# Reconfigurable Acceleration of Fitness Evaluation in Trading Strategies

Andreea Ingrid Funie, Paul Grigoras, Pavel Burovskiy, Wayne Luk, Mark Salmon
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
Email: andreea.funie09@imperial.ac.uk

***Abstract*—Over the past years, examining financial markets has become a crucial part of both the trading and regulatory processes. Recently, genetic programs have been used to identify patterns in financial markets which may lead to more advanced trading strategies. We investigate the use of Field Programmable Gate Arrays to accelerate the evaluation of the fitness function which is an important kernel in genetic programming. Our pipelined design makes use of the massive amounts of parallelism available on chip to evaluate the fitness of multiple genetic programs simultaneously. An evaluation of our designs on both synthetic and historical market data shows that our implementation evaluates fitness function up to 21.56 times faster than a multi-threaded C++11 implementation running on two six-core Intel Xeon E5-2640 processors using OpenMP.**

## I. Introduction

Genetic programming is an important machine learning technique which can be used to identify patterns in the financial markets. Genetic programming involves repeatedly generating a set of programs, evaluating them on a large data set and selecting the best performing ones. The evaluation step computes a *fitness* metric for each program, based on which the best performing programs can be selected for the next iteration. The potentially complex programs and large data sets on which they need to be evaluated make fitness evaluation one of the most computationally expensive components in genetic programming. In some cases fitness evaluation may take up to $95\%$ of the total execution time. [1].

In finance, genetic programming can enable the recognition of complex market patterns and behaviours [2], but the high computational demands make it an unfeasible technique in the context of high-frequency markets. However, recent developments in hardware acceleration tools have enabled the efficient use of flexible run-time reconfigurable algorithms which are able to rapidly react to changing market conditions [3].

We propose to use Field Programmable Gate Arrays (FPGAs) to accelerate the fitness evaluation of a genetic program, enabling identification of complex data patterns such as those within Foreign Exchange data which could lead to more advanced trading strategies [4]. Our contributions are:

- A novel pipelined architecture for evaluating the fitness function of complete expression trees;
- Support for mixed-precision computation with both fixed point and single-precision floating-point arithmetic targeting Maxeler systems;
- Demonstration of the proposed approach with sets of both synthetic and real market data, showing a speedup of up to $21.56$ times when compared to an optimised 12-core CPU implementation.

## II. Background

### A. Genetic Programming

A "genetic program" (GP) is a search method that mimics the process of natural selection. This method is used to generate individuals by repeatedly mutating and recombining parts of the best currently known individuals for a number of different optimisation and search problems [5]. GP searches a population of candidate individuals in order to identify the best performing individual out of the entire population.

Our approach adopts generational genetic programming [6] which works as follows. Firstly, solutions are randomly generated to form an *initial population*; the individuals are represented as *trees*, where a leaf contains a terminal, and an internal node contains a function (operator) whose arity is equal to the number of its children. Secondly, part of the existing population is selected to create a *new generation*; usually, for this purpose a problem specific *fitness* metric is defined, and better performing individuals are more likely to be selected. Furthermore, a pair of "parent" individuals are selected and, using genetic operators such as *crossover*, *mutation* and *reproduction*, a new solution is created. This step is repeated until a new population of solutions of the desired size has been generated. Finally, the new generation replaces the old generation and the above steps are repeated. This generational process is repeated until it reaches a termination condition such as a solution is found that satisfies minimum criteria or a fixed number of generations have been reached [7].

Computing the fitness value of each individual is a central computation task of GP applications, usually consuming most of the overall computation time. Thus, the main effort to speedup a genetic programming application is focused on the fitness evaluation. Our study aims to use hardware acceleration techniques such as Field Programmable Gate Array technology in order to significantly reduce the fitness evaluation execution time and obtain a better overall execution time for a genetic programming application (see Figure 1).

### B. High Frequency Trading in the Foreign Exchange Market

Currency Trading is the world's largest market which consists of a daily volume of almost trillion data points. Foreign

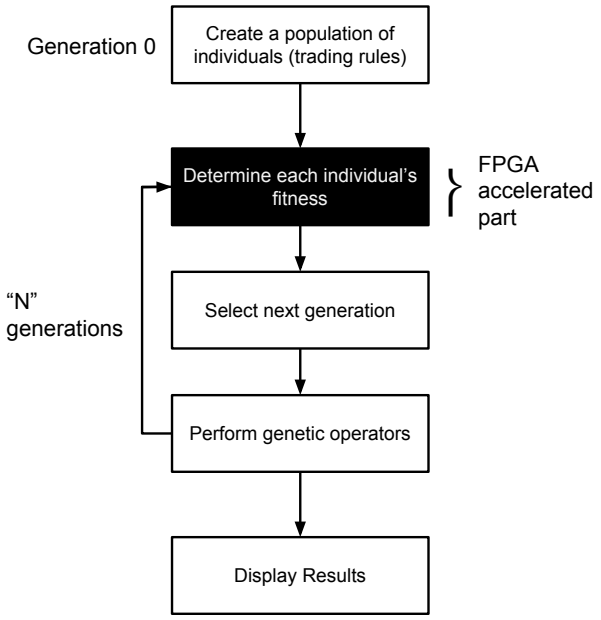


Fig. 1. Genetic Programming Algorithm

Exchange (FX) is considered to be the most liquid market in the world, thus differentiating itself from other financial markets [8]. There are four major currency pairs that are traded frequently. These include EUR/USD, USD/JPY, GBP/USD, and USD/CHF [9].

Some of the main characteristics of the Foreign Exchange Markets are: First, the market is tradable 24 hours a day, excluding weekends. Second, there is a high geographical dispersion thus giving rise to a variety of factors affecting exchange markets. Third, FX has a *huge trading volume* - the largest asset class in the world leading to high liquidity.

Foreign Exchange Markets, as well as other markets, are often suitable for *high-frequency trading* (HFT). Currently, HFT uses sophisticated tools and complex algorithms to trade on a rapid basis [10]. Banks and other institutions use HFT to apply proprietary trading strategies by a computer algorithm in real-time [11].

### C. Genetic Programming on FPGAs

Given the computational intensive nature of GP and close to real-time constraints on the execution time, a number of previous works have studied the use of FPGAs to accelerate various GP kernels.

Some of the previously proposed solutions show that a good speedup over software implementation is achievable but at the cost of restricting either the population size or the supported tree depth. For example [1] presents an interesting approach to a whole genetic programming implementation on FPGAs. The fitness evaluation targets a specific problem: having the trees represented by certain tree templates, thus not providing a flexible solution towards the overall GP approach. In that approach, the user would need to build different tree templates for different problems. This compares infavourably with our design in which the user has the freedom to build any complete binary tree with a range of given terminals and operators. The results obtained in the study presented in [1] include a 19 times speedup when performing an arithmetic intensive operation. This implementation is also limited to a significantly smaller population of 100 individuals, compared with our approach where up to 992 individuals are supported. [13] shows a different approach to a whole genetic programming implementation on FPGAs. This design supports a very small number of individuals, such as 8 or 16, with each individual tree being able to have a maximum depth of 2. Our approach does not have this limitation. [14] presents an interesting approach to implement an evolutionary algorithm for a hardware-based path planning architecture, created for unmanned aerial vehicle adaptation. This design proves to be highly efficient, managing to reach the 10 Hz update frequency of a typical autopilot system. However, the number of individuals evaluated at once is again limited to just 32.

Some solutions show that one other way in which we can obtain a significant speedup using FPGAs over the conventional software implementation is to restrict the type of the supported individuals. As an example of this, [12] implements an evolutionary computation coprocessor on FPGAs, in order to solve the iterated prisoners dilemma (IPD). The authors claim 200 times speedup in processing the IPD when compared to a 750 MHz Pentium processor. A limitation of this approach that we solve is the number and complexity restriction of the supported individuals (we support a flexible complete binary tree representation, while this paper enforces the use of bit-strings).

There are also a number of solutions available which try to accelerate genetic algorithms and some of them prove to be very successful. For example, [15] shows a good approach of implementing a parallel GA framework. The users can re-tune parameters at run-time and the authors claim a significant speedup of 26 times when compared to a optimised CPU version of the framework. This design deals with genetic algorithms, not genetic programs, thus trying to solve different complexity search problems.

## III. Design

We propose to exploit the high level of internal parallelism which can be achieved on FPGA based reconfigurable computers to accelerate fitness evaluation. In this section we describe a design which achieves the throughput rate of one data point per clock cycle. In Section IV we explain how our design can be extended to take advantage of larger commercial chips, where multiple parallel processing pipelines can be deployed concurrently to speed up the computation further. Our design targets a generic accelerator model:

- a CPU based system is connected to an FPGA accelerator;
- both CPU and FPGA have large on-board memory available;
- transfer speed from on-board memory is much faster than via the interconnect;
- all data are initially in on-board memory on the CPU;

- a substantial part of the computation is offloaded to the FPGA.

Figure 2 shows an example expression tree, which corresponds to a trading rule supported with the proposed design. These expression trees are generated on the CPU as part of the larger Genetic Programming Algorithm (Figure 1). They are then transferred to the FPGA where they are evaluated on a stream of past market data. On each market data tick, the outcome of the evaluation (true or false) indicates whether the algorithm should buy or sell stock at that particular point. For each trading rule, the fitness is computed as outlined in [16]:

$$R_c = \Pi_t(1 + z_t * r_t) - 1$$

where $r_t$ = ($p_t$ - $p_{t-1}$) / $p_{t-1}$ is the one-period return of the exchange rate and $p_t$ corresponds to either the bid or ask price, depending on whether the outcome of the evaluation is to buy or sell; $z_t$ takes the value 1 when buying and $-1$ when selling [17].

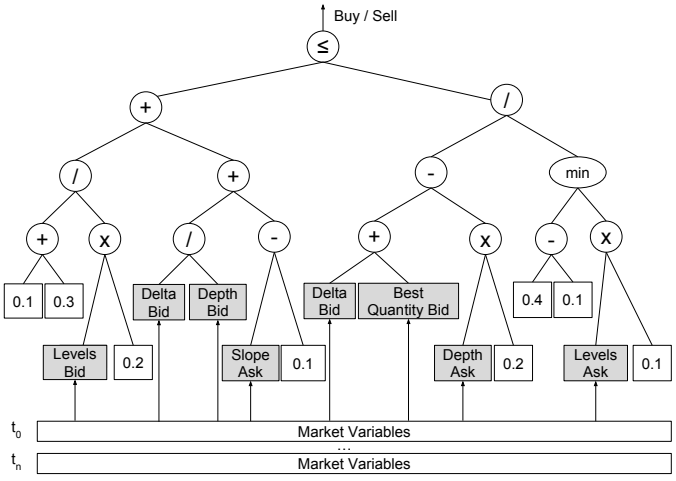


Fig. 2. Example expression tree for a trading rule. Terminals nodes are either market variables (shown in light gray) or constants. Internal nodes are binary arithmetic operators and the root node is a binary boolean operator.

To make fitness evaluation more amenable to an efficient hardware implementation, we make a number of assumptions in order to simplify the structure of the expression tree to be evaluated. First, we assume the expression tree is a complete, binary tree. This results in a well-defined connection pattern, which can be implemented efficiently on the FPGA and also entails that all internal nodes must be binary operators. Second, we restrict the set of internal arithmetic nodes, known as the GP function set, to any of the following operations: +, *, -, /, min, max. Although more complex expressions could be used, it is important to note that there is a trade-off between the complexity of the expression and resource usage. In general, the design should not consume a substantial number of resources for internal nodes which are not likely to be used in all expressions. Third, the root node must be a boolean operator, since the output of the evaluation must always be true or false. Supported operators are $\leq$ and $\geq$. The terminal nodes of the expression trees can be either constants (which are streamed from the CPU along with the expression) or market variables. The value of market variables may change in each time step. The number of market variables is arbitrary, but since market data are read from on-board memory on every clock cycles, it may be necessary to limit their number. Both constants and market values are single precision floating point numbers. In practice, we found that 16 variables provide sufficient basis to derive interesting trading strategies, although our design could support much more (up to 96) on large commercially available chips.

It is important to note that under these assumptions, it is possible to derive good trading strategies, based on market data. In Section V we show that some of the supported strategies are profitable, by evaluating them on past Foreign Exchange market data and performing appropriate financial tests. Furthermore, we note that the degree of profitability increases with the number of iterations of the genetic algorithm, which makes hardware acceleration even more important to reduce the overall computation time.

### A. Overview

The design is organised in multiple Processing Elements (PEs). Arithmetic Processing Elements (APEs) are used to evaluate the internal nodes of the tree. Terminal Processing Elements (TPEs) are used to resolve the access to either market variables or provided constants. The Root Processing Element (RPE) provides binary boolean operators. Since the design only accepts complete binary expression trees, the PEs are arranged in a binary tree structure. The depth of the tree ($T_{depth}$) is a design parameter. Figure 3 shows an example of an architecture for $T_{depth} = 4$, which could be used to evaluate the expression shown in Figure 2. There are in total 16 TPEs, 14 APEs (partially omitted for clarity) and one RPE.

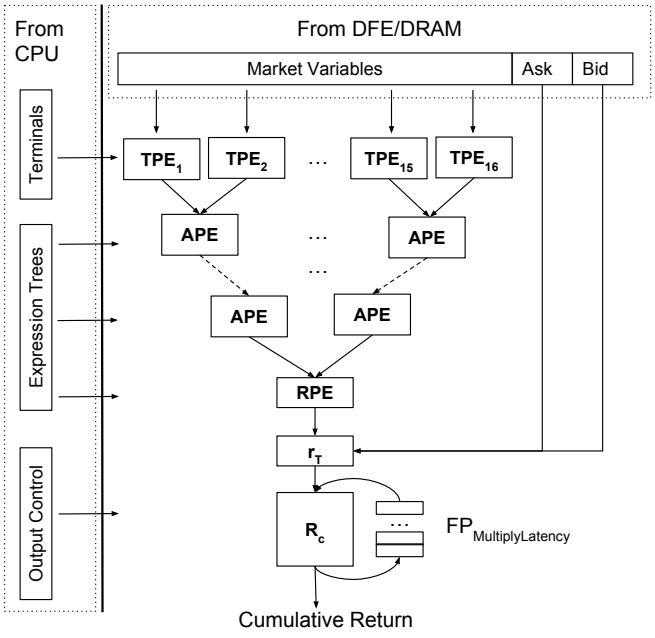


Fig. 3. Architecture Diagram for $T_{depth} = 4$. There are in total 16 TPEs, 14 APEs (partially omitted for clarity) and one RPE.

The result of the expression evaluation is used to decide whether to buy or sell the current instrument. Correspondingly,

either the bid or ask price for the current time step is used to compute the expected return of the action. This is done inside the $r_T$ block.

The return must then be accumulated across all market ticks. Performing partial accumulation on the FPGA, before sending the results back to the CPU, reduces traffic over the slow interconnect, and also reduces the volume of work required on the CPU. We accumulate the fitness values into partial values using a feedback multiplier. The number of partial values is equal to the latency of the feedback loop $FP_{MultLatency}$. Increasing the latency (in cycles) results in a more pipelined implementation of the floating point multiplier which in turn enables a higher maximum clock frequency. However, this also increases the amount of partial sums to be transferred back to the CPU and the amount of work to reduce these partial sums. In practice we find that a value of 16 cycles is sufficient to enable good clock frequency with small impact on the transfer and CPU reduction time.

The output control signal is used to enable the output to the CPU and is high only on the last $FP_{MultLatency}$ cycles of processing an expression.

### B. Processing Elements

**Terminal Processing Element (TPE)** The TPE is used to process expression terminals. These can be either constants or indices into the market variables read from DRAM. By convention we interpret values in $[0, 1)$ to be constants and values above (or equal to) 1 to be indices. Since the terminals are streamed from the CPU as floating point values, an additional cast to an integer is required for values which are to be used as indices. An index is then used to control a 16 input multiplexer to select the correct market variable.

**Arithmetic Processing Elements (APEs)** The APEs implement binary arithmetic functions. Their inputs are two real numbers (either from TPEs or from APE on a previous layer). Their output is also a real number.

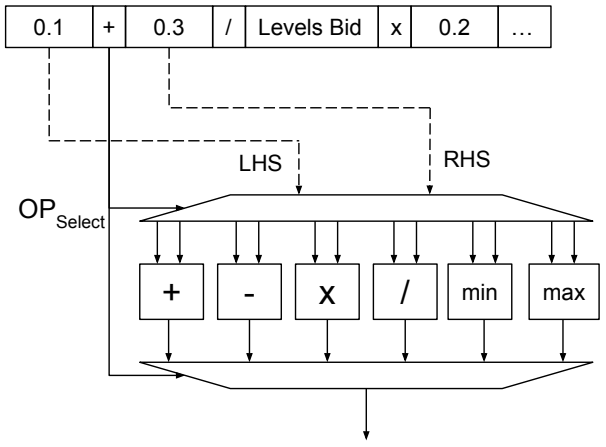


Fig. 4. Arithmetic Processing Element

The structure of an APE is shown in Figure 4. Operators in the current expression to be evaluated are decoded into an $Op_{select}$ signal. To simplify decoding, operator codes for arithmetic operations are integers starting from 0. A demultiplexer is required to route the left hand side (LHS) and right hand side (RHS) operands to the correct arithmetic unit. A multiplexer is used to select the output from the correct arithmetic unit and forward it to the next level of the tree.

**Root Processing Element (RPE)** There is a special root processing element evaluating comparison operators such as: $\leq, \geq$. The RPE has real numbers as inputs and boolean output, which ensures the output of the algorithm is boolean. The result produced by the RPE is used in the return evaluation to decide whether to buy or sell the particular instrument.

### C. Operation

Most accelerator configurations have a reduced bandwidth between the host CPU and the accelerator card in comparison to the memory bandwith. This is true for systems using PCIe, Inifiniband or Ethernet network as the transport layer.

We note that in the proposed Algorithm, market values, bid and ask prices will be reused for each expression that is evaluated. As such these will be stored in DRAM and only incur the transfer penalty over the slow interconnect between the CPU and FPGA once.

In contrast, the expressions to be evaluated are only loaded once. Therefore there is no need to store them in on-board DRAM. Instead, the expression trees and terminals are streamed over the CPU/FPGA interconnect. Since this interconnect is not likely to deliver data at a rate which allows one full tree and the operators to be read in one clock cycle, a BRAM buffer is used to store expressions and operators. This allows the design to only pay the large transfer penalty once: while the evaluation of the current expression progresses, the design can fetch the following expression to be evaluated (and terminals) in the background, at effectively no additional cost.

The operation of the design can be summarised as follows:

1) Load market data to accelerator DRAM;
2) Queue expression trees from CPU to FPGA BRAM;
3) Evaluate expression on market data;
4) Fetch next expression to FPGA BRAM;
5) Output partial results to CPU;
6) Repeat the above steps until done.

We assume all data, including market data and generated expressions, are initially in CPU memory - as part of the genetic programming algorithm. In this case, the time to initialise the design is the sum of the initial latency of the interconnect, the time to load the market data into accelerator DRAM and also the number of clock cycles required to load the initial expression into the on-chip expression buffer. For large problem sizes, this initial cost can be ignored.

After the initialisation the design is memory bound if the number of values which can be read from DRAM per clock cycle is smaller than the number of market variables in the TPEs.

Otherwise, assuming the design is compute bound in the limit, the compute time can be estimated by

$$T_{Total} = \frac{N_{Expr} N_{Ticks}}{F} \tag{1}$$

where $N_{Expr}$ is the total number of expressions to be evaluated, $N_{Ticks}$ is the number of market ticks to evaluate each expression tree and $F$ is the FPGA clock frequency.

## IV. Implementation

The implementation of the proposed design depends heavily on the properties of the target system. Our implementation targets a Maxeler MPCX node, which contains a Maia dataflow engine (DFE) with 48 GB of on board DRAM.

### A. Input/Output

The DFE is optimised for relatively large transfer bursts from on-board DRAM (384 Bytes). Maximum efficiency of DRAM transfer is achieved when more bursts are read in a linear access pattern. We note that this is the case for the market data variables in our problem, which are all read from DRAM in a sequential fashion.

In this situation, we can read up to 1536 bits per clock cycle from DRAM and an additional 128 bits per clock cycle from Infiniband. Our design makes use of both DRAM and the Infiniband interconnect. As a result, the design is compute bound, which is ideal for the FPGA.

Since market data variables are single precision floating point values (32 bits wide), we could read up to $1536/32 = 48$ different market variables from on-board DRAM without causing the design to become memory bound. This is well within the limits of our problem. We note that this value could, if needed, be increased even further by doubling the clock frequency of the memory controller from the default value of 400 MHz to 800 MHz. However in practice this results in higher resource usage since more pipelining is required to enable timing closure, and in longer compilation times. Since we use only 16 distinct market variables in our problem, we can use the default memory controller frequency.

### B. Parallelisation

The DFE has a large commercial FPGA chip, the Stratix V 5SGSMD8N1F45C2. This enables us to further improve the performance of the proposed design by implementing multiple parallel processing pipelines on-chip; we refer to these simply as *pipes*.

To parellelise the design, two options are possible:

- read more data points per clock cycle and evaluate the same tree on multiple ticks simultaneously;
- read more trees and evaluate just one data point on all trees in parallel.

The first option requires much higher DRAM bandwidth since DRAM bandwidth increases linearly with the number of pipes, and, depending on how many pipes can fit on the FPGA, the design may become memory bound before it is resource bound. Since in practice we observe that we can fit up to 8 pipes and possibly beyond, we do not consider this approach feasible.

The second option scales much better, since it does not increase DRAM traffic. It does however increase Infiniband traffic linearly. As explained in the previous section, the impact on overall latency is negligible. First, subsequent expressions are cached in on-chip BRAM while the current expressions are being evaluated, effectively hiding the latency of transferring the next expression over Infiniband. Second, we can also reduce the transfer size of the operators, by encoding them in smaller bit widths. In theory, since we support only 6 functions for the APEs, we could require as few as 4 bits to encode each operator. However, in practice, we find it easier to use 8 bits – this achieves sufficient savings to avoid making the design bounded by the Interconnect transfer bandwidth, while still being easy to implement on the CPU since 8 bit wide values are part of standard C++.

### C. Mixed precision

We implement mixed precision numerical computation to reduce on-chip resource consumption and improve design scalability. We split the computational flow into a full precision floating point part and a fixed point part. We provide single precision implementation of APEs for comparison. Table IV summarizes the logic utilisation for the single precision floating point implementation, while in Table V we can notice the resource utilisation for the fixed point precision version.

**Precision analysis** shows that market inputs belong to the interval $(1, 2)$ with only 4 significant digits. For division operations the dynamical range is constrained to $10^{-4}, ..., 10^{4}$, which is well covered by 32 bit fixed point representation. An important observation is that tree expressions are evaluated at independent inputs, so there is no round-off error accumulation associated with reduced precision.

Another numerical part of the design, the accumulation of returns and computation of current stock ($r_T$) is more sensitive to round-off error accumulation and thus implemented in floating point. However this part of the design has smaller impact on design scalability due to a lower amount of arithmetic operations.

We store market data in DRAM in single precision and convert it to a fixed point format on-chip as part of a hardware pipeline. These fixed point numbers form inputs to fixed point APEs, which provide the boolean output to choose between buy or sell choice.

**Dealing with division by zero.** The market data and terminal constants are guaranteed to be nonzero numbers. However a cancellation of terms may occur within expression trees, resulting in division by zero or a very small number. In order to avoid making marketing decisions based on numerically unstable arithmetic, we check whether a divisor is greater than $tol = 10^{-4}$ at any sample of the training set. Our APEs compute both the resulting expressions as well as validity flags, compensating for lack of *infinity* and *NaN* values in fixed point representation. Once expression evaluation yields invalid output, we effectively invalidate the whole tree expression and prune it from the GP population.

## V. Evaluation

The accelerator system we use is a Maxeler MPCX node. The system properties are summarised in Table I. It consists of a CPU node and a DFE node. The two are connected via Infiniband through a Mellanox FDR Infiniband switch.

TABLE I. SYSTEM PROPERTIES

| | |
|---|---|
| **CPU** | Dual Intel Xeon E5-2640, 6 cores per CPU |
| **CPU Cache** | 15 MB |
| **CPU DRAM** | 64GB DDR3-1333 |
| **CPU DRAM Bandwidth** | 42.6 GB/s (Peak) |
| **FPGA** | Stratix V 5SGSMD8N1F45C2 |
| **FPGA DRAM** | 48 GB |
| **FPGA DRAM BANDWIDTH** | 38 GB/s (Achieved) |
| **CPU to FPGA BANDWIDTH** | 2 GB/s |

### A. CPU Implementation

The CPU implementation is built using C++11 and parallelised using OpenMP. We compile the CPU implementation using g++ 4.9.2 with flags `-O3 -march=native -fopenmp` to enable general performance optimisations, architectural optimisations for the Intel XEON (such as generation of vectorised instructions) and the use of multithreading.

The CPU code is parallelised in a similar manner to the hardware implementation: each core is assigned one expression which it executes to measure the fitness of the entire data set. All the CPU times shown are obtained using a total of 12 threads (2 x Intel Xeon, with HyperThreading disabled to provide more accurate timing measurements and scalability information).

Table II shows the scalability of our CPU implementation with the number of threads. Since HyperThreading may cause the CPU implementation to scale sub-linearly with the number of threads, we disable it on the CPU node and only use 6 threads per CPU (for a total of 12 threads). Table II shows close to linear scaling (as expected) for the CPU implementation, when tested on 19.2M ticks and 992 expressions. The 19.2M ticks represent 16 days of high-frequency-trading activity for the GBP/USD market during different time periods. This is to be expected given that our parallelisation strategy requires minimal communication between threads and therefore, for large problem sizes, the computation times dominate.

TABLE II. CPU SCALABILITY RESULTS SHOW LINEAR SCALING FOR UP TO 12 THREADS.

| **# Threads** | 1 | 2 | 4 | 8 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| **CPU Comp. Time** (s) | 248.139 | 125.873 | 62.931 | 31.497 | 25.470 | 23.020 | 21.396 |
| **Speedup** | 1X | 1.97 | 3.94 | 7.88 | 9.74 | 10.78 | 11.60 |

In the C++ code the tree depth is marked as a constant which allows the compiler to unroll the expression evaluation loops and to resolve some computations at compile time to achieve better performance.

All run times are measured using the `chrono::high_resolution_clock` which is part of the C++11 standard library.

### B. FPGA Implementation

Table III shows the speedup achieved by our implementation, with respect to the previously described CPU implementation. The numbers shown correspond to the fixed point version of our design, but we note that for the same number of pipes, both single precision and fixed precision implementation have the same performance. We note that for the *fixed point* version it is possible to fit more parallel pipes on chip, which leads to a larger speedup. The FPGA execution does not include the time to load the initial market data in the FPGA DRAM (market variables, ask and bid prices) and reconfiguration overhead. It does include the time to stream market data from DRAM during expression evaluation, the time to queue and stream the expressions and the time to accumulate the partially evaluated sums on the CPU along with any other overhead incurred by the framework. The numbers shown correspond to a run over 3.84M ticks and 992 expressions. The performance evaluation has been performed on a synthetic benchmark, which contains randomly generated expressions, that comply with the assumptions presented in Section III. Table III also shows that for large problem sizes the estimated compute times (estimated by using Equation 1, with $N_{Ticks} = 3.84\text{M}$, $N_{Expr} = 992$, $F = 190MHz$) closely matched the observed execution times. This confirms that the design is compute bound as predicted.

The speedup obtained and presented in table III represents the speedup for the fitness evaluation acceleration only. If we are to take into account research that claims that the fitness evaluation part of an evolutionary algorithm can take up to 95% of the total execution time of the algorithms, then our best overall algorithm speedup would be 20.9340 times.

TABLE III. FPGA SPEEDUP RESULTS COMPARED TO 12 CPU THREADS.

| **# Pipes** | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| **Average CPU Time** | 50.4203 | 50.4203 | 50.4203 | 50.4203 |
| **FPGA Time (s)** | 20.0782 | 10.0532 | 5.04229 | 2.5353 |
| **Est. Speedup** | 2.47835 | 5.02975 | 9.81301 | 20.1063 |
| **Speedup** | **2.5111** | **5.0153** | **9.9994** | **19.8873** |

Figure 5 shows that the achieved speedup increases with the number of expressions and ticks that are processed as the impact of the initial transfer latency diminishes with respect to the overall execution time. This shows that the proposed approach works better for larger problem sizes where acceleration is most needed.

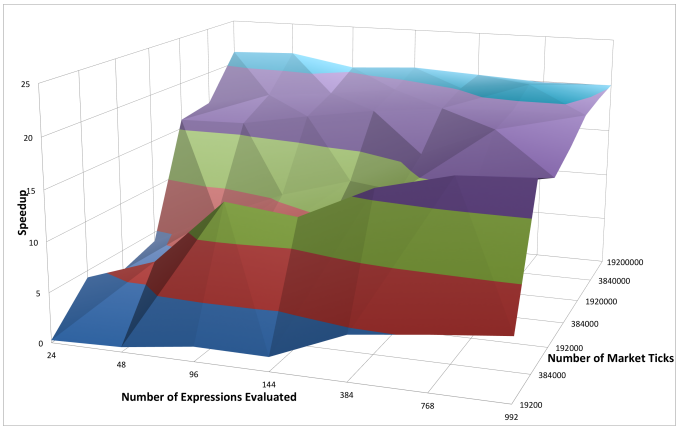


Fig. 5. Measured Speedup for Fixed Point Implementation

Table IV shows the total resource usage as a percentage of the total available resources on chip for the *single precision floating point* implementation based on 1 pipe and 4 pipes. We note that almost 10% of logic resources and 15% of BRAMs are used for the DRAM controller (since the Maxeler DFE uses a soft memory controller) and the Infiniband interface as

well as other overhead, including the expression buffer used to buffer the expressions on chip, which was described previously. We note that, in order to reduce logic usage substantially and fit more pipes on chip we have reduced the amount of pipelining used by default for floating point and/or fix point arithmetic units. The achieved clock frequency for our single precision floating point implementation is 200MHz.

TABLE IV. FPGA TOTAL RESOURCE USAGE FOR SINGLE PRECISION FLOATING POINT ARITHMETIC

| # of Pipes | LUTs | FFs | BRAMs | DSPs | of use |
|---|---|---|---|---|---|
| 1 | 10.69% | 7.49% | 15.82% | 0.00% | used by manager |
| 1 | 5.56% | 3.97% | 7.40% | 4.99% | used by kernels |
| 1 | 16.42% | 11.64% | 23.72% | 4.99% | total resources used |
| 4 | 10.86% | 7.61% | 17.65% | 0.00% | used by manager |
| 4 | 22.15% | 15.70% | 29.10% | 19.97% | used by kernels |
| 4 | 33.21% | 23.49% | 47.25% | 19.97% | total resources used |

Table V presents the FPGA total resource usage expressed as a percentage of the total available resource on the chip for the *fixed point precision* implementation based on 1 pipe and 8 pipes.

TABLE V. FPGA TOTAL RESOURCE USAGE FOR FIXED POINT ARITHMETIC

| # of Pipes | LUTs | FFs | BRAMs | DSPs | of use |
|---|---|---|---|---|---|
| 1 | 10.76% | 7.61% | 16.21% | 0.00% | used by manager |
| 1 | 7.40% | 4.35% | 3.12% | 1.88% | used by kernels |
| 1 | 18.33% | 12.14% | 19.83% | 1.88% | total resources used |
| 8 | 10.95% | 8.05% | 22.32% | 0.00% | used by manager |
| 8 | 51.49% | 30.98% | 22.52% | 15.08% | used by kernels |
| 8 | 62.61% | 39.21% | 45.34% | 15.08% | total resources used |

### C. Financial Tests and Results

We verify the applicability of trading strategies which can be supported using the proposed approach by evaluating expressions that conform to the assumptions set out in Section III. Historical GBP/USD tick-data from the Foreign Exchange Market corresponds to time-periods from 2003 and 2008.

*1) Individual Returns:* Table VI shows the daily returns of the best fit trading strategy for a different number of expressions (X) used in the GP, after $10,000$ iterations. The results show a clear decrease in the return levels from 2003 and 2008, this might indicate greater FX market efficiency in 2008 due to the growth in electronic high-frequency trading that occurred during the 2003 - 2008 period. This shows that we can use supported trading strategies to identify underlying characteristics of the financial market (e.g. market efficiency).

TABLE VI. 2003–2008 GP INDIVIDUAL RETURNS

| X | Jan(20-24) '03 | Feb(17-21) '03 | March(10-14) '03 | March 31 '08 |
|---|---|---|---|---|
| 992 | 1.278 | 1.188 | 1.103 | 1.076 |
| 768 | 1.047 | 1.024 | 0.998 | 0.937 |
| 384 | 0. 904 | 0.856 | 0.889 | 0.793 |
| 144 | 0.789 | 0.683 | 0.654 | 0.578 |

TABLE VII. 2003–2008 GP INDIVIDUAL RETURNS

| Work | X | Jan(20-24) '03 | Feb(17-21) '03 | March(10-14) '03 | March 31 '08 |
|---|---|---|---|---|---|
| [16] | 150 | 1.142 | 1.094 | 1.003 | 0.991 |
| This paper | 144 | 0.789 | 0.683 | 0.654 | 0.578 |

As seen in Table VII, comparing the same number of iterations/individuals results in inferior performance due to the reduced tree depth of our implementation. Table VI shows that for a smaller tree depth but a higher number of iterations/individuals than in [16] our solution is able to produce better overall returns due to the immense performance of the FPGA. Such performance allows us to evaluate 10 times as many iterations across 6.5 times as many individuals with a faster fitness evaluation approach than the CPU based equivalent. Hence, even with a smaller tree depth and less complex strategies, overall performance is preferable. Additional future enhancement will enable us to increase tree depth, thus improving performance further.

Figure 6 shows that ensuring genetic diversity throughout our genetic program, by increasing the number of expressions and the number of market data we evaluate them on our expressions on, shows promising results giving rise to potentially better performing trading strategies. Hence, we demonstrate that with the FPGA's computational acceleration power, we are now able to explore historical market prices without needing to worry too much about the reduced computation time that a GP often brings into place. Being able to evaluate trading strategies quickly, brings us closer to obtaining significant market feedback in reasonable time.

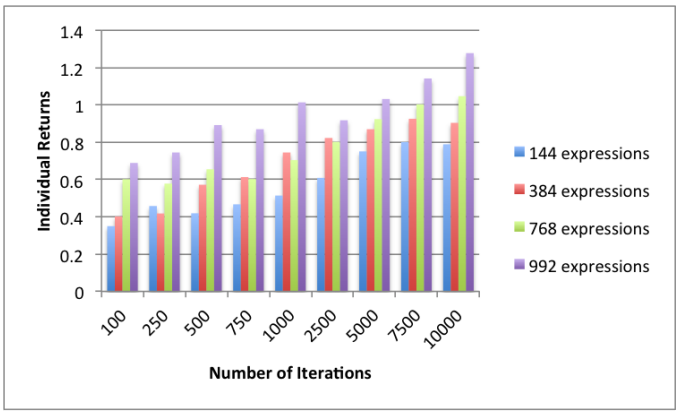


Fig. 6. Individual Returns

*2) Anatolyev-Gerko Tests:* Table VIII shows the results of the Anatolyev-Gerko Test (AG) for a range of expressions (X) after 10,000 iterations. This test statistically compares return levels based on conditioning information with those that might randomly occur within the correct distribution. For the test data, returns are adjusted to a daily basis, taking into account the transaction costs reflected in the bid-ask spread. For efficiency, we apply the "majority" rule in which 99 independent runs of 10,000 iterations are conducted. If the majority of these best in-sample trading strategies produce a buy/sell signal, then our rule will produce the same signal. The resulting trading rule will then be tested out-of-sample. In order to perform the AG test we need to make use of the data in order to obtain the real returns values. For the test, real return values are obtained from the data, while predicted return values are calculated according to the "majority" rule.

We notice that our test results are statistically significant, so despite the apparent predictability shown in Table VI, our algorithm does not deliver good profitability in 2008

TABLE VIII. 2003-2008 GP AG TEST

| X | Jan (20-24) 2003 | Feb (17-21) 2003 | March (10-14) 2003 | March 31, 2008 |
|---|---|---|---|---|
| 992 | 1.82 | 1.52 | 1.73 | - 0.37 |
| 768 | 1.56 | 1.63 | 1.22 | + 0.46 |
| 384 | 1.20 | 1.17 | 0.93 | - 0.66 |
| 144 | 0.84 | 0.92 | 0.44 | - 1.02 |

compared to 2003. This confirms our earlier observation that the efficiency of FX markets significantly increases in the 2003-2008 period.

*3) T-statistics Tests:* Table IX shows the results of the T-Statistics test for a different number of expressions (X), after $10,000$ iterations. This test allows us to examine if the mean of the returns is significantly different from zero. Table IX shows that these results do not appear to be significant at a $5\%$ level, however they show what looks like a systematic predictable pattern for the 2003 data set, but no pattern for the 2008 one. This further confirms the AG test results, showing increases in the FX markets efficiency potentially due to high frequency trading. This shows that we can use our resulting trading strategies not only for increased profitability, but to identify different market behaviours in different regimes, as well as to identify potential trading rules which could cause significant market changes (e.g a sudden GBP/USD price drop due to a certain market condition).

TABLE IX. 2003-2008 GP T-STATISTICS

| X | Jan(20-24) '03 | Feb(17-21) '03 | March(10-14) '03 | March 31 '08 |
|---|---|---|---|---|
| 992 | 1.625 | 0.801 | 0.543 | 0.463 |
| 768 | 1.608 | 0.768 | 0.477 | 0.494 |
| 384 | 1.486 | 0.698 | 0.312 | 0.395 |
| 144 | 1.583 | 0.606 | 0.301 | 0.434 |

## VI. CONCLUSION

We show that FPGAs can effectively accelerate genetic programming approaches used to identify and evaluate high-frequency trading strategies. We demonstrate that one of the most computationally intensive tasks associated with this process, fitness evaluation, can be accelerated substantially by exploiting the massive amounts of on-chip parallelism available on commercial FPGA chips. Our single precision floating point and fixed precision implementations are up to 11.25 and 21.56 times faster respectively than a corresponding multi-threaded C++11 implementation running on two six-core Intel Xeon E5-2640 processors. An evaluation on historical Foreign Exchange market data shows that trading strategies supported by the proposed design are reliable and, if further exploited, can increase profitability from trading high frequency FX markets.

Future work opportunities include extending the GP alphabet and increasing the maximum supported depth for expression trees. We will also look into adding support for more flexible strategies that could potentially be built using unbalanced trees. These improvements could lead to more profitable trading strategies as outlined in [16]. Given the nature of the problem, real-time/time-constrained tests will be performed in order to provide more useful insights into the performance of our tool. The spare on-chip resources required for these extensions could be provisioned by using runtime reconfiguration [18] to remove expensive functions when they are not being used (e.g. division). In addition, resulting spare resources could be used to increase the level of parallelism, further improving overall performance when necessary.

## VII. ACKNOWLEDGEMENT

This work is supported in part by the United Kingdom Engineering and Physical Sciences Research Council, by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by the Maxeler University Programme, and by Altera.

## REFERENCES

[1] R. P. S. Sidhu, A. Mei, and V. K. Prasanna, "Genetic Programming using Self-Reconfigurable FPGAs", in *Field Programmable Logic and Applications - Lecture Notes in Computer Science* , Vol. 1673, pp. 301-312, 1999.

[2] J. Y. Potvin, P. Soriano, M. Vallee, "Generating trading rules on the stock markets with genetic programming", in *Computers & Operations Research*, Vol. 31, pp. 1033-1047, 2004.

[3] S. Wray, W. Luk, P. Pietzuch, "Exploring algorithmic trading in reconfigurable hardware", in *International Conference on Application-specific Systems, Architectures and Processors*, pp. 325-328, 2010.

[4] Q. Jin, D.B. Thomas and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models", in *Internation Conference on Field Programmable Custom Computing Machines*, pp. 73-78, 2009.

[5] P. Nordin et. al., "Genetic Programming: An Introduction", 2013.

[6] W. B. Langdon and R. Poli, "Foundations of Genetic Programming", Springer, 2002.

[7] D. A. Coley, "An Introduction to Genetic Algorithms for Scientists and Engineers", World Scientific, 1999.

[8] T. Weithers, "An Introduction to Forex Trading - A Guide for Beginners", Wiley, 2013.

[9] M. Driver, "Foreign Exchange: A Practical Guide to the FX Markets", in *CreateSpace Independent Publishing Platform*, 2012.

[10] H. Zhang, R. Ren, "High Frequency Foreign Exchange Trading Strategies Based on Genetic Algorithms", in *International Conference on Networks Security Wireless Communications and Trusted Computing (NSWCTC)*, Vol.2, 2010.

[11] M.A.H. Dempster and C.M. Jones , "A real-time adaptive trading system using genetic programming", in *Topics in Quantitive Finance*, 2000.

[12] Y. Yamaguchi, A. Miyashita, T. Marutama, and T. Hoshino, "A co-processor system with a Virtex FPGA for evolutionary computations", in *10th Int. Conf. Field Programmable Logic Appl. (FPL2000)*, Lecture Notes in Computer Science, Vol. 1896, Springer, pp. 240-249, 2000.

[13] P. Martin, "A Hardware Implementation of a Genetic Programming System Using FPGAs and Handel-C—", in *Genetic Programming and Evolvable Machines*, Vol. 2, Springer, pp. 317-343, 2001.

[14] J. Kok, L. F. Gonzalez, and N. Kelson, "FPGA Implementation of an Evolutionary Algorithm for Autonomous Unmanned Aerial Vehicle On-Board Path Planning", in *IEEE Transactions on Evolutionary Computation*, Vol.17, No. 2, 2013.

[15] L. Guo, D. Thomas, C. Guo, W. Luk, "Automated Framework for FPGA-Based Parallel Genetic Algorithms", in *Field Programmable Logic and Applications (FPL)*, 2014.

[16] A. I. Funie, M. Salmon and W. Luk, "A Hybrid Genetic-Programming Swarm-Optimisation Approach for Examining the Nature and Stability of High Frequency Trading Strategies", in *13th International Conference on Machine Learning and Applications (ICMLA)*, pp. 29-34, 2014.

[17] R. Kozhan and M. Salmon, "The information content of a limit order book: The case of an fx market", in *Journal of Financial Markets*, pp. 1-28, 2012.

[18] X. Niu, T.C.P. Chau, J. Qiwei, W. Luk, L. Qiang, "Automating Elimination of Idle Functions by Run-Time Reconfiguration", in *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 97-104, 2013.