Stock Trading Strategy Creation Using GP on GPU

Dave McKenney
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
dmckenne@connect.carleton.ca

December 19, 2010

Abstract

This paper investigates the speed improvements available when using a graphics processing unit (GPU) for evaluation of individuals in a genetic programming (GP) environment. An existing GP system is modified to enable parallel evaluation of individuals on a GPU device. Several issues related to implementing GP on GPU are discussed, including how to perform tree-based GP on a device without recursion support, as well as the effect that proper memory layout can have on speed increases when using CUDA-enabled nVidia GPU devices. The specific GP implementation is designed to evolve stock trading strategies using technical analysis indicators. The second goal of this research is to investigate the possible improvement in performance when training individuals on a larger number of stocks and training days. This increased training size is enabled due to the speedups realized by GPU evaluation. Several different scenarios were used to test various speed optimizations of GP evaluation on the GPU device, with a peak speedup factor of over 600 (when compared to sequential evaluation on a 2.4Ghz CPU). Also, it is found that increasing the number of stocks and the length of the training period can result in higher out-of-training testing profitability.

1 Introduction

The introduction of general purpose computing on GPU devices has revolutionized the parallel computing field. Now, anybody with a small amount of initial investment can have the computing power of a small cluster contained inside their desktop computer. In fact, many people with powerful home computers already have this power sitting inside their boxes, without even knowing it is there.

Genetic programming, as developed originally by Koza in [14], lends itself particularly well to parallelization. Generally, nearly all computing time throughout a genetic programming run is taken by the evaluation of different individuals of the population. The evaluation of individuals is computationally expensive due to two factors: the number of individuals to evaluate and the number of fitness cases that must be evaluated for each individual. Using a parallel approach, these evaluations can be spread across many processors, resulting in massive speed increases. This speedup allows for more individuals, or more fitness cases, to be evaluated in the same amount of time, which can lead to better results.

The work reported in this paper aims to apply the benefits offered by a GPU device to the problem of genetic programming. Specifically, it aims to use genetic programming

(parallelized on a GPU device) to generate stock trading rules. This parallelization will be achieved by modifying the evaluation method of an already existing genetic programming package. This work also identifies and elaborates on the major implementation and performance issues (such as memory access optimization and recursion on GPU devices) present when migrating traditional GP to GPU devices. With the evaluations being executed on the GPU device, varying the number of stocks and length of training periods will be tested to see which produce the best out-of-training results.

In Section 2 of this paper, related works in both the area of massively parallel GP and the area of stock strategy creation using GP are presented. Section 3 outlines the trading model and genetic programming setup, as well as the algorithm used to evaluate the GP population. Details on implementation of parallel GP on GPU, including a stack-based interpretation algorithm and GPU memory access optimizations (coalescing), are given in Section 4. Section 5 compares the running time of the parallel GPU evaluation (and several optimization experiments) to the speed of a sequential evaluation approach, while Section 6 compares the profitability of varying amounts of training data. Future areas of improvement are identified and explained in Section 7, and the paper concludes with a summary of the work completed and results in Section 8.

2 Literature Review

2.1 Evolutionary Stock Trading

Due to the relatively new advances of general purpose computing on GPU devices, there is little previous work completed on massively parallel GP on GPU devices. Furthermore, there is no published work on generating stock trading rules using this approach. For these reasons, previous work on stock trading rule generation will be briefly presented, followed by a summary of the work on massively parallel GP on GPU devices.

In [5], many applications of evolutionary computation in the financial field are outlined. There are several sections on trading rule generation and algorithmic trading (the use of computer programs in the trading of financial assets). Also, background information is included, describing how technical analysis can be used to generate trading rules. While there are no practical results presented, it can serve as a useful starting point for those interested in the field of evolutionary computational finance.

A genetic algorithm approach has been used in [17] and [12] to generate stock trading strategies. In [17], the profitability of stock trading rules generated by a genetic algorithm using either a direct or indirect encoding of stock indicators. Using a direct encoding (where parameter values for functions are encoded within the genome of an individual), a total profit of 1628 Japanese yen (JPY) was realized throughout experimental trading. With direct encoding, however, the search space is extremely large. Each parameter value can range anywhere from the minimum value to the maximum value. For this reason, [17] proposed the use of indirect encoding, where parameter values are selected from a short list of possibilities (e.g. 5, 10, 15, 25, 50, and 100). This drastically cuts down on the search space, since each parameter can have only 6 possible values, instead of 100. With the decreased search space, a profit of 2370 JPY was realized (nearly 50% higher than with direct encoding). [12] encoded several Boolean indicators, connected by Boolean operators, into an individual. Each individual's chromosome was then evaluated for each day, with the input (indicators) determining the action for that given day. The evolutionary algorithm trained individuals over a 2 month period of trading on two foreign exchange markets (GBP/USD

and USD/DEM). A loss of 2% (GBP/USD) and 3.3% (USD/DEM) was realized by the best individual over the two month training data period. The losses increased further, to 9.3% (GBP/USD) and 15% (USD/DEM) in the following 3 month testing period.

The main problem with these genetic algorithm approaches for stock trading rule generation is that the program maintains a fixed length. Genetic programming (or a variable length genetic algorithm, such as [9]) however, allows programs to grow/shrink as evolution takes place. Also, using genetic programming, any combination of technical indicators can be used together to generate trading rules (which is not seen in a genetic algorithm approach). In [23], genetic programming is used to generate trading rules to be tested on four foreign exchange markets (CAD/USD, EUR/USD, GBP/USD, and JPY/USD). After trading on the exchange markets for a total of 1 year, the profitability of the generated rules were compared to the profitability of a buy-and-hold strategy (buying the currency at day 1 and selling it on the last day). Several different fitness approaches were tested, with the best approach resulting in an average of 5% higher return than the buy-and-hold approach. This approach also had a higher return than the buy-and-hold approach in 3 of the 4 markets, with a maximum difference of +13.14%. In [8] a genetic programming approach was implemented, created individuals by combining a subset of 6 Boolean trading rules together using standard Boolean functions. Each individual consists of both a buy and a sell tree, which are used to determine the action to be taken. The individuals were trained on 3 months of data from the GBP/USD market, with trades being executed every minute (while indicator values were updated every 15 minutes or at the end of every day). While profit was realized during the training interval, losses generally occured on the out of sample test data. The work presented in [8] was a significant motivation for this work.

2.2 Massively Parallel Genetic Programming

There are several approaches to parallelizing the evaluation stage (the most computationally expensive phase) of genetic programming implementations. The two most popular methods for parallelizing the evaluation of a single population are data parallel and population parallel. In a data parallel approach, a single individual's fitness cases are evaluated in parallel. This is easy to accomplish when the fitness cases are independent and easily separable (for example, symbolic regression and Boolean multiplexer problems, as described in [14]). A data parallel approach is a poor fit, however, when the fitness cases are dependent on each other. This problem is present in problems such as stock trading, where past profits/losses have an effect on future profits/losses. In these cases, it is much easier to implement a population parallel approach, where the fitness cases of a single individual are evaluated sequentially, while parallelization is achieved by evaluating several individuals at a time. In other cases, these two approaches can be combined into a hybrid evaluation approach, with both individuals and fitness cases being evaluated in parallel. This can be seen in the BlockGP approach in [21], and is also present in the work presented here.

One of the first and most often cited works on massively parallel genetic programming is [13]. This work was implemented on a MasPar MP-2 machine which used a single instruction multiple data (SIMD) architecture. However, a multiple instruction multiple data (MIMD) architecture was simulated using a stack-based interpreter which accepts genetic program individuals as data. Using this method, the requirements of single instruction multiple data architecture are met because each process core is executing the same step of the interpreter, using the individual programs as input. This allows many different individuals within the genetic programming population to be evaluated at the same time (in this specific case, 4096

individuals at a time). While the speedup of this approach when compared to a traditional CPU implementation was not presented, the benchmark tests were capable of evaluating thousands of individual programs in approximately 1 second.

In one of the original works involving genetic programming on a GPU device, [7] used a data parallel GP approach to parallelize evaluation of individuals. Instead of implementing a stack-based interpreter (as above), only one individual was evaluated at a time on the GPU. This parallelization was achieved by evaluating different fitness cases for the current individual at the same time. The implementation was tested on several GP problems (including the classic symbolic regression and 11-way multiplexer problems, as described in [14]), with a varying number of fitness cases. The length of time required for evaluation on the GPU was compared to the time needed to evaluate all individuals on a sequential CPU approach. In the case of 100 fitness cases, the CPU approach executed more than twice as fast as the GPU approach. With 150 fitness cases however, evaluation time was nearly equal. When the number of fitness cases increased, to 400, the GPU approach completed nearly ten times faster. This speedup was further realized by increasing the number of fitness cases to 2048, where the CPU took nearly 30 times as long as the GPU to evaluate all individuals. It would seem then, that using a low number of fitness cases does not allow the computing power of the GPU to be optimally used. In fact, with a small number of fitness cases, the overhead of GPU evaluation results in the CPU implementation being faster. For a data parallel approach to be most effective on a GPU then, a high number of fitness cases are required. Otherwise, some of the processing cores on the device will be underutilized, resulting in a smaller speed increase.

Another data parallel GP approach was implemented on a GPU device in [10]. Both the number of fitness cases and the maximum program length were varied, with performance being compared to that of a CPU implementation. For each test, GP individuals were randomly generated and evaluated (no genetic operations were performed, as the emphasis was on evaluation performance). It was found in all tests that speedup factors increased with both maximum program length and number of fitness cases. Speedup is seen with increasing number of fitness cases because fitness cases are evaluated in parallel on the GPU and sequentially on the CPU. Speedup is realized with increasing program length because the GPU needs to parse the individual tree fewer times (with many fitness cases evaluated in parallel), while the CPU implementation must parse the large individual trees for each fitness case. The first test involved individuals consisting of floating point operations (+, -, *, /) and terminals. Speedup factors (when compared to the CPU implementation) ranged from 0.04 (for a program length of 10 and 64 fitness cases), to 7351.06 (program length of 10000 and 65536 fitness cases). Similar results were also found when testing on the real GP problem of symbolic regression. Speedup factors in this case ranged from 0.02 for program length of 10 and 10 fitness cases, to 95.37 for program length of 10000 with 2000 fitness cases.

In [21] and [22] a population parallel approach to GP on GPU devices was implemented using the CUDA development kit from nVidia (similar approaches can also be found in [16] and [15]). As in [13], individuals are evaluated in parallel using a stack-based interpreter which accepts individual programs as input. Within [21], two approaches to evaluation distribution are presented. In the ThreadGP approach, each thread within the GPU is assigned to evaluate one individual, with each fitness case for the individual being evaluated on the same thread. The BlockGP method takes advantage of a newer architecture of nVidia GPU devices, which operate using a single-program-multiple-data (SPMD) architecture instead of SIMD. Within the newer GPU devices, there are a number of multiprocessors

(MPs), each of which maintains its own instruction pointer. With this architecture, each MP is capable of being at a different point in the program than the others. The BlockGP implements a hybrid parallel approach, evaluating each individual on a single MP, with all threads within the MP being used to evaluate different fitness cases in parallel. Divergence is avoided using the BlockMP approach, as the multiprocessors are capable of executing different instructions of the interpreter program and each thread within a multiprocessor is evaluating the same individual/instruction. Divergence is a major source of inefficiency when using an approach such as ThreadGP, as many threads will not be executing at a given time due to the fact that all threads within a multiprocessor must be at the same instruction within the interpreter. This inefficiency is what caused the BlockGP approach to perform faster evaluation over all tests carried out. Tests were completed using a symbolic regression problem with different population sizes and number of fitness cases. The highest speedup of BlockGP was found with the combination of the smallest number of individuals (512) and highest number of fitness cases (1024). ThreadGP performs poorly in this case because 512 is the minimum number of threads required to fill all stream processors on the GPU, so the GPU scheduler cannot swap out threads that are waiting for threads that are ready. BlockGP on the other hand involved 512 blocks, each with 32 threads. When one block is waiting, it can be substituted out for a block that is ready to perform computations, resulting in speedup. Furthermore, since BlockGP spreads fitness evaluations of an individual across 32 threads, a higher number of fitness cases allow the block to use all of its computational resources. With less than 32 fitness cases, it is impossible to fill all of a block's threads, resulting in poor performance. This work also found that once population increases beyond 2500 individuals, speedup remains approximately the same. This is because with 2500 individuals, there are nearly 5 times as many threads as there are stream processors, which allows the scheduler to substitute waiting threads efficiently. BlockGP still performs faster however, due to the lack of divergence.

When implementating parallel GP then, the number of fitness cases plays a key role in the parallelization scheme. With a high number of independent fitness cases, a data parallel approach will work well, as the high number of fitness cases will create enough threads to fully utilize the power of the GPU device. The current rate of GPU growth, however, will most likely relegate this approach as a massive number of fitness cases will soon be required to fill the processing cores of a device. When a smaller number of fitness cases are available, a population or hybrid parallelization approach will produce better results. These approaches allow the different multi-processors of a GPU device to work on different individuals at the same time, while fitness cases are evaluated either sequentially (population-parallel) or in parallel (hybrid approach). When using these approaches, however, care must be taken to address the problem of divergence. With a number of individuals being evaluated on the same multiprocessor, as in a population parallel approach, divergent code branches can greatly decrease the efficiency of the computation. This problem can be addressed using a hybrid scheme, as a evaluating a single individual on a multi-processor (with fitness cases in parallel) can eliminate the problem of divergence.

3 Genetic Programming and Stock Trading Model

3.1 Lilgp

To perform the genetic programming operations (population creation, breeding, mutation, etc.), Lilgp ([19]) was used. Lilgp is a genetic programming tool written in the C program-

ming language with a number of goals including speed and ease of use. The version of Lilgp used also includes strongly typed GP [18], which allows for typed functions, inputs, and trees. This cuts down on the manual checking necessary to ensure that a tree is valid, since only valid inputs will be supplied for each function. The main modification to Lilgp, for the research reported here, was the removal of the standard sequential population evaluation function, which was replaced with a parallel GPU evaluation function. This parallel function uses a hybrid parallelization, with individuals and fitness cases evaluated in parallel. Since it is difficult to parallelize the fitness evaluations for single stocks, one thread is allocated for every individual/stock combination. These threads are organized in a way such that all threads of an individual are contained within the same warp, which decreases the amount of divergence, since a multi-processor will be evaluating a single individual (and thus, a single instruction) at any time.

3.2 Functions and Terminals

The stock problem implementation consisted of 31 possible input values (terminals) and 5 Boolean functions (AND, OR, NOT, <, >). The inputs were calculated using formulae from [4], which has much more information on the calculation and use of these indicators. Below is a list of all inputs used, with a brief explanation of each.

Floating Point Terminals

- Simple Moving Average: Over a period of either 1 (MA1), 5 (MA5), 10 (MA10), 15 (MA15), 25 (MA25), 50 (MA50), 75 (MA75), 100 (MA100), 150 (MA150), or 200 (MA200) days.
- Exponential Moving Average: Over a period of either 5 (EMA5), 9 (EMA9), 15 E(MA15), 20 (EMA20), or 25 (EMA25) days.
- Price: The closing price (CP) and the typical price (TP).

Boolean Terminals

- Volume Indices: Whether the NVI (negative volume index) or the PVI (positive volume index) are above (NVIG/PVIG) or below (NVIL/PVIL) zero.
- Moving Average Convergence/Divergence (MACD): Whether the MACD is above (MACDGZ) or below (MACDLZ) zero, as well as whether the MACD greater than (MACDG) or less than (MACDL) the MACD signal line.
- Money Flow Index (MFI): Whether the MFI is above 80 (MFIG), or the MFI is below 20 (MFIL).
- Ease of Movement (EOM): Whether the ease of movement is greater (EOMG) than or less than 0 (EOML).
- Commodity Channel Index (CCI): Whether the CCI is greater than 100 (CCIG), or less than 100 (CCIL).

3.3 Individuals

Each individual in the population consisted of two trees, with each evaluating to either true or false. The first tree represented a buy signal, while the second represented a sell signal. On each training day and for each stock, the values of these trees (given input values for the specific stock/day combination) would be used to determine the trading action for the day/stock. The decision matrix used to select an action can be seen in Figure 1.

		Buy Tree	
		True	False
Sell Tree	True	No Action	Sell All Shares
	False	Buy Max. Shares	No Action

Figure 1: Decision matrix for a single stock/day combination

3.4 Training and Fitness Calculation

A strategy's (individual's) fitness is dependent on how much profit that strategy would make while trading a set of stocks over the set of training days. To begin the evaluation of an individual, an initial amount of money ($10 000) is allocated for each stock. For every stock and training day, the individual's trees are evaluated and the appropriate action is taken. A pseudocode algorithm for population evaluation is presented below in Algorithm 1.

3.4.1 Buy Action

When an individual i generates a buy signal for a specific stock s on day d (and the individual does not currently own any shares of s) the following calculations are performed to adjust the state of that individual:

$$\begin{aligned} Money_{i,s} &= Money_{i,s} - CF \\ Shares_{i,s} &= \left\lfloor \frac{Money_{i,s}}{CP_{s,d}} \right\rfloor \\ Money_{i,s} &= Money_{i,s} - (Shares_{i,s} \times CP_{s,d}) \end{aligned}$$

Where CF represents the commission fee for trading (set to a constant value of $1/trade) and $CP_{s,d}$ is the closing price of stock s on day d. It is easy to see that this model of stock trading is quite simple (e.g., it does not address slippage, or what effect large quantities of trading may have on the stock price). The main focus of this work however, was using GPU devices for GP evaluation. Further enhancements to the stock trading model are proposed in Section 7.1).

3.4.2 Sell Action

When an individual i generates a sell signal for a specific stock s on day d (and the individual currently owns a number of shares of s) the following calculations are performed to adjust

the state of that individual:

$$\begin{aligned} Money_{i,s} &= Money_{i,s} + (Shares_{i,s} \times CP_{s,d}) - CF \\ Shares_{i,s} &= 0 \end{aligned}$$

These actions are also carried out for each stock s that the individual owns shares in after the last training day, so the overall profitability of the individual can be measured.

3.5 Fitness Calculation

Initially, once evaluation had completed, the raw fitness of an individual was calculated solely on ROI (return on investment) using the formula below (where S is the set of all stocks trained on):

$$Fitness_i = ROI_i = \frac{\sum_{s \epsilon S}(Money_{i,s} - 10,000)}{|S| \times 10,000}$$

This approach, however, biased the population towards stocks which increased in value over the training period. As an example, if a single stock increased 5000% over the training data, the entire ROI of an individual could be increased significantly just by buying this stock on the first day and holding it until the end of training. Also, a high ROI may not necessarily be indicative of a superior trading strategy. For example, a trading strategy may achieve an ROI of 100% over the training data, but the set of stocks being trained on increased 150% in total. While the 100% ROI seems impressive, more profit would have been made using a buy and hold strategy (buying the stocks on the first day and selling on the last). Comparing profitability of trading strategies to a buy and hold approach is common, and can be seen in [16], [8] and [11]. For these reasons, the fitness function was modified to include both the ROI achieved by the trading strategy and the ROI of a buy and hold strategy. This improved fitness calculation follows (where ROI_{BH} is the return on investment when buying all stocks on the first day and selling all stocks on the final day, and ROI_i is calculated as above):

$$Fitness_i = ROI_i - ROI_{BH}$$

4 Genetic Programming on GPU

While GPU devices offer extremely high computational power, they have lacked the execution stack necessary to perform straightforward tree-based GP. This problem is being addressed by nVidia's Fermi architecture [3], which will implement its own call stack, thus allowing recursion. This work however, was initially completed without the ability to use recursion on the GPU device. For this reason, a stack-based interpreter kernel was used on the device, allowing the tree GP individuals to be evaluated.

4.1 Conversion to RPN

Generally, GP trees are parsed such that the resulting expression is in prefix notation. The original GP work by Koza in [14] represented programs as Lisp-like S-expressions, which

Input: The GP population, with individuals/trees represented in RPN format
Output: An array $Fitness$ which stores the raw fitness of each individual
foreach *Individual i in population (in parallel)* **do**
  $Fitness_i = 0$;
  foreach *Stock s in the set of training stocks (in parallel)* **do**
    $Money_{i,s} = 10000$;
    $Shares_{i,s} = 0$;
    $Signal = 0$;
    foreach *Day d in the set of training days* **do**
      `/* Tree evaluations done using Algorithm 2 */`
      $Signal = eval(buy\_tree_i, s, d)$;
      $Signal = Signal - eval(sell\_tree_i, s, d)$;
      if $Signal > 0$ *and* $Shares_{i,s} = 0$ **then**
        $Money_{i,s} = Money_{i,s} - CF$;
        $Shares_{i,s} = \left\lfloor \frac{Money_{i,s}}{CP_{s,d}} \right\rfloor$;
        $Money_{i,s} = Money_{i,s} - (Shares_{i,s} \times CP_{s,d})$;
      else if $Signal < 0$ *and* $Shares_{i,s} > 0$ **then**
        $Money_{i,s} = Money_{i,s} + (Shares_{i,s} \times CP_{s,d}) - CF$;
        $Shares_{i,s} = 0$;
      end
    end
  end
  foreach *Stock s in the set of training stocks* **do**
    $Fitness_i = Fitness_i + (\frac{Money_{i,s} - 10000}{10000} - ROI_{BH})$;
  end
end
return *Fitness*

Algorithm 1: Evaluation of a GP population

are recursive and represented in prefix notation. Using this method, a string representation of the tree in Figure 2 would be as follows:

$$(\; AND \; (\; > \; MA10 \; MA50 \;) \; (\; OR \; (\; < \; EMA9 \; MA25 \;) \; NVIG \;) \;)$$

To perform recursion using a stack-based interpreter on the GPU, however, individuals must be represented in postfix (reverse polish) notation, as in [16, 15, 20]. To convert individuals into this notation, the GP tree is parsed in much the same way. Functions however, are added to the string only after their input values have been parsed. The same tree from Figure 2, represented as a RPN string would be:

$$MA10 \; MA50 \; > \; EMA9 \; MA25 \; < \; NVIG \; OR \; AND$$

It can be seen, that RPN does not require the use of brackets. Furthermore, it can be executed in a left to right manner, without the need to store previous function calls. These strings can then be executed using the stack-based interpreter as described below.

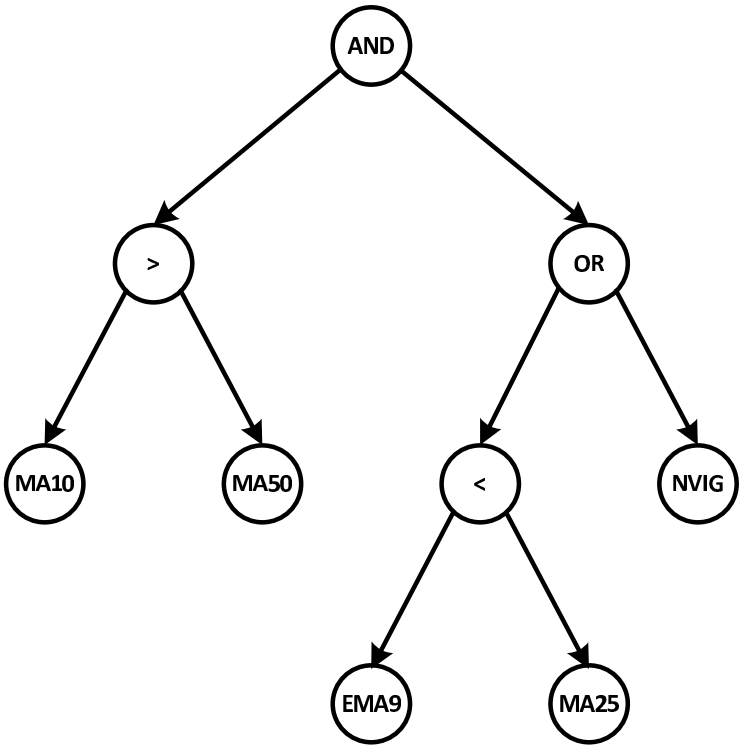

Figure 2: An Example GP Tree

4.2 Stack-based interpreter kernel

The stack-based interpreter kernel is a relatively simple device which evaluates individuals supplied in RPN. Each thread on the device is assigned its own stack space, and can therefore execute in parallel with all other threads with no synchronization/locks required. Each thread on the device is assigned an individual and a specific stock number. It will then loop through all training days, evaluating the individual's buy/sell trees using the indicators for the stock it has been assigned to.

Every function and terminal allowed in the GP algorithm is assigned a unique operation code. For each tree evaluation, the stack-based interpreter parses the string representation of the tree from left to right, performing a simple case statement on each of the bytes. This case statement looks at the current byte to decide which operation code it matches. Once the operation code is determined, the correct action can be taken. If the operation code belongs to a terminal, the terminal's value for the current stock/day is retrieved and pushed onto the stack. If the operation code belongs to a function, the previous values (the number of values varies depending on the arity of the function) are popped from the stack and taken as input to the function. The resulting value of the function evaluation is then pushed onto the stack. When the end of the individual's tree is reached (represented by a null character within the string), the value of the first value of the stack is returned as the value of the tree. The algorithm used to evaluate a specific tree is shown below in Algorithm 2.

```
Input: A stock s, day d, and tree in RPN rpn_string
Output: 0 (false) or 1 (true)
pc = 0;
sp = 0;
while true do
    op = rpn_string[pc];
    switch op do
        case op = null
            return stack[0];
        case op = AND /* Similar behaviour for other functions */
            arg1 = stack[sp − 1];
            arg2 = stack[sp − 2];
            sp = sp − 2;
            stack[sp] = arg1&&arg2;
            sp = sp + 1;
            break;
        case op = MA10 /* Similar behaviour for other terminals */
            stack[sp] = MA(s, d, 10);
            sp = sp + 1;
            break;
    endsw
    pc = pc + 1;
end
```

Algorithm 2: Stack-based interpreter algorithm for a single tree represented in RPN

4.3 Important Performance Factors

In many ways, performing GP on GPU devices seems at odds with much of the basic GPU programming performance guidelines. The stack-based interpretation loop consists of a large if-else statement. This of course can lead to a high level of divergence as different threads follow very different execution paths. Global memory accesses are also seen as something that can decrease performance significantly. Evaluating GP individuals results in an extremely high number of global memory accesses, as inputs to functions must be read, stacks must be updated, and results must be written. Fortunately, with a small amount of analysis/design, the use of CUDA and nVidia devices, combined with the large number of fitness cases in this particular GP problem, alleviate many of these concerns.

4.3.1 Work layout

As has been mentioned, a CUDA-enabled GPU device consists of a number of multiprocessors, each of which has a set of (generally 32) stream processors which at anytime are executing a warp (a group of 32 threads). These stream processors are only capable of executing a single instruction at a time. For this reason, the large number of execution paths possible using a large switch statement (as in the stack interpreter) can result in a large percentage of stream processors being inactive at any given time. When analyzing a large and varied amount of stock data per individual though, this problem can be overcome easily. We chose a multiple of 32 as the number of stocks to analyze over the specific training days (the number of stocks tested is either 32, 64, or 128). Using these numbers, each warp (of

32 threads) will be evaluating the same individual on a different stock. This eliminates the problem of divergence, as each thread in the warp will be evaluating the same individual, and thus the same instruction, at all times.

In general terms, an effort needs to be made to minimize the amount of divergent branches occuring with a warp. With GP, if an individual can use 32 (or a multiple of 32) threads, it ensures that any given warp will be evaluating the same individual at all times. This way, there will be no divergence within the stack interpreter because the same operation will be executed by all 32 threads. With GP problems where a large number of independent fitness cases are available, this is an easy problem to deal with, as it will be easy to assign a subset of fitness cases to each of the 32 threads. In fact, with as little as 32 fitness cases, all threads of a warp will be able to deal with a single individual, eliminating divergence. It is also possible to use more than 32 threads per individual, however it is best to keep the number of threads per individual to a multiple of 32. This ensures that all threads within a warp are evaluating the same individual (this approach is used here, where 32, 64, or 128 threads are used per individual).

4.3.2 Memory layout

In the case where a large amount of memory accesses are required, coalesced memory accesses are extremely important for speedup. CUDA-enabled GPU devices access device memory using 32-, 64-, or 128-byte memory transactions, as explained in [2]. Threads residing in the same warp, which are accessing consecutive memory locations, can have what would normally be multiple accesses, coalesced into a single access. The degree of coalescence then, is determined by the warp-level data locality, which is determined by both the memory layout and the thread layout. As an example, with each of the 32 threads in a warp evaluating one of the 32 training stocks, accessing a single-byte Boolean indicator for each could take 32 memory read operations, with extremely poor data locality, or a single 32-byte memory read with the data located consecutively in memory. Generally, a thread and memory layout should be designed such that consecutive threads will access consecutive memory locations.

One of the most memory-intensive operations in this case, is the reading/writing of the stack. As was mentioned above, setting the number of stocks being analyzed to a multiple of 32 ensures that all threads in a warp will be executing the same operation at all times. For this reason, we know that all threads in a warp will be accessing the same depth of their stack at all times (will have the same stack pointer). This observation is important when deciding on a layout for the array, which will store the stacks for all of the threads. Two possible layouts are shown in Figure 3, one of which performs much better than the other. In the bottom layout, the stack of each individual is placed consecutively in the array. As can be seen, the operation PUSH(MA10), which would push the 10 day moving average onto the stack of each individual in the warp, results in memory writes that are far apart. These writes will not be coalesced, resulting in 4 write operations being performed. Using the top layout, elements of the same stack depth of all threads are stored sequentially in memory. Now, when the threads of a warp wish to write the 10 day moving average, they are writing to consecutive memory blocks. What took 4 write operations originally, will now be coalesced into a single memory operation.

The resulting speed increase of improved memory coalescence is shown and explained further in Section 5.2.

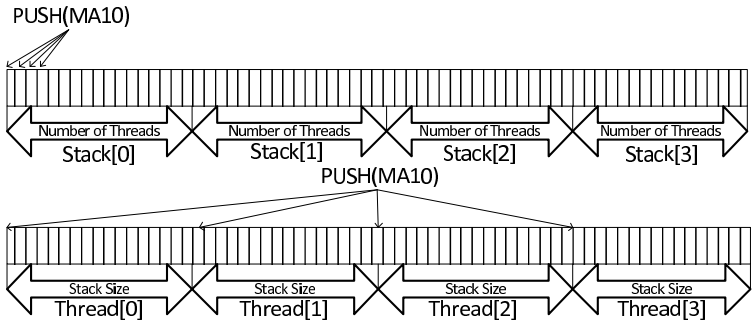

Figure 3: Two Possible Layouts for the Stack Array on a GPU Device

5 Running Time Comparisons

5.1 Setup

Each test was ran with identical parameters: 25000 individuals in the population with 32 stocks evaluated over 768 training days for 50 generations. Each test (aside from the sequential test, which was run only 3 times due to the time required) was also ran 10 times using different seed values for the Lilgp random number generator. The values presented here are averages over the 10 runs for each test. On the GPU, each block contained 512 threads, with individuals taking 32 threads (one for each stock) consecutively. The number of blocks was set such that there would be enough threads for all individual/stock combinations.

5.2 GPU vs. GPU Results

Figure 4 shows the running times of the original GPU evaluation (Ind&Stock/Thread), as well as several modified approaches. Within the chart legend, CI specifies that the individual RPN strings were stored in constant memory (which is cached) as opposed to global memory (which is not cached), SMS specifies that the stack was moved into shared memory which can be accessed much faster than global memory, and CO refers to an improved layout of the array containing the stock indicators (resulting in much higher coalescence).

From Figure 4, it can be seen that simply moving the RPN strings to constant memory, or the stack to shared memory, had no real effect on the running times. This is due to the fact that memory access to this information was already highly coalesced, which allowed global memory to match the speed of the cached constant memory and the on-chip shared memory. These two results once again show the importance of coalescing memory accesses, as the decreased memory accesses combined with the latency hiding of the GPU scheduler allows the much slower global memory accesses to perform as well as faster/cached memory.

In the initial tests, stock indicators were stored in a flattened 3-dimensional array (indexed by stock/indicator/day). This however led indicators of different stocks to be stored far apart within the memory on the device. The memory reads then, will not have good data locality and will not coalesce, resulting in poor performance. In the final two tests, a better layout was designed where the 3-dimensional array was indexed as day/indicator/stock. This way, all threads in a warp (who are evaluating the same individual, and thus the same day/indicator, but consecutive stocks), will be reading consecutive memory blocks. This modification is much the same as the different stack memory layouts explained in Section

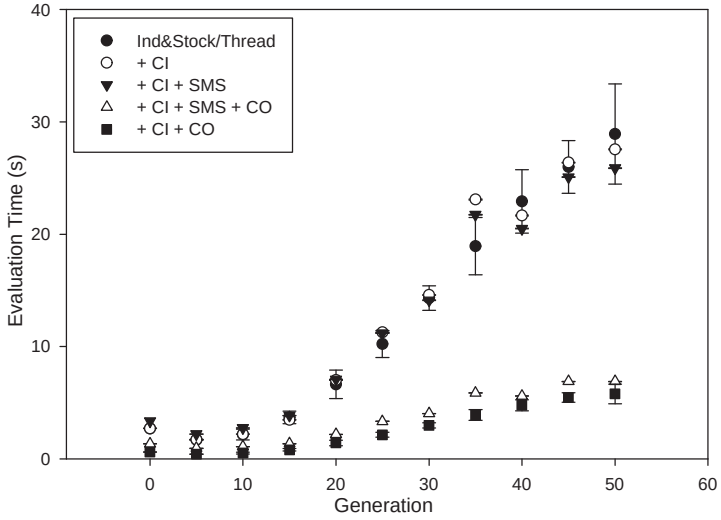

Figure 4: Average Evaluation Time for Different GPU Evaluations With 1 SD Error Bars

4.3.2 above. The massive increase in memory throughput resulted in an evaluation time under 20% of the original tests.

5.3 GPU vs. Sequential Results

The time taken to evaluate each generation sequentially can be seen in Figure 5; while Figure 6 charts the speedup attained using the best GPU approach found (as compared to the sequential version). At one point (generation 45), the GPU evaluation speedup reaches a maximum peak of 590.28 times faster than the sequential evaluation. The speedup line, however, appears to be increasing, so evolution was continued to generation 75 where an average speedup factor of 667 was attained. This number will most likely continue to grow slightly, as the average length of individuals in the population tends to increase as evolution continues. The speedup graph also shows that there is a large amount of variance in the results. One of the main causes of this is the extremely small evaluation times using the parallel approach. For example, evaluating generation 10 using the fastest GPU implementation took 0.4050 seconds on average. An evaluation time change of only 0.1 seconds represents a 25% change in running time. This, in turn, transfers to the speedup calculation, which divides the sequential evaluation time (typically very large) by the parallel evaluation time (very small), resulting in high variance.

6 ROI Comparison

In much of the previous work on stock trading strategy creation using evolutionary algorithms, one of the biggest problems present was overfitting to training data. This can be seen in [17] and [8] and is most evident in the divergence of training case profit vs. testing case profit. Overfitting to the training data occurs when a strategy becomes too specialized to a specific data set. Generally, the training data has consisted of a small number

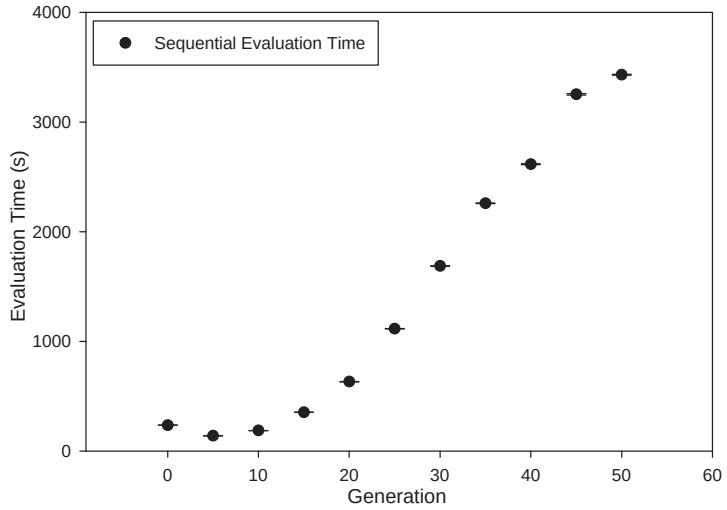

Figure 5: Evaluation Time Using a Sequential Evaluation Approach With 1 SD Error Bars

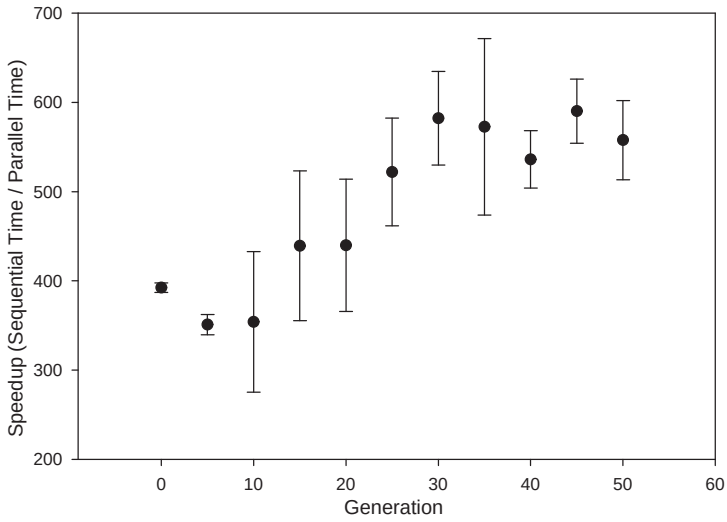

Figure 6: Speedup Factor (Sequential Evaluation Time / Parallel Evaluation Time) With 1 SD Error Bars

of entities (whether it is stocks in the stock market, currencies in the foreign exchange market, etc), and/or a small time frame. The evolutionary algorithms applied evolve very effective trading strategies for the training data; however, these strategies generally do not apply outside of the training area (as can be seen with much different training and testing profits). The main focus (from a trading strategy generation point of view) of this work was to investigate the effects that increasing the training data size may have. Using GPU devices to speedup evaluation allows more training data to be looked at in the same amount

of time. So perhaps, if we increase both the number of stocks and the number of training days, we can reduce this overfitting problem and evolve strategies which are more profitable outside of the training set.

Figures 7 and 8 plots the average ROI/100 Days above the buy and hold ROI on training data for 50 generations using different numbers of stocks/training days, while Figures 9 and 10 shows the same values for testing data. Both attempts at training using 32 stocks reach the highest ROI, with 32 stocks over 768 days of training data managing over 80% more ROI than the buy and hold strategy. Within Figures 9 and 10 , however, it can be seen that these two strategies also achieve the lowest ROI on the testing data. This is typical of overfitting, where the strategy has become too specific to the training data and fails to generalize to the testing data. In both cases, where 128 stocks are used for training, the lowest training ROI is generated; however, the strategies generated achieve the first and third highest testing ROI (the second highest test ROI was generated using the large number of days and medium number of stocks). It can also be seen in Figures 9 and 10, that all strategies evolved using 768 days of training data, outperform the same number of stocks trained over 256 days. It would seem then, that increasing your training data set will more than likely result in better out-of-training results. It was also found that using a higher number of stocks (64 or 128) resulted in less variance over the testing data. Individuals training on a higher number of stocks then, are producing results with less risk (negative variance), but also less reward (positive variance). It should also be noted though, that all solutions still show a slight downward trend, which may signify that overfitting is still happening (although much slower) with larger training sets. Further tests will include an even larger number of stocks, in hope that the power of GP on GPU devices can generate even more effective trading strategies that are applicable outside of the training dataset.

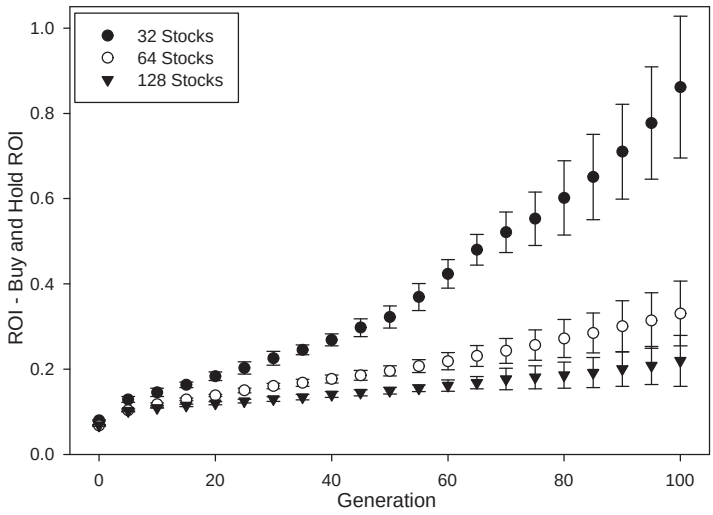

Figure 7: Profitability of Different GP Runs on Training Data With 1 SD Error Bars (Using 768 Training Days)

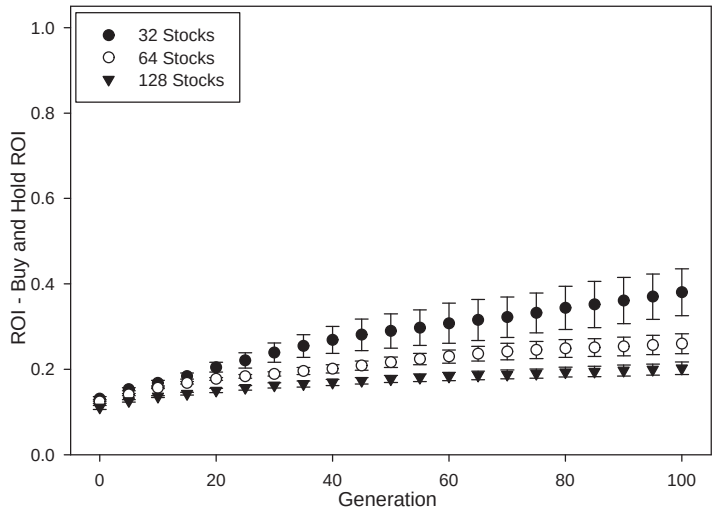

Figure 8: Profitability of Different GP Runs on Training Data With 1 SD Error Bars (Using 256 Training Days)

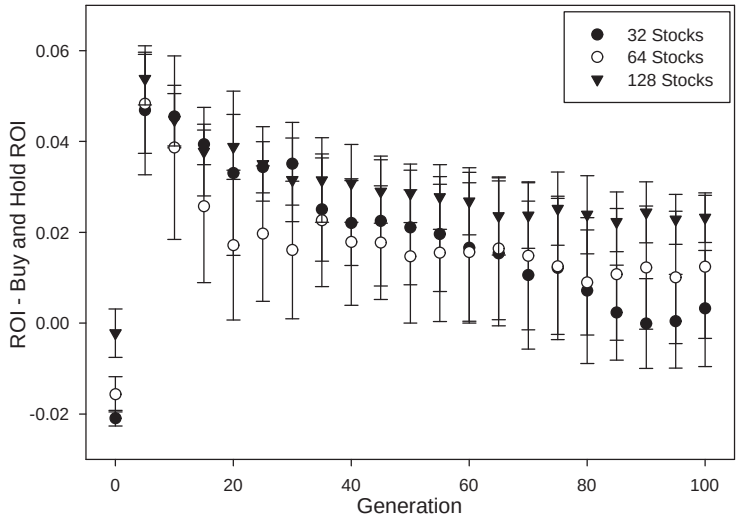

Figure 9: Profitability of Different GP Runs on Testing Data With 1 SD Error Bars (Using 768 Training Days)

7 Future Work

7.1 Stock Trading Strategy Creation Using GP

There are many ways in which the stock trading strategy creation work presented here can be improved. First of all, the stock trading model used is very simplistic. Currently a

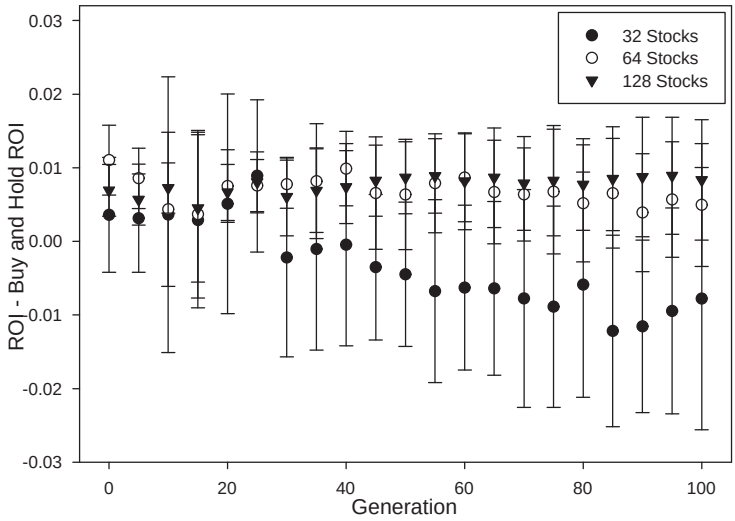

Figure 10: Profitability of Different GP Runs on Testing Data With 1 SD Error Bar (Using 256 Training Days)s

specific amount of money is allotted to each specific stock; however, some level of portfolio management could be used to distribute the available money among all possible stocks. The GP trees could also be modified to generate real numbers representing the potential profitability of a stock at the current time. This way money could be probabilistically allocated to stocks, with more money being placed into what are deemed to be the most profitable stocks. The work in [11] recognized significantly higher profits with the addition of leverage and the ability to buy more of a stock after the initial purchase, so perhaps higher returns could be seen when including these extensions as well.

The implementation used also attempts to generate a single strategy to use across all time periods. This could be modified to train on multiple shorter periods, with smaller testing periods following immediately after the training data, as in [17]. Also, a relatively small number of easy-to-implement indicators were taken from [4]. Future work could address this by including a wider range of indicators, as well as other available economic information (e.g., interest rates, stock index values). This could result in the evolution of economy-aware individuals who select different plans based on basic economic data.

Finally, the fitness function used to evaluate individuals (Training ROI - Buy and Hold ROI) is quite simple, and may carry hidden biases that result in poor out of sample testing results. A second evolutionary algorithm could be implemented to evolve a fitness function which, when used to evolve individuals on training data, will result in high profitability on testing data. Inputs to this algorithm could include measurements such as average ROI over all stocks, deviation of ROI over all stocks, maximum ROI attained on a single stock, etc. With the problem broken down into numerous testing periods (as explained above), these inputs could be combined/weighted to maximize the effectiveness of generated individuals.

7.2 GP on GPU Devices

One of the biggest breakthroughs for tree-based GP on GPU devices may be nVidia's Fermi architecture, which will support recursive function calls. This will allow tree-based GP to be performed directly on the GPU, which eliminates the conversion of individuals and stack interpretation that was necessary without recursion support. Also, the entire GP engine could easily be moved to GPU, which would eliminate the costly process of copying individuals/results between the host and device.

It has also been mentioned that linear genetic programming (in which individuals consist of a sequence of imperative instructions or machine code as explained fully in [6]) may be a better fit for GPU genetic programming. With linear genetic programming, no recursion is required, so even without recursion support, a linear GP approach could be implemented directly on a GPU device. Furthermore, even with recursion support on a GPU, the depth of the tree-based GP individuals would be limited to the size of the stack. With linear GP however, there are no stack depth limitations, allowing programs to grow as long as necessary.

To speedup execution with the current implementation, a double-buffering approach could be used to evaluate a subset of the population while another subset is being converted and copied to the GPU. Once again, with the Fermi architecture, many CUDA operations are asynchronous (including kernel launches), which would allow the double buffering approach to be easily implemented. Finally, it is obvious from the results that memory access optimization plays an extremely important role in determining the speed at which the GPU can evaluate GP individuals. The Compute Visual Profiler supplied with the CUDA toolkit allows in depth analysis of GPU operation. This tool can be used to identify further areas of memory optimization, as well as other possible speed improvements which could increase speedup even more. More information on the Compute Visual Profiler can be found in [1].

8 Conclusion

Within this work, genetic programming was used in an attempt to solve the real-world problem of stock trading strategy generation. A GPU device was used to evaluate individuals within the GP population through stack-based interpretation (due to the lack of recursion support on many GPU devices). With a small amount of memory access optimization, a speedup factor of over 600 was reached when compared to a sequential evaluation of the same data running on a 2.4Ghz CPU.

The effect of increasing the size of the training set (through the addition of more stocks and longer training periods) was also investigated. It was found that using small training sets resulted in the worst testing results. Furthermore, the best test results were found when using the largest training sets. These results supported the hypothesis that analyzing more stocks over a longer period of time can generate a more general and effective stock trading strategy. The speedup gained using GPU devices for evaluation enable this large training set to be evaluated quickly, while a sequential implementation would make this approach unfeasible.

Finally, several areas of improvement for both GP on GPU and stock trading strategy creation using GP were identified. Continuing work and addressing these possible areas of improvement may result in faster evaluation of individuals, as well as a much more profitable trading solution.

References

[1] Cuda Compute Visual Profiler, October 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf.

[2] Cuda Programming Guide 3.2, 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[3] nVidia Fermi Architecture, 2010. http://www.nvidia.com/object/fermi_architecture.html.

[4] Steven B. Achelis. *Technical Analysis from A to Z.* Irwin, 1995.

[5] Anthony Brabazon, Michael ONeill, and Ian Dempsey. An introduction to evolutionary computation in finance. *IEEE Computational Intelligence Magazine*, 2008.

[6] Markus Brameier. *On Linear Genetic Programming.* PhD thesis, Universitt Dortmund, 2004.

[7] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573, New York, NY, USA, 2007. ACM.

[8] M.A.H. Dempster and C.M. Jones. A real-time adaptive trading system using genetic programming. *Quantitative Finance*, 1:397–413, 2001.

[9] David Goldberg, K. Deb, and B. Korb. Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[10] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In *EuroGP'07: Proceedings of the 10th European conference on Genetic programming*, pages 90–101, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Akinori Hirabayashi, Claus Aranha, and Hitoshi Iba. Optimization of the trading rule in foreign exchange using genetic algorithm. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1529–1536, 2009.

[12] A. Hryshko and T. Downs. An implementation of genetic algorithms as a basis for a trading system on the foreign exchange market. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 3, pages 1695 – 1701 Vol.3, dec. 2003.

[13] Hugues Juillé and Jordan B. Pollack. Massively parallel genetic programming. *Advances in Genetic Programming*, 2:339–357, 1996.

[14] John R. Koza. *Genetic programming : on the programming of computers by means of natural selection.* MIT Press, Cambridge, 1992.

[15] W. Langdon. A many threaded cuda interpreter for genetic programming. In *Genetic Programming*, volume 6021 of *Lecture Notes in Computer Science*, pages 146–158. Springer Berlin / Heidelberg, 2010.

[16] W. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85. Springer Berlin / Heidelberg, 2008.

[17] Kazuhiro Matsui and Haruo Sato. A comparison of genotype representations to acquire stock trading strategy using genetic algorithms. *Artificial Intelligence Systems, IEEE International Conference on*, pages 129–134, 2009.

[18] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1995.

[19] Dr. Bill Punch. Lilgp, September 1998. `http://garage.cse.msu.edu/software/lil-gp/`.

[20] Denis Robilliard, Virginie Marion, and Cyril Fonlupt. High performance genetic programming on GPU. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS '09, pages 85–94, New York, NY, USA, 2009. ACM.

[21] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Population parallel GP on the G80 GPU. In *EuroGP'08: Proceedings of the 11th European conference on Genetic programming*, pages 98–109, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10:447–471, 2009.

[23] Garnett Wilson and Wolfgang Banzhaf. Interday foreign exchange trading using linear genetic programming. In *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1139–1146, New York, NY, USA, 2010. ACM.