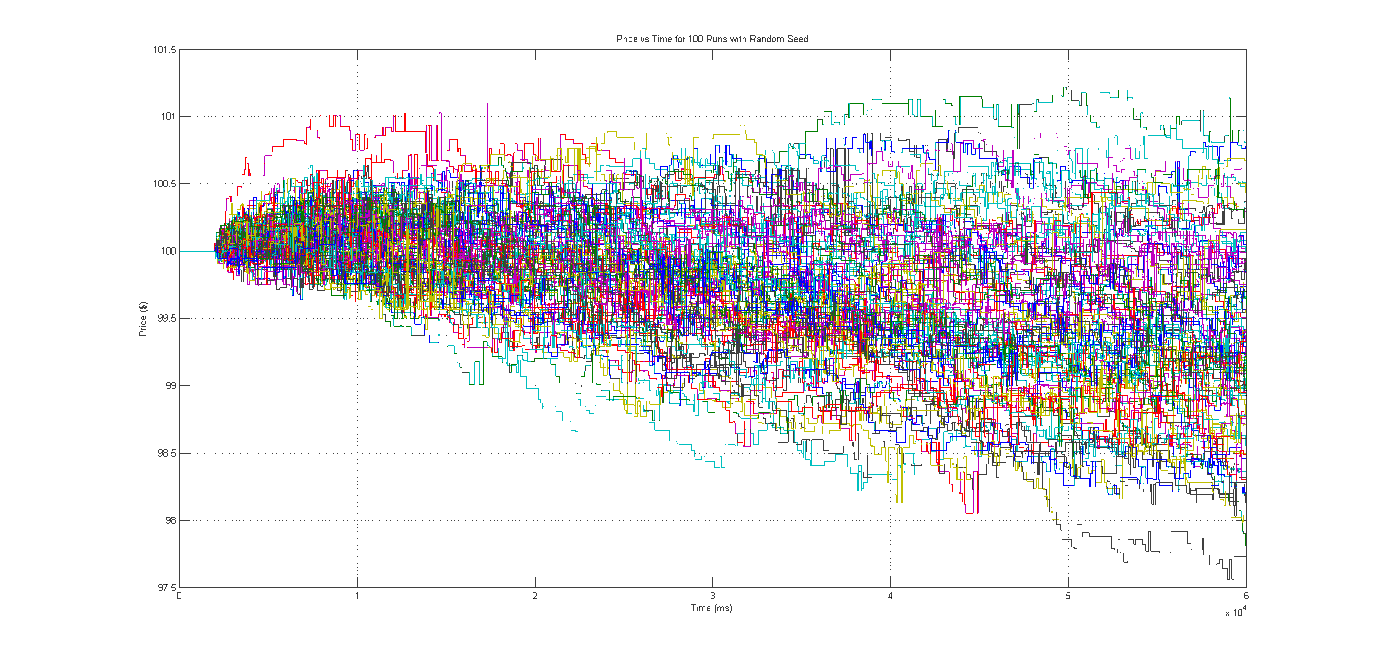
The curves closely mirror each other indicating that the kernel execution time is likely additive to a cost overhead of processing traders. This is because there is a section of code along with the OpenCL kernel which will only be executed when a trader is ready to trade. See Section X for the explanation of how the traders are executed.

### Varying the Duration of the Simulation

### Varying the Number of Traders With the Same Trader T

## Verification of Determinism

The model was designed to be as deterministic as possible (see Section X). Due to the nature of such a complex system the easiest way to observe determinism is to observe the value of the price at each tick of a simulation. Thus 100 simulations were run in which the value used to seed the random number generators was based off of system time which is always different at the start of a simulation, and 100 simulations were run keeping this value constant. The price at each tick of each simulation for these two scenarios can be seen in Figure X and Y.



If the model was completed deterministic, we would expect to single line in Figure X, however this is clearly not the case. There are approximately six different lines which are the same as each other plus or minus a set value. This indicates the system is deterministic within a range a possibilities, i.e. for 100 runs there are approximately six different outcomes that can occur and running the system a small number of times is highly likely to produce a duplicate simulation.

Other important measures of the determinism of the system are the financial characteristics of the model. These include the volatility per minute, the average spread and the number of trades per second. Another good indication of traders in the model being deterministic is to look at the average trader T.

The volatility per minute from the scenarios above can be seen in Figures X and Y.

From Figure Y, 75% of 100 runs have the same volatility, indicating that the majority of the time the system will be deterministic. The other volatility per minute values are produced by the variations that could be seen in the Price vs Time diagrams above (Figure X). The average spread follows a very similar pattern to the volatility graphs and can be seen in Figures X and Y.

Finally the number of trades per second and the average trader T holds a variety of values as the traders are generated with different trader Ts when the random number seed is varied. These values stay constant across all 100 simulations when the seed is kept constant. This can be seen in Figures X,Y,Z,A. Indeed the trader Ts generated for any given seed remain the same as long as that seed is kept. This can be seen in Appendix Data somewhere.

Thus we can conclude that for a given seed there is a high probability of producing a simulation that is identical to another. In a test with 100 runs, 75% of simulations had the same prices, volatilities, average spreads, average trader Ts and trades per second, indicating a 75% chance of producing exactly the same simulation.

## Model 1

The first model to be analysed had a baseline of 12 of each types of trader (RT, LRT, PT, MT) with a trading range of between 2000 and 4000ms. These values were chosen to ensure that this baseline model was not only quick to run enabling thousands of simulations to be run, but also so that the effects of the number of each type of trader as well as the size and values of the trader range could be tested. All simulations were run for a duration of 60,000ms of simulation time.

### Performance Analysis

The performance of the model was assessed against a varying number of each type of trader (only one type varying at a time), the total number of traders (with an equal number of each type maintained) and the average trader time. The key metrics for measuring performance were:

* Total Time
* Average Kernel Execution Time
* Maximum Kernel Execution Time
* Average Matching Time
* Average Trader Process Time
* Maximum Trader Process Time
* Size In Memory

Graphs which include comparisons of the four type of trader can be found in X the Appendix.

#### Effect of Random Trader

Increasing the number of RTs increases the total time taken to complete a one minute simulation as a squared function of the number of RTs. The speed up over real time decreases correspondingly. This can be seen in Figure X.

The relationship between the number of RTs and the time taken to match orders is a linear function, thus this cannot be the cause of squared relationship between number of RTs and the total time taken. This linear relationship can be observed in Figure X. This linear relationship is expected as the matching algorithm iterates through the list of orders in the opposite buy/sell direction to the order being matched. List indexing is an O(n) operation, thus we would expect the matching algorithm iterating through the list of orders to also be an O(n) operation.

The relationship between the number of RTs and the average and maximum times taken to execute the kernel on the GPU are both squared relationships. This can be seen in Figure X.

The relationship between the number of RTs and the average time taken to process the traders each tick is a squared relationship. The value of the average trader process time is less than that of the average kernel execution time as the kernel is not executed every tick, but rather only when a trader is ready to process. It is also worth noting that the maximum time taken to process traders remains roughly constant, but is significantly larger – it is of the order of ms, rather than 0.01ms. This can be seen in Figure X.

#### Effect of Large Random Trader

The relationship between the number of LRTs and the total time taken to complete a one minute simulation is a second order polynomial relationship. This can be seen in Figure X.

The average matching time is logarithmically related to the number of LRTs. This is because the LRT is designed such that it only places orders when there are a larger number of orders on one side of the book than the other. It then places its large order in such a way that will create trades with a number of the orders of the opposite direction on the book. Thus the order placed will always be matched within the first several in the opposite direction, making it unnecessary to scour the entire list. This can be seen in Figure X.

Both the average and maximum kernel executions times are related the number of LRTs with a second order polynomial. This can be seen in Figure X.

The average time taken to process traders per tick remains related to the number of LRTs with a second order polynomial. This is expected as the relationship between the number RTs and average trader process time was also related with a second order polynomial; LRTs are in effect RTs but with larger constant multipliers used to determine the volume of a trade. (see Effect of Random Trader). Once again the maximum time remains roughly constant around a value of 5.6385 seconds, although the standard deviations for the maximum values are again large. The relationship can be seen in Figure X.

#### Effect of Position Trader

An increase in the number of PTs results in an increase in the time taken to complete one simulation that is related with a second order polynomial. This can be seen in Figure X.

The average matching time is once again linearly related to the number of PTs (see 5.3.1.1). This can be seen in Figure X.

The relationship between the average or maximum kernel execution times and the number of PTs is a second order polynomial. This can be seen in Figure X.

Finally the average trader time is related to the number of PTs with a second order polynomial, whilst the maximum trader time remains roughly constant around a value of 5.745 seconds. This can be seen in Figure X.

#### Effect of Momentum Trader

Increasing the number MTs increases the average amount of time taken to complete a simulation with a second order polynomial. This can be seen in Figure X.

Once again the average matching time increases linearly with an increasing number of traders – in this case MTs. This can be seen in Figure X.

Again both the average and the maximum kernel execution times increase with an increasing number of MTs having a second order polynomial relationship. This can be seen in Figure X.

Finally the average trader time increases with increasing numbers of MTs with a second order polynomial relationship. This account for the relationship between total time taken and number of MTs. The maximum trader time remains roughly constant with a value of 5.7802 seconds. This can be seen in Figure X.

#### Effect of Total Number of Traders

Given that the increase in total time taken for a simulation with an increase in any of the four trader types was related with a second order polynomial (see 5.3.1.1-5.3.1.4) we would expect that when the total number of traders increases with an equal number of each that this would increase the time taken with a similar relationship to that observed for each of the traders. This can be seen to be the case in Figure X. Additionally even when there are no traders of any kind the total time taken is not zero. Instead the average time for no traders is 2,015.404ms which is the overhead time for running a 60,000ms simulation. Thus the maximum speed up achieved over real time is just under 30x.

The average matching time increases linearly with increasing numbers of traders as is to be expected given the results in 5.3.1.1-5.3.1.4. This can be seen in Figure X. It is worth noting that unlike in the results for each of the trader types individually there are larger margins for the values at 84 and 96 of each trader, however the average matching time is well below one second. Indeed using the formula observed in Figure X, it would not be until there were 2,500 of each trader type (a total of 10,000 traders) that the average matching time would exceed one second. However there are other bottlenecks preventing such a large number of traders being reached.

The average kernel time again increases with a second order polynomial relationship to the number of traders, however the relationship is tentative as several points do not lie on the curve, which does not even fall within their error margin. The maximum execution is related similarly however it is even more erratic than the average execution time, albeit following the same pattern. This can be seen in Figure X.

The average trader time increases parabolically with the total number of traders as would be expected given the parabolic results in 5.3.1.1-5.3.1.4. Additionally the max time is increasing linearly. This indicates that the maximum trader process time may be increasing linearly for one or more of the trader types discussed previously in 5.3.1.1-5.3.1.4. The average and maximum trader times can be seen in Figure X.

#### Effect of Trader Time Range

As the average trader T decreases the time taken increases as a power of the average trader T. For low average trader T’s (of the order of high frequency trading – 1-10ms) the amount of time taken becomes incredibly large – up to 11 times longer than real time. This is expected as for low average trader T’s order will be placed up to every tick of simulation. Whether this is caused by an increase in matching time, kernel execution time or trader process time will be explored in the next graphs. First however the relationship between average trader T and time taken can be seen in Figure X.

At first inspection it would appear that the average matching time is not the source of the bottleneck producing such large increases in total time taken against average trader T as the average match time remains relatively low, below 5ms for an average trader T of 16.544 (see Appendix for data). However the maximum match time becomes incredibly large (with large error margins) taking up to 7818.369ms for average trader T of 16.544. Given that real time execution requires the matching time to be less than or equal to 1ms this is an extremely bad worst case scenario for a matching time. This can be seen in Figure X.

The average kernel time is also related to the average trader T with a power relationship; however its gradient is less steep than that of the average match time or the average trader process time. The maximum graph behaves similarly to the average graph, however with a steeper gradient. It does not reach a maximum value anywhere as near as high as the maximum value reached by the maximum kernel time, and the average kernel time at average trader T equal to 16.544 is less than that of the average matching time. This can be seen in Figure X. The relationship can be attributed to the increase in number of past prices caused by having more trades.

The average trader time is in general less than the equivalent average matching time indicating that is likely that the matching process is the bottleneck. The average and maximum trader process times and related to the average trader T with a power relationship which can be seen in Figure X.

### Financial Analysis

#### Effects of Random Trader

Increasing the number of RTs increases the volatility per minute of the simulation. However volatility remains relatively low – less than 5% with up to 96 RTs. It is also worth noting that the volatility of these simulations put the annual volatility extremely high (of the order of 100s or 1000s % volatility). See Appendix for these calculations. The effect of the number of RTs on the volatility per min can be seen in Figure X. The increase in volatility against number of traders is decreasing with number of traders. It would be necessary to run simulations with considerably larger number of RTs to see if the curve tends to a particular value of volatility or begins to decrease with a certain number of traders, however it is much more likely that the curve tends towards a value. This is because increasing the number of RTs should increase the number of orders with random volumes and prices per minute resulting in more opportunity for the price to fluctuate and thus for volatility to be higher. Power relationship?

As the number of RTs increases the average spread (the difference between the lowest sell price and highest buy price) increases, however decreasingly so. Originally the average spread was expected to decrease with the number of RTs as increasing the number of traders should increase the number of orders thus increasing the change of being able to match any given order (since there are more outstanding orders on the book). Instead the average spread decreases, although it appears to be a tending to a value. This can be attributed to the fact that as the number of RTs increases the number of ticks with outstanding orders increases thus increasing the average spread. The relationship can be seen in Figure X. A higher average spread indicates a lower liquidity and thus liquidity is decreasing with the number of RTs added to the model.

The other measure of liquidity used in this report is Trades per Second. The number of trades per second increases linearly with the number of RTs. This is to be expected as more RTs results in more orders being placed and therefore and increase in the change of a trade being completed. This can be seen in Figure X.

Finally as the number of RTs increase we can see that the average profit per minute of a RT tends to a constant value of $42,023. In addition to this the maximum and minimum profits form a mirror image of each other around the constant average value. This can be seen in Figure X. The effects of increasing the number of RTs on the other traders profitability can be seen in the Appendix.

#### Effects of Large Random Trader

The volatility per minute is related to the number of LRTs logarithmically/power. This can be seen below in Figure X. Notably adding LRTs does not increase the volatility per minute nearly as much as adding RTs (see 5.3.2.1). This is because the LRTs tend to wipe out large numbers of outstanding orders that are on the book (indeed they are designed to do this (see implementation section)) in a single tick resulting in only one price movement for a greater or equal to one number of orders matched. For example a trader placing small buy orders against a large stack of sell orders would alter the price every tick they placed a small buy order until the stack of sell orders were matched. This would result in a change in price each tick and therefore an increase in volatility as the price is changing more often.

The number of LRTs has a less damaging effect on average spread than the number RTs and it is still related logarithmically. This can be seen in Figure X.

Despite decreasing the average spread measure of liquidity increasing the number of LRTs increases the number of trades per second; most notably adding just one LRT drastically increases the average number of trades per second from just below 8 to over 12. For one LRT and greater the number of trades per second is related to the number LRTs linearly. This can be seen in Figure X.

Unlike the average profit of RTs which tends towards a constant value, the average profit of LRTs has a power relationship with the number of LRTs. This is expected as the fewer LRTs there are the more opportunity there is for a single one to profit from wiping out all opposing trades on the book. Additionally the maximum and minimum profit values tend towards the average logarithmically with increasing numbers of LRTs. This is the opposite behaviour as that seen for RTs (see sections 5.3.2.1) in which the maximum and minimum are diverging – here there are converging. This can be seen in Figure X. The error margins for the maximum and minimum values are extremely large (of the order of 10^9 and have thus been left off the graph in order that it may be displayed. The error values can be found in the Appendix. The effects of increasing the number of LRTs on the other traders profitability can be seen in the Appendix.

#### Effects of Position Trader

Unlike the effect of increasing the number of RTs or LRTs, increasing the number of PTs results in a linear increase in the volatility per minute, although it is difficult to say this with certainty due to the error margins, which can be seen in Figure X.

Given the error margins it is possible that increasing the number of PTs does not increase the volatility at all or if it does, by very little. This is expected as the PTs are designed to emulate market making activity. This means that they are designed to increase the liquidity of the market by placing orders on either side of the order book, whilst trying to remain with as neutral a position as possible. Thus we would not expect them to increase the volatility of the market by much (if at all), but we would expect them to increase the liquidity of the market. The increase in liquidity that they provide can be seen in both Figure X and Figure X. The average spread decreases logarithmically with an increasing number of PTs and the number of trades per second increases linearly as has been the case with any increasing number of traders (see 5.3.2.1, 5.3.2.2).

The average PT profit is always negative, however is increasing logarithmically with increasing numbers of traders. The cause of such large losses may be that the PT is designed to run over longer periods of time than one minute of simulation time; given that an order is placed on average every 3,000ms, the PTs will on average be placing only 20 orders per simulation. This does not give the PTs sufficient time to close out their positions and return to being neutral. This can be tested by measuring PT average profit against the duration of simulation. Additionally the maximum profit is also increasing logarithmically with increasing numbers of PTs, whilst the minimum profit is decreasing more steeply than the maximum profit is increasing. This can be seen in Figure X. The effects of increasing the number of PTs on the other traders profitability can be seen in the Appendix.

#### Effects of Momentum Trader

Increasing the number of MTs increases the volatility per minute logarithmically/linearly/geometrically/power. However, it is difficult to state this with certainty due to the error margins which can be seen in Figure X. The increase in volatility with an increase in number of MTs is very small and is perhaps the smallest increase in volatility with an increase in the number of specific type of trader.

Average Spread decreases logarithmically as the number of MTs increases. This is to be expected as the MTs are designed to observe ‘momentum’ in price movements (see section about MT design) and trade accordingly. For example when an MT observes that the price has been falling for some time it will place buy orders as it expects the price to go up soon. MTs place order with a price close to the last price thus we would not expect them to increase the average spread. It is worth noting that although average spread is decreasing with increasing MTs, it is not decreasing nearly as much as it did with increasing PTs. This is to be expected. It is also worth noting that the error margins for this relationship are much larger than they were for the equivalent relationship for PTs. The relationship between volatility per minute and the number of MTs can be seen in Figure X.

Additionally the number of trades per second once again increases linearly with increasing numbers of traders. This can be seen in Figure X. Thus on both measures of liquidity MTs improve the liquidity of the market.

The average profit of a MT remains roughly constant around a negative value of -$130,595, however increasing the number of MTs increases the maximum MT profit parabolically reaching up to over $100,000 of profit. The error margins for the average maximum profit are quite large, however for 96 MTs the whole error bar is positive. Additionally increasing the number of MTs does decrease the average minimum profit, however the decrease is not as large as the increase in the average maximum. The average minimum profit is related logarithmically to the number of MTs. This can all be seen in Figure X. The effects of increasing the number of MTs on the other traders profitability can be seen in the Appendix.

#### Effects of Total Number of Traders

The volatility per minute has a power relationship with the number of traders. This is the combination of effects from the linear increase provided PTs, the linear/power increase from MTs, the parabolic/power increase from RTs and the logarithmic/power increase from LRTs. This can be seen in Figure X.

Given that the relationship between average spread and the number of each of the types of traders is either logarithmic, or inversely logarithmic, it is expected that the relationship between the average spread and the total number of traders is logarithmic. This can be seen to be the case in Figure X, whilst the number of trades per second remains linearly related to number of traders as can be seen in Figure X.

Minimum, Average and Maximum RT profit behave as they did when only the number of RTs was being increased. Similarly the average LRT profit behaves as it did when only the number of LRTs was being increased, however the minimum and maximum LRT profit behave considerably more erratically and have even more colossal error margins indicating large variations in the min/max LRT profit. The data for this can be seen in the Appendix. Minimum, average and maximum PT profit also behave as they did when only the number of PTs was being increased, however this is not the case for MTs. When increasing the number of all types of traders equally the maximum MT profit no longer increases parabolically; instead it follows a logarithmic pattern increasing slowly. The average MT profit no longer stays around a constant value, but initially decreases before beginning to increase again. It thus appears to be increasing parabolically, albeit very slowly. Finally the minimum MT profit is no longer inversely logarithmic, but is instead increasing parabolically. These results can be attributed to the fact that when there are larger numbers of all the types of traders in the market, the number of trades and therefore prices movements will increase. Since the MTs work by observing price momentum over a set number of past price changes (see section something) they will thus see more ‘momentum windows’ which will cause them to place orders and make profit. Note, however that despite the change in relationships the minimum, average and maximum MT profit remain negative. See Figure X.

#### Effects of Trader Time Range

Decreasing the average trader time (T) increases the volatility per minute with a power relationship up to a point. This can be seen in Figure Xa and Figure Xb, however between an average trader T of 10ms and 100ms the volatility per minute begins to decrease. This can be seen in Figure Xc. This average trader T range needs further investigation to determine whether the volatility does actually decrease as this point and to confirm that the present data is not anomalous. Unfortunately it can take up to a week to run 500 simulations with average trader Ts in this range, so there has not been time to run these simulations. It is also worth noting that the coefficient in the equation varies considerably between Figures Xa and Xb; this can be attributed to the large error margins present for the points with the lowest average trader T, which are increasing the gradient of the graph. The increase in volatility up to average trader Ts of less than 100 is expected as we would expect that more trades would result in more price movements, resulting in higher volatility. The potential decrease in volatility can perhaps be attributed to the decrease in number of trades per second which will be discussed later in this section.

Similarly to the relationship between volatility and average trader T, the relationship between the average spread and average trader T behaves with a power relationship up until an average trader T of less than 100. After this point the average spread decreases to become negative indicating good liquidity. This can be seen in Figure X. This would seem to indicate that high frequency trading does in fact improve liquidity considerably, however as with volatility additional data points in this average trader T would be useful to confirm that the average spread does indeed go negative.

The number of trades per second increases in a log-log manner with the average trader T up until average trader T reaches between 10 and 100ms. At this point it begins to trail off potentially down to a fixed value. Unlike the comparisons between volatility per second or average spread and average trader T, the error margins are small for this graph indicating that this behaviour may be as observed. This can be seen in Figure X.

Indeed comparing the average spread to the number of trades per second indicates that when there are over 600 trades per second the average spread begins to decrease very steeply. Before this the average spread increases linearly as the number of trades per second increases. This can be seen in Figure X. This can be explained as trades occur when there are orders with a less than or equal to zero spread between them (or when there is market order). Whenever there are no orders that can be matched the average spread increases, whilst when there are orders that cannot be matched the average spread decreases and can go negative. Thus when there are insufficient order matches (trades) per second the average spread should remain approximately constant and perhaps decrease slightly. This can be approximately seen in the linear section of the graph in Figure X. As the number of orders placed per second increases the chance of a trade occurring increases. There will be a threshold value of the number of orders being placed after which trades will be executed more frequently than not. In this model we would expect this value to be 500 trades per second as there are 1,000 ticks per second. This can be seen on the graph as the horizontal error margin for the point just before the drop off includes 500 trades per second. Given more time to run more simulations at the high average trader Ts necessary to achieve this number of trades per second this could be investigated further.

The average and maximum RT profit increase with decreasing average trader T, with a power relationship; whilst the minimum RT profit decreases logarithmically with decreasing average trader T. This is expected as when the frequency with which a RT places orders increases, its chance to participate in a trade increases and thus the opportunity for profit (or loss increases). This explains the divergence of minimum and maximum profit from one another. This can be seen in Figure X. The values for average trader T approximately equal to 16ms have been left off due to large error margins.

The minimum, average and maximum LRT profit behaves similarly to RT profit as expected. However the minimum decreases more drastically than it does for RTs. Additionally the first three data points have been left off the graph due to colossal error margins, as well as all the other error margins being left off for similar reasons. The actual values can be found in the Appendix. The data points can be seen in Figure X.

Minimum average and maximum PT profit all increase logarithmically with decreasing average trader T up to average trader T of around 700ms. After this point they all tend towards approximately constant or slightly increasing values. This is consistent with the idea that the profitability of PTs would increase given a longer simulation period (see Section something) as increasing the frequency with which they trade is equivalent to increasing the duration of the simulation. This indicates that the PTs are behaving approximately as designed (see section about design of PTs) whereby they attempt to maintain a neutral position and therefore neutral profit whilst adding liquidity to the market. The graphs can be seen in Figure X. Again three of the first points have been left off due to colossal error margins.

Minimum, average and maximum MT profit all decrease with decreasing average trader T. This is not expected as we expect MTs to become more profitable when there are more price movements. However it can be explained due to the fact that the number of past prices passed to each MT is limited to 4,096 due to an OpenCL out of resources exception if more were passed. The minimum and maximum diverge with decreasing average trader T. This can be seen in Figure X.

## Model 2

### Performance Analysis

#### Effect of High Frequency Position Trader

#### Effect of High Frequency Momentum Trader

### Financial Analysis

#### Effect of High Frequency Position Trader

#### Effect of High Frequency Momentum Trader

#### Effect of Equal Numbers of High Frequency Position and Momentum Traders

#### Effect of Introducing Limit on Orders Per Unit Time

## Conclusions of Analysis

### Performance

### Financial

# Future Work

# Conclusion

# Bibliography

# Appendix (Stored in a separate file currently)

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)