# LITERATURE REVIEW: Accelerating Stock Trading Rule Creation Using Genetic Programming on a GPU Device

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

October 17, 2010

## 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 regular desktop computer. In fact, many people with powerful home computers already have this power sitting inside their boxes, without even knowing its there.

The problem of genetic programming 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. This step is extremely computational 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 should result in better results.

This project 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 done by modifying the evaluation method of an already existing genetic programming package. With the evaluations being executed on the GPU device, several different work distribution approaches will be tested against each other, so see which results in the best performance gains.

## 2 Literature Review

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. Also due to this fact, there is absolutely no published work on generating stock trading rules using this approach. For these reasons, previous work on stock trading rule generation will be breifly presented, followed by a summary of the work on massively parallel GP on GPU devices.

[1] outlined many applications of evolutionary computation in the financial field. Among this work, there are several sections on trading rule generation and algorithmic trading itself

(including a background on how techinical analysis can be used to generate trading rules). While there are no practical results presented, this work is a great starting point when delvign into the field of evolutionary computational finance.

Using a genetic algorithm approach, [7] generated stock trading strategies using two different approaches. Using a direct encoding (where parameter values for functions are encoded within the genome of an individual), this work realized a total profit of 1628 Japanese yen (JPY) throughout experimental trading. With a 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, [7] 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 now each parameter can have only 6 possible values, instead of 1000), and a profit of 2370 JPY was realized.

The main problem with the use of a genetic algorithm for stock trading rule generation is that the program maintains a fixed length. Genetic programming however, allows programs to be smaller/larger. 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). [10] used genetic programming 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 365 days, the profitability of the generated rules were compared to the profitability of a buy-and-hold strategy (buying the currency at day 1 and holding it until the end). 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 of 13.14% higher return.

One of the first and most often cited works on massively parallel genetic programming was completed by [4]. This work was implemented on a MasPar MP-2 machine which used a single intstruction multiple data (SIMD) architecture. However, a MIMD architecture was simulated using a stack based interpreter which accepts genetic program individuals as data. This way, single instruction architecture is met because each process core is executing the same step of the interpreter, using the individual programs as data inputs. Using this approach, many different individuals within the genetic programming population can 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 tradional CPU approach was not presented, the benchmark tests were capable of evaluating thousands of individual programs in approximately 1 second.

One of the original works involving genetic programming on a GPU device was completed in [2], and used a data parallel approach to GP. Instead of implementing a stack-based interpreter (as above), only one individual was evaluated at a time on the GPU. The parallelization was achieved by evaluating fitness cases for the current individual in parallel. The implementation was tested on several GP problems (including the classic symbolic regression and 11-way multiplexer problems), with 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 ran more than twice as quickly as the GPU approach. With 150 fitness cases however, evaluation time was nearly equal. When the number of fitness cases increased further, 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. The main conclusion here is that the performance of a data parallel GPU approach (when compared to a sequential CPU implementation) achieves greater speedup with a higher number of fitness cases. This is because a low number of fitness cases does not allow the computing power of the GPU to be optimally used. In fact, with a small enough number of fitness cases, the overhead of GPU evaluation results in the CPU implementation being faster.

Another data parallel GP approach was implemented on a GPU device in [3]. This time, 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 peformed, 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 only once (with all 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 floation point operations (+, -, *, /) and terminals. Speedup factors (when compared to the CPU implementation) ranged from 0.04 (for 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 for 0.02 for program length of 10 and 10 fitness cases, to 95.37 for program length of 10000 with 2000 fitness cases.

In [8] and [9] a population parallel approach to GP on GPU devices is implemented using the CUDA development kit from NVIDIA (similar approaches can also be found in [6] and [5]). As in [4], individuals are evaluated in parallel using a stack-based interpreter which accepts individual programs as implemented. Within this work, two aprroaches to evaluation distribution are presented. In the ThreadGP approach, each thread within the GPU is assigned to evaluate one individual, with each fitness case being evaluated on the thread. The BlockGP method takes advantage of the 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 approach then evaluates each individual on a single MP, with all theads within the MP being used to evaluate different fitness cases in parallel. Divergence is avoided using the BlockMP approach, since the multiprocessors are capable of executing different instructions of the single interpreter program. 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 because all must be at the same inscruction within the interpreter. This inefficiency is what caused the BlockGP approach to perform faster evaluation over all tests carrierd out. Tests were completed using a symbolic regression problem with different population size and number of fitness cases. The highest speedup of BlockGP was found for the combination of smallest number of individuals (512) and highest number of fitness cases (1024). ThreadGP performs poorly in this case becase 512 is the minimum number of threads that can be running on the GPU device at a time, so the GPU scheduler cannot swap out threads that are waiting for threads that are not. BlockGP on the other hand invovled 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 when compared to the ThreadGP approach. Furthermore, since BlockGP spreads fitness evaluations of an individual across 32 threads, a higher number of fitness cases allows 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, which results 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, ThreadGP will have 5 blocks operating per MP, which allows the scheudler to substitute waiting threads efficiently. BlockGP still performs faster due to the lack of divergence however.

## References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] W. Langdon and Wolfgang Banzhaf. A simd interpreter for genetic programming ongpugraphicscards. In *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85. Springer Berlin / Heidelberg, 2008.

[7] 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.

[8] 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.

[9] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, 2009.

[10] 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.