Working with OpenCL to Speed Up a Genetic Programming Financial Forecasting Algorithm: Initial Results

James Brookhouse
School of Computing
University of Kent
Canterbury, UK
jb733@kent.ac.uk

Fernando E. B. Otero
School of Computing
University of Kent
Chatham Maritime, UK
F.E.B.Otero@kent.ac.uk

Michael Kampouridis
School of Computing
University of Kent
Chatham Maritime, UK
M.Kampouridis@kent.ac.uk

ABSTRACT

The genetic programming tool EDDIE has been shown to be a successful financial forecasting tool, however it has suffered from an increase in execution time as new features have been added. Speed is an important aspect in financial problems, especially in the field of algorithmic trading, where a delay in taking a decision could cost millions. To offset this performance loss, EDDIE has been modified to take advantage of multi-core CPUs and dedicated GPUs. This has been achieved by modifying the candidate solution evaluation to use an OpenCL kernel, allowing the parallel evaluation of solutions. Our computational results have shown improvements in the running time of EDDIE when the evaluation was delegated to the OpenCL kernel running on a multi-core CPU, with speed ups up to 21 times faster than the original EDDIE algorithm. While most previous works in the literature reported significantly improvements in performance when running an OpenCL kernel on a GPU device, we did not observe this in our results. Further investigation revealed that memory copying overheads and branching code in the kernel are potentially causes of the (under-)performance of the OpenCL kernel when running on the GPU device.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms

Algorithms

Keywords

Genetic Programming, OpenCL, financial forecasting, GPU

1. INTRODUCTION

EDDIE is a Genetic Programming algorithm that makes predictions about potential opportunities in the stock market. Recently additional features have been added to EDDIE, which significantly further improved its predictive performance. However, these features also led to a greatly increased execution time [13, 11], making the algorithm impractical to run in big scale experiments.

Speed can be an important aspect in financial trading. For example, in the field of algorithmic trading, the trading orders take place electronically, by using pre-programmed trading strategies. Therefore, slow algorithms can have devastating effects on a traders' profitability, as other, faster, algorithms might place an important order first, and thus attract all financial gain.

Our aim in this paper to develop an EDDIE implementation that can take advantage of multi-core CPUs and GPUs providing EDDIE with a substantial performance increase. A number of routes could be taken to improve the execution time of EDDIE. GPUs have already been used to perform general purpose computing. Additionally, they have also been successfully used in the area of evolutionary algorithms. A number of different architectures exist for GPGPU including NVIDIA's CUDA and OpenCL. OpenCL has an advantage of working over a number of platforms including both CPUs and GPUs. For the purposes of this work, OpenCL was chosen, as it provides a platform agnostic view through a hardware abstraction, allowing programs to be developed that can run on different hardware with minimal changes.

The paper is structured in the following way: Section II will present the genetic programming tool EDDIE including a brief history. Section III will summarise relevant works that have been carried out in the field of GPUs. Section IV provides a description of the OpenCL implementation added to EDDIE. Finally in sections V and VI we will present our results and conclusions.

2. EDDIE

2.1 Description of the Algorithm

EDDIE is a Genetic Programming (GP) [14, 18] financial forecasting algorithm, which learns and extracts knowledge from a set of data. The question EDDIE tries to answer is 'will the price increase within the n following days by $r\%$?' The user first feeds the system with a set of past data; EDDIE then uses this data and through a GP process, it creates and evolves Genetic Decision Trees (GDTs), which make recommendations of buy (1) or not-to-buy (0).

The set of data used is composed of three parts: (i) daily closing price of a stock, (ii) a number of attributes, and (iii) signals. Stocks' daily closing prices can be obtained online on websites such as `http://finance.yahoo.com` and also from financial statistics databases like *Datastream*.[1] The attributes are indicators commonly

[1]Available at: `http://thomsonreuters.com/datastream-professional/`

<Tree> ::= `If-Then-Else` <Condition> <Tree> <Tree> | Decision
<Condition> ::= <Condition> `AND` <Condition> |
    <Condition> `OR` <Condition> |
    `NOT` <Condition> |
    <VarConstructor> <RelationOperation> Threshold
<VarConstructor> ::= MA period | TBR period | FLR period | Vol period
    | M period | MMA period
<RelationOperation> ::= ">" | "<" | "="
Terminals:
    MA, TBR, FLR, Vol, Mom, MomMA are function symbols
    Period is an integer within a parameterized range, [MinP, MaxP]
    Decision is an integer, Positive or Negative implemented
    Threshold is a real number

Figure 1: The Backus Normal Form of EDDIE

used in technical analysis [6]; which indicators to use depends on the user and his belief of their relevance to the prediction. The technical indicators that are used in this work are: Moving Average (MA), Trade Break Out (TBR), Filter (FLR), Volatility (Vol), Momentum (M), and Momentum Moving Average (MMA).[2]

The signals are calculated by looking ahead of the closing price for a time horizon of *n* days, trying to detect if there is an increase of the price by *r%* [20]. In other words, the GP is trying to use some of the above indicators to forecast whether the daily closing price is going to increase by *r%* within the following *n* days.

After feeding the data to the system, EDDIE creates and evolves a population of GDTs. Figure 1 presents the Backus Normal Form (BNF) [4] (grammar) of EDDIE. As we can see, the root of the tree is an `If-Then-Else` statement. The first branch is either a boolean (testing whether a technical indicator is greater than/less than/equal to a value), or a logic operator (`AND`, `OR`, `NOT`), which can hold multiple boolean conditions. The `Then` and `Else` branches can be a new GDT, or a decision, to BUY or NOT-BUY (denoted by `1` and `0`, respectively).

A sample GDT of EDDIE is presented in Figure 2. As we can see, if the *2 days MMA* is greater than *127.994* and the *17 days M* is less than *152.158*, then the user is advised to not-buy; otherwise, the user is advised to buy.

Depending on the classification of the predictions, we can have four cases: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). As a result, we can use the metrics presented in Equations 1, 2 and 3:

Rate of Correctness

$$RC = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Rate of Missing Chances

$$RMC = \frac{FN}{FN + TP} \tag{2}$$

Rate of Failure

$$RF = \frac{FP}{FP + TP} \tag{3}$$

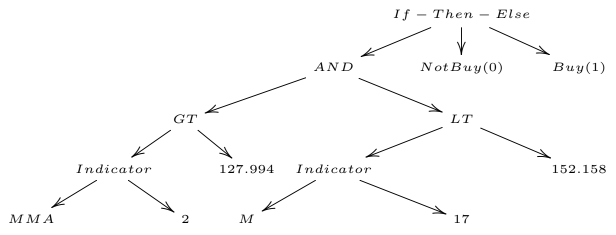


Figure 2: A sample EDDIE Tree generated by executing EDDIE, when it is executed the first branch of the If-Then-Else statement, if it evaluates true then the first branch is taken and it then follows that a false evaluation will take the second branch.

The above metrics combined give the following fitness function, presented in Equation 4:

$$ff = w_1 * RC - w_2 * RMC - w_3 * RF \tag{4}$$

where w_1, w_2 and w_3 are the weights for RC, RMC and RF respectively. These weights are given in order to reflect the preferences of investors. For instance, a conservative investor would want to avoid failure; thus a higher weight for RF should be used.

2.2 Brief History

EDDIE was originally created from a horse prediction algorithm, which was then adapted to predict the stock markets [20]. One of the latest EDDIE versions is EDDIE 8. This version is the one presented in the previous section. The advantage of this version is that the algorithm is not constrained to pre-specified periods, as is usually the case in industry.[3] As a consequence, it is up to the GP and the evolutionary process to look for the optimal periods values from the period range provided. For instance, if this range is 2 to 65 days, then EDDIE can create Moving Averages with any of these periods, e.g., 20 days MA, 25 days MA, and so on.

However, while the above approach returned positive predictive results [13, 12], it also dramatically increased the search space of EDDIE. This then led the algorithm to occasionally miss good solutions, due to ineffective search. To address this issue, local search algorithms were applied to the period leaf nodes of the trees, by means of hyper-heuristic frameworks [10, 11]. These frameworks used a number of different hill climbers and mutators at every generation, and made marginal changes to the period leaf nodes of the trees of the population. As a result, the search became more effective, as more exploitation was taking place.

While the fitness of the trees produced by EDDIE further increased thanks to the hyper-heuristics, the execution time also increased dramatically. Initial EDDIE 8 versions would have average single run times of around 3 minutes. With the introduction of hyper-heuristics, however, this increased to 12 minutes per run [11].

[2]We use these indicators because they have been proved to be quite useful in developing GDTs in previous works like [16], [1] and [3]. Of course, there is no reason not to use other information like fundamentals or limit order book. However, the aim of this work is not to find the ultimate indicators for financial forecasting.

[3]In the literature, the users of similar algorithms pre-specify certain periods that they consider useful. For instance, *20 days* MA, and *50 days* MA. The indicators (e.g., MA) together with their respective period (e.g., 20) are treated by the GP as a single leaf node. Thus, the numbers 20 and 50 cannot change during the evolutionary process. In our previous work [13, 12], we questioned this approach, because nobody can guarantee that, for instance, a 20 days MA is better than a 25 days MA. To address this issue, we created EDDIE, which is able to search in the space of technical indicators and their periods.

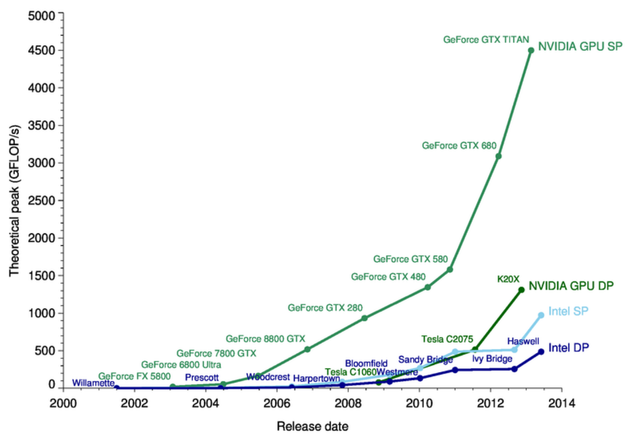


Figure 3: Comparison of NVIDIA graphics cards and Intel x86 CPUs in single and double precision floating point operations. Figure is reproduced from: http://hemprasad.wordpress.com/2013/07/18/cpu-vs-gpu-performance/.

As we can observe, the addition of the hyper heuristic framework slowed EDDIE down significantly. This was because of the increased number of fitness evaluations that were taking place. Effectively, every time a marginal change would take place in a leaf node of a tree, the fitness would have to be re-calculated and compared to the old one. Since this was taking place multiple times per generation (in addition to the typical GP fitness evaluations) the execution time was affected, slowing down the algorithm. This thus led us to consider ways of parallelising the process, especially since the additional computational times were due to the same process (i.e., fitness evaluation) happening multiple times.

3. RELATED WORK

The performance of GPUs has massively increased in recent years. Floating point performance of GPUs is now an order of magnitude higher than those found in high-end CPUs. Figure 3 shows that while CPU performance has increased over the last 10 years to 150 GFlops/second, GPUs now operate at over 4000 GFlops/second, illustrating how powerful GPUs have become when compared to CPUs.

This performance has been utilised in many computational intelligence applications, including neural networks and ant colony optimisation algorithms [15]. Delevacq et al [5] created a solution that used GPUs to speed up an ant colony optimisation algorithm. This was accomplished by deploying parallel ants in two separate modes: one where each ant has its own stream processor; the second gives each ant a block (group of stream processors). The first strategy has the advantage of allowing a huge number of ants to be evaluated at the same time. The second strategy allows the evaluation of each ant to take advantage of parallelism in finding the optimum path. Speed ups of between 5 and 15 times are observed, with a greater improvement seen when using the second strategy [5].

Moving on to parallelism in genetic programming application, Langdon [15] describes a method of using stacks and RPN (Reverse Polish Notation) to create a stack based SIMD (Single Instruction, Multiple Data) interpreter, which will leave the results on the top of the stack at the end. This method does require the tree to be stored in an appropriate format or converted to one [15].

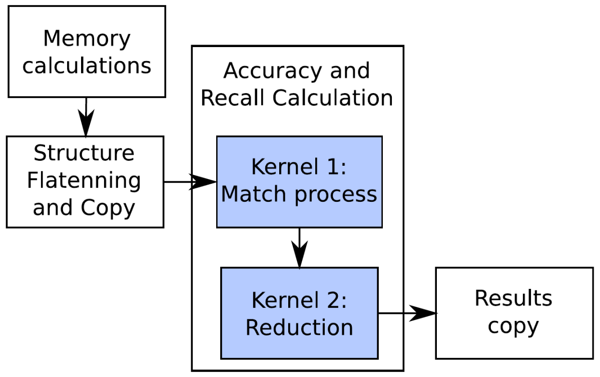


Figure 4: The process involves flattening and copying data from host to GPU, then the evaluation can be performed. The second kernel is required to reduce the amount of data returned from the kernel. Finally the data is copied back to the host platform. Figure is reproduced from [7].

An alternative approach to an interpretor is to compile each tree, since historically interpreter based platforms in computer science are regarded to be slower than compiled solutions. This is due to the benefits gained when the compiled code is run many times. However, as the programs created by a Genetic Programming tool are only evaluated a few times, interpreted based methods are usually faster due to compilation overhead [15].

Franco et al. [7] have implemented a GPU based fitness evaluation for the BioHEL evolutionary learning system. The implementation uses the CUDA platform with a two stage kernel solution. Figure 4 shows the stages of the fitness evaluation [7]. First, memory requirements are calculated to ensure all data will fit on the GPU or if multiple evaluations are required. The structure is then flattened to become compatible with GPUs. Flattening is the process of converting data structures from a many dimensional structure such as a tree or 2+ dimensional array into a one dimensional array. This is important as GPUs will only accept one dimensional arrays as kernel arguments. The data is then copied on to the GPU, where two kernels are executed: the first is to calculate the fitness function; the second is to reduce the amount of data that is read back from the GPU. The results are then read back to the host platform.

Franco et al. [7] achieved very large speed ups in their implementation, with a maximum speed up of 52.4x when compared to the original speed ups. This was achieved when the testing data was between 25,000 and 50,000. It is hypothesized that for values lower than 25,000, the serial method can keep all of the data in its fast cache, while the peak in performance and drop off over 50,000 may be due to the transfer of larger amounts of data.

It was also found that experiments that contained more decisions were slowed down compared to those that were linear. This is to be expected as the SIMD architecture of GPUs is heavily penalised by divergent code, where many threads may be idling when the other branches are being evaluated [7].

Data transfer times are a key overhead that occurs when attempting to use GPUs. This has also been discussed by Franco et al. [7], who executed an additional kernel that would reduce the data that was sent back to the host device. This is a known issue and as such, NVIDIA have released plans for reduction algorithms that have a higher bandwidth than the PCI Express bus, allowing a more efficient kernel to be utilised [8].

While early attempts to use GPUs mainly used NVIDIA's CUDA platform, successful implementations using the OpenCL platform do exist. Augusto and Barbosa achieved this with GPOCL [2]. The implementation was built from scratch with the intention to use OpenCL. This enabled the authors to implement a linear prefix stack based structure which is inherently suitable to evaluation using OpenCL. This allows the flattening stage seen in [7] to be avoided; instead, an interpretor-based kernel could then be used to evaluate the genetic programming programs. GPOCL achieved impressive results, where a 12-core Intel Xeon can evaluate 1.590 Billion GPop/s while a Nvidia GTX-285 achieved 10.75 billion GPop/s, representing a speed up of over 5 times.

Augusto and Barbosa [2] investigated optimisation techniques that can be applied to OpenCL programs. These include various memory optimisations, such as the use of constant memory where the GPU can detect and optimise multiple reads to the same memory location. Local memory usage is very fast and can be shared across all process elements. GPOCL caches genetic programming programs prior to execution and performs a reduction inside the local memory before copying the result back to global memory. Native function usage can also increase performance. OpenCL provides a number of mathematical functions in three flavours, IEEE-754 float, IEEE-754 half float and native. While the native functions can violate the IEEE standard, they do provide increased performance [2].

The previous literature identifies the fitness calculation performed by genetic programming algorithms as the most computationally expensive. This operation requires many calculations on different data independently,. For the reasons listed above a parallel approach becomes obvious. For these reasons it is the first part that is normally implemented on a GPU [15].

Pusniakowski and Bednarczyk [19] proposed an implementation that executes the entire genetic algorithm on the GPU using the OpenCL platform. The algorithm has been split into a number of different kernels, which are then called in the appropriate order. The first kernel called will create the initial population. As has been discussed earlier, the fitness function will be calculated in a second kernel. This kernel uses private memory so it can take advantage of the increased performance of this fast memory.

One difference between this implementation than those discussed previously is the parallelism of the genetic operations. The authors implemented two algorithms: the roulette wheel which is a fitness weighted method; and tournament selection, which takes n individuals (where n is equal to two or greater) and uses the fitness to determine the winner. While both implementations were effective and gave reasonable gains, it was found that the roulette wheel operation was much less efficient, taking considerably longer to produce very fit populations.

4. OPENCL IMPLEMENTATION

Previous work identifies that the biggest gains are found when using an OpenCL or CUDA platform to calculate the fitness function, since many solutions' fitnesses can be calculated in parallel. Given that EDDIE calls the fitness function many times, an straightforward OpenCL-enabled implementation will simply replace these multiple calls by a singe call to the OpenCL kernel.

The OpenCL enhancement in EDDIE can be split into a sequence of steps: the first is the process of flattening of trees from an object-orientated tree to a flatter GPU friendly structure; the second step is the OpenCL environment preparation; and the third is the fitness evaluation (kernel) that needs to be queued ready for execution on the OpenCL device. The OpenCL kernel can then be executed before the results are read back from the OpenCL device and analysed to produce a fitness for each tree.

4.1 Tree Flattening Into Byte Arrays

Before trees can be executed on the OpenCL device, they need to be prepared into a data structure that can be used in an OpenCL environment. OpenCL can only use a limited set of data structures as inputs due to its ability to run on a number of different hardware architectures. The flattening process must be quick to ensure that the speed improvements gained by executing the fitness function on the device are not wasted at this stage.

The Java API provides a `ByteBuffer` data structure in the `java.nio` package [17]. This data structure allows the construction of arrays containing multiple data types, as longer primitives are split into bytes and their bit representations are then written to the array. This also allows the programmer to retrieve the values stored in the array as long as they know the underlying data structure. The advantage of a byte buffer is the decreased size of the flattened data structure. This decreases the overall memory size required to store the GP population, which in turn decreases the time taken to transfer the population to the OpenCL device.

The tree-based structure used by EDDIE can be split into nodes, which are based around a boolean test. The test takes two arguments, an indicator and a constant. The result of this test will then determine the next action to be taken. In the array structure this will involve jumping to a new location (a new index in the array) in the `ByteBuffer`. Jump can either be positive, when the test evaluates to true, and negative, when the test evaluates to false.

The general node layout is shown in Table 1. All jumps in the byte array are absolute jumps expressed in the number of bytes since the beginning of the array. Figure 5 shows an example EDDIE tree and the corresponding nodes stored in the byte array (each line of the table in Figure 5 corresponds to a node). The total size of the array for this tree is 48 bytes (4 times the size of a single node), where the size using a float array would have been 96 bytes. Therefore, the trees produced using a byte array are half the size of the float array. Another advantage of using a byte array is that integer constants do not have to be converted to floats and then back again to integers in the OpenCL kernel.

4.1.1 NOT operator

An interesting problem found when flattening trees was the NOT operator. The first idea to deal with this involved flipping the positive / negative jumps of the associated operators. This 'jump flipping' ensured that a true test evaluation resolves to the false branch while a false evaluation will jump to the positive branch, mimicking the flip created by a NOT operator. While this worked for most cases, it produced incorrect trees in cases involving nested NOT operators.

A solution was finally found when NOTs were 'pushed' down the tree until they reached a comparison function. If an AND or OR function is encountered, the rules of logical equivalence found in Equations 5 and 6 allow the NOT operator to be pushed through the function descending down the tree further. This solution successfully works for all trees that contain the NOT operator.

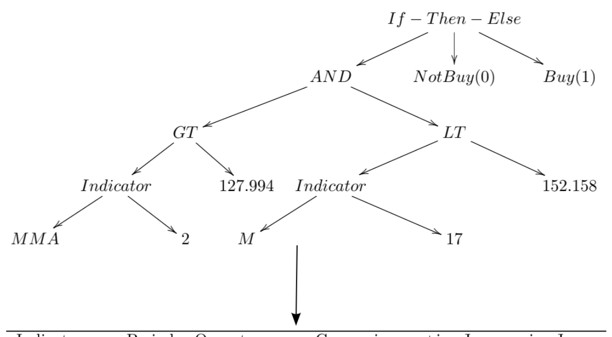
$$\neg(A \vee B) = \neg A \wedge \neg B \tag{5}$$

$$\neg(A \wedge B) = \neg A \vee \neg B \tag{6}$$

When the tree reaches a comparison operator they are changed from 'Less Than' to 'Greater Than or Equal To'; 'Greater Than'

Table 1: The general node structure along with brief description of each element used to convert trees into byte arrays. Each node takes a total of 12 bytes.

Type	Name	Description
short	- Indicator	Which Indicator to use
byte	- Period	Period of the chosen indicator
byte	- Operator	What Operation to perform
float	- Comparison	Comparison value for the operator
short	- Positive Jump	Where to go if operator evaluates true
short	- Negative Jump	Where to go if operator evaluates false
Total Size - 12 Bytes		



Indicator	Period	Operator	Comparison	+ive Jump	-ive Jump
MMA	2	Greater Than	127.994	12	36
M	17	Less Than	152.158	24	36
Not Buy (0)	-	-	-	-	-
Buy (1)	-	-	-	-	-

Figure 5: The representation of an EDDIE tree as nodes using the structure presented in Table 1.

becomes 'Less Than or Equal To.' Finally the third test 'Equal To' becomes 'Not Equal To.'; thus completing the NOT chain.

4.2 JOCL

Java OpenCL (JOCL) [9] is an open-source project which provides OpenCL bindings for Java. Since EDDIE was originally written in Java, the use of JOCL avoids the requirement to pass program parameters to a native program, which would require the use of either the Java Native Interface (JNI) or the Java Native Access (JNA). Therefore, to implement the OpenCL-enabled fitness evaluation in EDDIE we used the JOCL library - this extended EDDIE version is called EDDIE-JOCL.

There are two parts that makes up the OpenCL/JOCL implementation of any application: the host program and a kernel.

The host program creates the OpenCL device and deals with the scheduling and execution of kernel instances. It provides the bridge between the two architectures. The general process requires the selection and initialisation of an OpenCL device. Once an OpenCL device has been created you can then specify the data that is to be sent across and queue work items to be processed. After execution, data can then be retrieved from the OpenCL device.

The OpenCL kernel contains the code that will be executed on an OpenCL enabled device (CPU or GPU device). It generally consists of the internals of a *for loop*, which has been unravelled so that it can be executed in parallel. Each work unit on the OpenCL device has its own `id` which can be obtained with the following snippet of code:

```
int gid = get_global_id(0);
```

This allows each work unit to identify which bit of data it should be using in its execution. In EDDIE, this parameter is used to identify which day of the training or test data we want to evaluate.

Inside EDDIE's kernel, we have a struct that is overlaid onto the incoming tree to convert the bytes back into their original types. The struct displayed below corresponds to the OpenCL version of the same node structure in Table 1:

```
// node structure used to flatten
// the trees in EDDIE-JOCL
struct node {
    short indicator;
    char period;
    char operator;
    float comparison;
    short pos_jump;
    short neg_jump;
};
```

First the kernel places this node struct over the start of the byte array received by the kernel. It then evaluates the node and decides whether it should move to the position specified by positive jump or negative jump. The node is then repositioned and then the new node is evaluated. This process is shown in the following code:

```
// we will now decide whether we
// will true jump or false jump
int pos = eval(ind,
               node->operator,
               node->comparison);

if (pos == 0) {
    ptree = &tree[node->neg_jump];
} else if (pos == 1) {
    ptree = &tree[node->pos_jump];
} else { // catch evaluation error
    // insert error code
    fitnessValue[gid] = -1;
    return;
}

// move the position of the node
node = ptree;
```

The test is evaluated and then the code decides whether to use the positive or negative jump. We then reassign the void pointer `ptree` before then assigning its address as the next node to evaluate.[4]

This process will then continue until a BUY or NOT BUY node is reached. If malformed trees are introduced to the system, a potentially dangerous situation could occur where the OpenCL kernel could loop indefinitely. While a maximum iteration could be applied to the loop this would reduce performance as this comparison will have to be checked every loop. In this implementation no protection is provided against malformed trees, apart from the correctness of the tree flattening process, which ensure that maximum performance can be extracted from the OpenCL device.

[4]The complete OpenCL kernel used by EDDIE-JOCL can be found at `http://cs.kent.ac.uk/∼febo/EDDIE_kernel.cl`.

Table 2: The parameter values of EDDIE used in the experiments.

Experiment Parameter	Value
Stock names:	Aggreko – FTSE 100 stock
	GSK – FTSE 100 stock
	Lloyds – FTSE 100 stock
Maximum initial depth:	6
Maximum depth:	8
Primitive probability in Grow method:	0.6
Terminal node crossover bias:	0.1
Hill Climbing probability:	0.001
No of generations:	51
Population size:	*variable*
Tournament size:	2
Crossover probability:	0.9
Reproduction probability:	0.1
Mutation probability:	0.01
Elitism percentage:	0.01
[w1, w2, w3]:	[0.6, 0.1, 0.3]
Training Days:	1000
Testing Days:	300

Table 3: Summary of the experiments concerning the execution time in seconds (*average time* [*standard error*]).

Stock	Pop.	EDDIE	EDDIE-JOCL	
			CPU	GPU
Aggreko	50	84.1 [20.5]	4.0 [0.4]	109.7 [12.5]
	250	447.3 [84.2]	22.3 [1.9]	574.4 [52.9]
	500	885.3 [119.9]	47.3 [2.8]	1170.5 [102.9]
	750	1406.8 [186.5]	71.3 [3.1]	1743.9 [131.5]
	1000	1597.3 [201.1]	95.2 [6.1]	2474.4 [168.0]
GSK	50	81.7 [23.9]	4.5 [0.5]	95.8 [12.6]
	250	422.7 [91.4]	19.8 [3.7]	563.2 [60.7]
	500	1005.7 [145.1]	41.9 [3.0]	1208.3 [167.5]
	750	1498.3 [198.0]	73.0 [4.2]	1807.6 [143.8]
	1000	1617.0 [267.2]	98.2 [6.0]	2398.6 [232.8]
Lloyds	50	77.1 [18.3]	3.7 [0.5]	94.2 [11.3]
	250	457.7 [79.1]	23.1 [2.6]	611.4 [65.5]
	500	809.4 [124.0]	44.1 [3.0]	1386.1 [208.8]
	750	1451.6 [203.6]	84.8 [3.6]	1761.2 [173.8]
	1000	1603.0 [278.1]	103.3 [6.0]	2397.4 [221.4]

5. RESULTS

We evaluated the performance of the improved EDDIE-JOCL implementation against the original EDDIE, running on both the CPU and GPU. All performance tests were run on the following hardware:

CPU - Intel Core i7-3770K

RAM - 8GB DDR3

GPU - AMD 7970 (Tahiti XT)

OS - Ubuntu 12.10 x86-64

The parameter values of EDDIE used in the computational experiments are shown in Table 2. The only parameter that is changed during the experiments is the population size. To test the scalability of the improved EDDIE-JOCL, we used population sizes {50, 250, 500, 750, 1000}. We also tested both the CPU and GPU devices. While there are less work units—less parallel fitness evaluations by consequence—when using the CPU, there is no memory copy overhead, since both the host and OpenCL processes are using the main memory. By using the GPU, we can significantly increase the number of work units, but we also introduce a memory copy overhead to copy the data from the main memory to the GPU memory.

The computation results are summarised in Table 3 and presented graphically in Figure 6. Overall, large speeds up were obtained by the EDDIE-JOCL when running in the CPU—average speed up of 19 times across all 5 population sizes, with the maximum speed up seen being 21 times the original EDDIE implementation. It is also interesting to note that the execution times do not vary significantly across different datasets (stock data)—the nature of the data (daily closing prices) and the problem are the same.

As can be seen, the execution time of EDDIE-JOCL using the GPU was considerably slower than the CPU. This was unexpected as the large number of stream processors available in the GPU should have given it a significant advantage—the more work units, the more parallel fitness evaluations. To explore the reason behind this occurrence, the fitness evaluation times of both GPU and CPU execution were recorded. Figure 7 shows the time taken for each EDDIE variation to evaluate a single tree. A single run with a population size of 50 trees was used across all three versions and the same seed was also used to ensure that the same trees were evaluated every time. The trees were then sorted by length and the average times for each tree length was calculated. The slowest version is EDDIE-JOCL when running on the GPU, with a time approximately 30% slower than EDDIE; the faster is EDDIE-JOCL running on the CPU is over 6 times faster than EDDIE.

The evaluation times confirmed the previous results: EDDIE-JOCL running on the GPU is the slowest of all three versions. To further investigate what was causing the difference between the two EDDIE-JOCL versions, both the evaluation and kernel times were recorded for EDDIE-JOCL. These times are shown in Figures 8 and 9, respectively. They compare the average time taken to evaluate one tree including all overheads, such as OpenCL set-up times, memory transfer times when sending and receiving data from and to the OpenCL device. The kernel time corresponds to the time to execute the kernel. The difference between the two time measures corresponds to the overheads created by JOCL and the OpenCL environment.

It can be seen that when running on the CPU, the kernel time accounts for a large proportion of the total evaluation time, since there is no memory copy overhead. This is not the case on the GPU, where the kernel time corresponds to very small proportion of the total evaluation and the OpenCL overheads dominate.

5.1 Discussion

5.1.1 *CPU speed ups*

EDDIE-JOCL running on the CPU showed large speed ups between 21 and 16 times, depending on the population size, and an average of 19 over the five population sizes. This speed up is very significant as only four cores and eight threads are available to EDDIE-JOCL. If the both evaluation methods on EDDIE and EDDIE-JOCL were equally as efficient, then a maximum speed up of eight times would be expected. There must be another reason for the extra efficiency and we hypothesise that this extra efficiency is due to the change from an object orientated paradigm to a procedural paradigm.

The original evaluator found in EDDIE uses the `eval` method

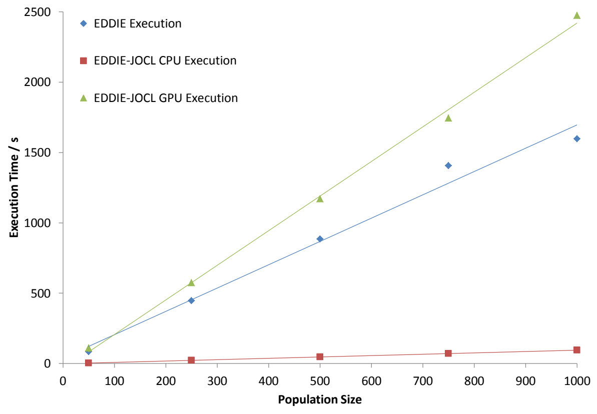


Figure 6: Execution times at five different population sizes for EDDIE and EDDIE-JOCL. Large speed ups are achieved when EDDIE-JOCL is using the CPU device—the maximum speed increase of 21 times over EDDIE was achieved at a population size 1000.

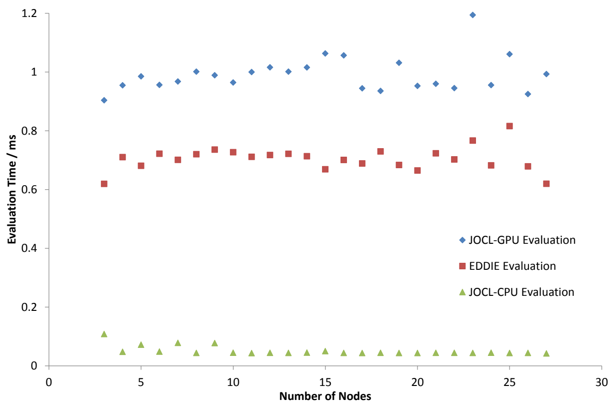


Figure 7: Evaluation times according to the number of nodes in the trees of the original EDDIE algorithm compared with EDDIE-JOCL running on both the CPU and GPU.

found inside every function that makes up a tree, following a object orientated paradigm. Therefore, to evaluate a tree, there is a need to traverse an object structure, which may incur method overheads when the program wants to move to the next object. In the EDDIE-JOCL implementation the trees are stored as arrays and the time to traverse an array is much faster than traversing an object structure

5.1.2 GPU (under-)performance

Despite the performance increases found when EDDIE-JOCL was run on the CPU, EDDIE-JOCL when running on the GPU proved to be slower than EDDIE, as can be seen in Figure 7. This was surprising as the highly parallel nature of the GPU architecture should have lent itself to speeding up the fitness function of EDDIE.

This result suggests that the losses are occurring as a result of OpenCL overheads. When running on the CPU, memory transfer overheads are not relevant as no memory copies are necessary—both host and OpenCL processes are using the main memory. But when running on the GPU, a memory copy does have to take place. This is confirmed by Figures 8 and 9, where the kernel times for

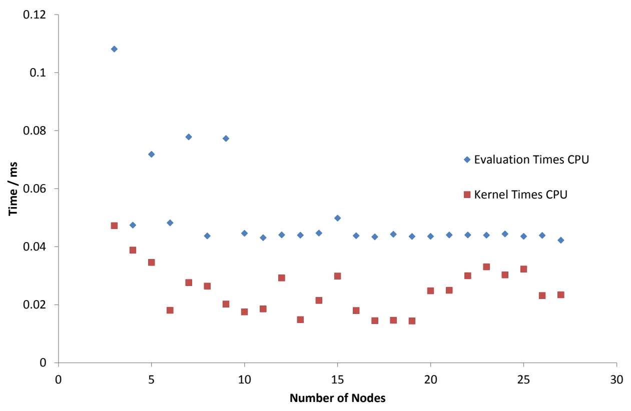


Figure 8: Evaluation and kernel times according to the number of nodes in the trees for EDDIE-JOCL when running on the CPU.

the CPU (Figure 8) are much closer to the total evaluation time. On the other hand, Figure 9 shows that the kernel times on the GPU are less than a tenth of the total evaluation time. The two configurations share the same code, so the only differences between the two is the memory transfer times, which seem to be very expensive in our case.

Looking only at the kernel times, the average kernel time on the CPU are 0.025ms whilst on the GPU they are 0.089ms. These differences could have a number of possible reasons. First, the cores contained on the CPU are much more complex than those found on the GPU. This, combined with the higher clock rate (1050Mhz on the GPU compared with 3500MHz on the CPU), allows the CPU to execute more instructions in the same time period. Secondly, the training data only contained 1000 samples, a small data size could cause some of the GPU stream processors (work units) to be under utilised.

Thirdly, the CPU does not subscribe to the SIMD (Single Instruction, Multiple Data) architecture, so is not penalised by branching code. In EDDIE, this occurs when the test results in each node evaluate differently for different elements in the testing data. On the CPU this is not a problem and all cores execute separately. However on the GPU, both positive and negative jumps on the node have to be evaluated sequentially and the stream processors waste cycles as they evaluate branches that are of no relevance to their calculation—the stream processors in the GPU must execute the same line of code. This was also noticed by Franco et al. in their implementation [7].

5.1.3 Possible improvements

The implementation discussed here could be further improved by changing the way the trees are dispatched to the OpenCL kernel. All the trees could be dispatched in one go. This is likely to decrease the transfer time as a single large block transfer would be quicker than many small transfers.

Once all the trees are on the OpenCL device there are two possible routes that could be taken to calculate the fitness function: the first is to let each stream processor calculate the entire fitness function of a single tree; the second is to parallelise both the trees and fitness function evaluation. This would require more logic in the kernel, as each instance would have to know which tree to evaluate along with the specific data that is to be used.

Finally a large amount of data is returned by the kernel, as the

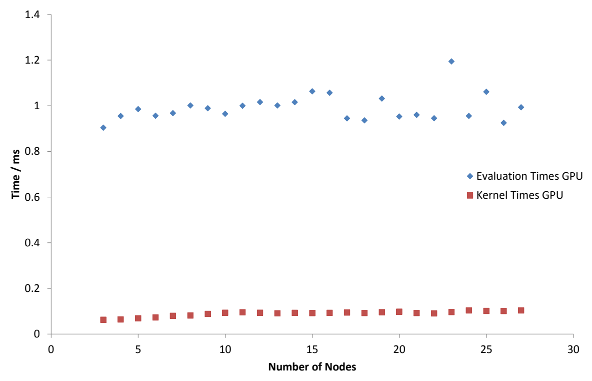


Figure 9: Evaluation and kernel times according to the number of nodes in the trees for EDDIE-JOCL when running on the GPU.

final fitness calculation is still calculated by EDDIE. A second kernel could be created to reduce this data into a single fitness value for each tree, which would then be returned, reducing the amount of memory that needs to be copied back.

6. CONCLUSION

In this paper we presented an extension to the financial forecasting algorithm EDDIE, called EDDIE-JOCL. As explained, speed is a very important aspect for financial forecasting algorithms, and can lead to significant increases in investors' profitability. Our goal thus in this paper was to improve EDDIE's computational performance by using an OpenCL kernel to evaluate candidate solutions. This allowed the parallel evaluation of solutions, both using multi-core CPUs and GPUs, by implementing the fitness function as an OpenCL kernel—following the assumption that the fitness evaluation is usually the expensive process in the GP run. EDDIE-JOCL achieved a maximum speed up of 21 times and an average of 19 times over the original EDDIE, when the OpenCL kernel was run in a multi-core CPU. Differently than previous results in the literature, running the kernel on a GPU device did not resulted in improvements over the CPU results. We identified the memory copy overheads from the main memory to the GPU memory and branching code (code that executes different instructions based on the input data) as potential causes of the somewhat unexpected (under-) performance of the GPU. As an overall observation, it seems that it is not guaranteed that by using GPUs, a performance increase will be achieved—specific refactorings to the code might be needed to fully use the potential of GPUs.

We have identified future improvements to better use the GPU resources and optimise memory copy overheads, which have the potential to lead to further performance gains. This include the copying of all candidate solutions (trees) in one operation, a more complex kernel that parallelises the evaluation of multiple trees and fitness cases and the use of a second kernel to compute the final fitness value of solutions.

7. REFERENCES

[1] F. Allen and R. Karjalainen. Using genetic algorithms to find technical trading rules. *Journal of Financial Economics*, 51:245–271, 1999.

[2] D. A. Augusto and H. J. Barbosa. Accelerated parallel genetic programming tree evaluation with OpenCL. *Journal of Parallel and Distributed Computing*, 73(1):86–100, 2013.

[3] M. Austin, G. Bates, M. Dempster, V. Leemans, and S. Williams. Adaptive systems for foreign exchange trading. *Quantitative Finance*, 4(4):37–45, 2004.

[4] J. Backus. The syntax and semantics of the proposed international algebraic language of Zurich. In *International Conference on Information Processing*, pages 125–132. UNESCO, 1959.

[5] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[6] R. Edwards and J. Magee. *Technical Analysis of Stock Trends*. Taylor & Francis, 1992.

[7] M. A. Franco, N. Krasnogor, and J. Bacardit. Speeding up the evaluation of evolutionary learning systems using GPGPUs. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1039–1046. ACM, 2010.

[8] M. Harris. Optimizing parallel reduction in CUDA.

[9] M. Hutter. JOCL API documentation.

[10] M. Kampouridis. An initial investigation of choice function hyper-heuristics for the problem of financial forecasting. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 2406–2413, 2013.

[11] M. Kampouridis, A. Alsheddy, and E. Tsang. On the investigation of hyper-heuristics on a financial forecasting problem. *Annals of Mathematics and Artificial Intelligence*, 2012. Accepted for Publication.

[12] M. Kampouridis and E. Tsang. EDDIE for investment opportunities forecasting: extending the search space of the GP. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[13] M. Kampouridis and E. Tsang. Investment Opportunities Forecasting: Extending the Grammar of a GP-based Tool. *International Journal of Computational Intelligence Systems*, 5(3):530–541, 2012.

[14] J. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press, 1992.

[15] W. B. Langdon. Graphics processing units and genetic programming: an overview. *Soft Computing*, 15(8):1657–1669, 2011.

[16] S. Martinez-Jaramillo. *Artificial Financial Markets: An agent-based Approach to Reproduce Stylized Facts and to study the Red Queen Effect*. PhD thesis, CFFEA, University of Essex, 2007.

[17] Oracle. java.nio.bytebuffer API.

[18] R. Poli, W. Langdon, and N. McPhee. *A Field Guide to Genetic Programming*. Lulu.com, 2008.

[19] T. Puźniakowski and M. A. Bednarczyk. Towards an OpenCL implementation of genetic algorithms on gpus. pages 190–203, 2012.

[20] H. Wang and A. S. Weigend. Data mining for financial decision making. *Decision support systems*, 37(4):457–460, 2004.