CUDA-enabled Optimisation of Technical Analysis Parameters

John O'Rourke (Allied Irish Banks)
School of Science and Computing
Institute of Technology, Tallaght
Dublin 24, Ireland
Email: John.ORourke@ittdublin.ie

Dr. John Burns
School of Science and Computing
Institute of Technology, Tallaght
Dublin 24, Ireland
Email: John.Burns@ittdublin.ie

***Abstract*—The optimisation of Technical Trading parameters is a computationally intensive exercise. Models comprising a modest number of Technical Indicators require many thousands of simulations to be executed over a sample period of data, with the best performing sets of parameters employed to generate future trading signals. The purpose of this research is to investigate the suitability of GPU Computing for running the simulations in parallel and to develop a working Prototype optimiser based on the CUDA architecture. The cumulative nature of Profit and Loss over a sample period is a restricting factor in the design of a data-parallel trading simulator. Thus, different approaches to the distribution of the parallel workload are researched and an appropriate design for the Prototype is derived. Past studies are examined, including parallel Genetic Programming implementations. The remarkable speedups enjoyed by the Prototype are discussed in detail and a number of key design strategies are proposed. These include a per-thread solution identification methodology, a modification to Welford's Standard Deviation algorithm which results in the avoidance of divergent threads, and a suitable parameter distribution policy.**

I. Introduction

In this research, the suitability of employing CUDA technologies for the area of financial Technical Analysis is investigated and a Prototype CUDA-enabled optimiser is developed. Technical Analysis refers to the practice of examining the historical data of a trading instrument (such as a stock or commodity) to identify patterns in the daily price fluctuations. The emergent patterns are used to optimise (or "tune") the parameters of trading models. The optimised models are then applied to current data and generate signals which instruct a trader how to react to daily price changes. In order to optimise parameters, many thousands of trading simulations must be executed against the historical data. The possibility of parallelising this task in a cost-effective way is attractive; complex optimisations typically take many hours to complete and require extensive computing resources.

Section II introduces the Technical Trading concepts which are used in this research, discusses trading model parameter optimisation, and outlines ways in which CUDA may help this process. The design decisions (resulting from literature review) for the Prototype optimiser are described in section III. Section IV outlines the remarkable speedups enjoyed by the Prototype and proposes an optimal design strategy.

II. Technical Analysis

Stock traders employ a range of indicators or models to perform statistical tests on historical data which are designed to generate appropriate trading signals based on patterns of supply and demand. By optimising a model's parameters, the analyst can fit a version of the model to the past data which would generate trading signals yielding an optimum return over the period. The optimised model is then applied to current daily data in order to generate trading instructions which should, in theory, yield returns in the future. The premise of Technical Analysis is that prices tend to form repeating and identifiable patterns over time and that daily closing prices reflect all relevant factors including Fundamental data and investor psychology.

Technical Indicators are mathematical rules which are applied to market data and instruct a trader how to behave under current market conditions. Models comprising of one or more such formulae are usually designed to identify trends in a sample data set. By applying a model to historical data and systematically tuning its parameters to yield the best return over the sample period, an optimised model emerges which should suggest successful trading signals based on current price movements.

This paper explores two of the most popular trend following models; Simple Moving Average Crossover and Channel Breakout. In addition, a filtering method is employed to reduce spurious trading signals during volatile periods. Employing high performance GPU technology to optimising the combined parameters over a portfolio of trading instruments forms the core thrust of this research.

A. Moving Average Crossover

In this model, trends are identified and buy/sell indicators are generated by a pair of moving averages over the instrument's daily closing prices. The model typically has two parameters; the long-period and the short-period (i.e., the number of closing prices used to compute the longer and shorter term moving averages). When the short-term moving average curve crosses above the long-term moving average curve it is considered that an upward trend is initiated and the market is bullish, thereby generating buy signals. Conversely, when the short-term moving average curve crosses below the

long-term, there is a downward market trend and sell signals are generated.

Figure 1 shows a short term and long term simple moving average calculated over the price of crude oil in 2010. The durations are 5 and 100 days respectively. The short term crosses above the long term at point A at a price of $91, and crosses back below it at point B. According to the model rules, a buy signal is generated at point A during the upward trend in the market. A trader will take a long (buy) position, and realise daily profit and loss (P&L) during this period. A sell signal is generated at point B during the downward trend and the trader will take a short position. An upward trend is identified again at point C where a buy signal is generated for a price of $88. If the duration of the moving averages and changed, the intersections and thus the trading signals (and corresponding prices) are generated at different places. To optimise this model over the sample period, values must be found for the two parameters (i.e., the short and long term moving average durations) which yield the best possible return in P&L.

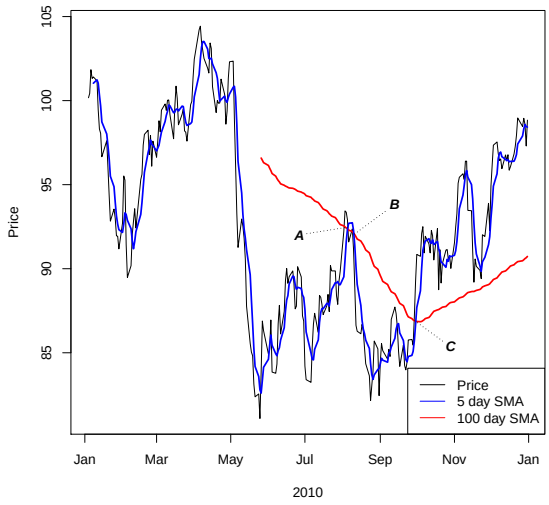

Fig. 1. Simple Moving Average Crossover over Crude Oil price in 2010. Short term is 5 days and long term is 100 days.

B. Channel Breakout

The channel breakout is a trend following model which tracks the highest and lowest price over the past number of days. The highest price occurring over the days forms an upper bound called the resistance level. The lowest price over the same period forms a lower bound called the support level. The model generates a buy signal when the current price rises above the resistance level and a sell signal when the price drops below the support level. The only parameter in this model is the number of days which should be included in the look-back period (i.e., the channel length).

Figure 2 illustrates a 60 day channel breakout model covering the same data sample and period as described in section II-A. The channel is based on closing prices during July to September (2010). If the price level breaks out of the channel formed by these upper and lower bounds, an upward or downward trend is identified and a signal is generated to buy or sell at current price. Optimising the channel breakout model involves identifying the number of back days which yields the best possible return over the sample period.

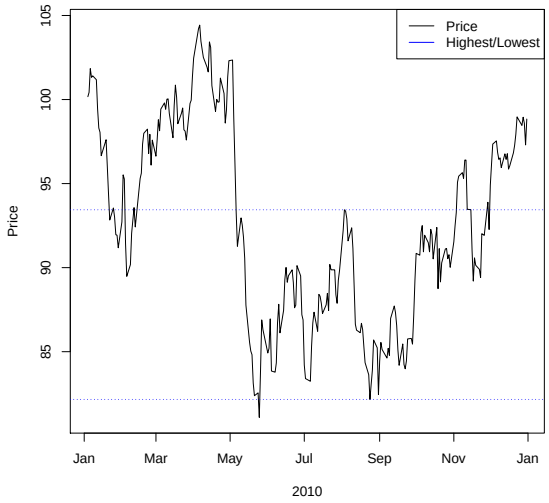

Fig. 2. Channel Breakout example indicating highest and lowest prices in a 3 month channel July - September 2010.

C. Volatility Ratio

The Prototype described in this paper implements an additional filter based upon the market volatility over a short and long period of past trading days. The volatility formula is given by :

$$v = \frac{\sigma(short\ term\ window)}{\sigma(long\ term\ window)}$$

where σ is the standard deviation of the closing price over short and long term windows. A third parameter sets a threshold for the volatility ratio and will signal a trade only if the volatility falls above the parameter value.

D. Optimization Problem Size

The major difficulty with this form of optimisation is that in order to calculate the optimal parameters, all permutations are tested against a set of sample data.

Table I shows a trading model where two parameters are used for the Moving Average Crossover, thus creating 2715 combinations. Using the Volatility Ratio introduces three new parameters; the short and long term periods for calculating the volatility ratio and the volatility threshold itself. Using typical range bands, the model now has over 6 million parameter combinations and would take an inordinantly long time to optimise using a sequential approach over a lengthy underlying dataset. It is clear that optimisations with several parameters

Indicator	Parameter	Value Range	Total candidate values
Moving Average Crossover	Short Term	5-19 days	15
	Long Term	20 - 200 days	181
			= 2715 combinations
Volatility Threshold	Short Term Vol	5-19 days	15
	Long Term Vol	20-50 days	31
	Threshold	0.5 - 0.9	5
			= 6m combinations

TABLE I
TRADING MODEL PARAMETERS

rapidly become impractical using the exhaustive sequential approach.

E. Key CUDA Considerations

[1] describe how CUDA's SIMT architecture enables the efficient management of hundreds of threads running different programs. Similar to SIMD in that it applies the same instruction to multiple data-parallel threads, the fact that the threads are physically grouped into independent warps allows for finer levels of thread granularity in CUDA.

Previous SIMD parallel architectures enabled data-parallel processing in which similar threads could process many data items simultaneously. However, the time-series trading data of Technical Analysis cannot be processed independently as P&L is calculated cumulatively throughout each simulation; each day's P&L is dependent on that of the previous day. In contrast, the CUDA architecture allows a more granular approach which could aid the design of an accelerated back-testing procedure. [2] refer to CUDA's implementation of the Single-Program Multiple-Data (SPMD) model in which the multiprocessors execute different instructions of the same program on multiple parts of the data. This opens the possibility to spread the optimisation parameter combinations over many cores.

III. PROTOTYPE DESIGN AND IMPLEMENTATION

The aim of the Prototype optimiser is to investigate the feasibility of using CUDA technology to derive global optimal parameters for a homogeneous trading model. In many situations an analyst will know the technical indicators they wish to employ (e.g., Moving Average Crossover and Channel Breakout) and the underlying stock instrument they wish to trade. The difficulty arises in finding the the optimal parameters to apply to that model. Thus, the optimiser must find the most attractive parameters by testing thousands of combinations using the same underlying indicators and finding those that yield the greatest cumulative profit. This Prototype seeks to parallelise the process.

Although the correct cumulative P&L is calculated as part of the optimisation process, the aim is to develop an efficient approach to the parameter optimisation of trading models rather than test the veracity of the models themselves.

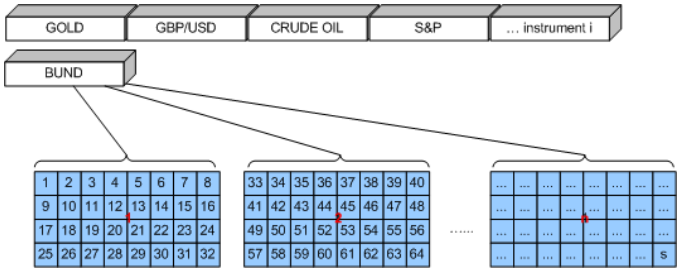

Fig. 3. The thread-centric approach employed by this Prototype. Each thread evaluates a different solution, regardless of their block. One underlying instrument is evaluated by the kernel at a time, with multiple kernel launches to cover all instruments.

A. CUDA Kernel Distribution

In this Prototype an enhancement of the ThreadGP implementation presented by [3] is employed; each thread executes a different solution defined by its parameters, irrespective of thread block. Illustrated in figure 3, the kernel evaluates just one instrument at a time with the candidate solutions distributed amongst all available threads. Multiple kernel invocations are performed for each instrument of the portfolio. Whilst BlockGP is found in the cited works to be a superior model for genetic programming solutions, it is shown that the modified ThreadGP approach has merit when performing pure parameter optimisations for known trading models. Additionally, it will be explained that with careful ordering of thread specific parameters within the blocks this approach can leverage the Single Program-Multiple Data (SPMD) capabilities of later NVIDIA architectures. Divergence is cited in the work reviewed as a key concern in ThreadGP. However, the kernel code can be implemented and distributed in such a way as to minimise the instances of branched execution.

B. Per-thread Solution Encoding

A key challenge in providing a thread-centric solution is for each thread to identify the set of parameters on which it is assigned to work. GPU solutions for Evolutionary Computing exhibit a number of ways in which solutions are represented in memory in various implementations, with bit-strings and LISP style S-expressions commonly found. These solutions are stored in GPU global memory and referenced via the unique thread indices. The solutions must be transferred from global memory to each thread for execution, thereby restricting

processing time due to global memory transfers and bandwidth limitations.

This Prototype proposes an algorithm which deduces the set of thread specific model parameters using the unique thread indices provided by CUDA. The `getTradingParameters` function maps each unique thread ID to a unique set of parameters related to the model's technical indicators. This lightweight solution requires that each thread need only be aware of the technical indicators to be optimised (e.g., Moving Average Crossover and Volatility Ratio), their associated parameter ranges, and the unique thread identity number; from these the algorithm deduces the unique set of parameters to be executed by each thread. Algorithm III.1 sets out the method.

Algorithm III.1: Parameter deduction algorithm (Moving Average only)

grouping $\longleftarrow 1$;
solutionID $\longleftarrow threadIdx.x + blockIdx.x * blockDim.x$;
if *moving_average_crossover* **then**
    $ma\_short\_term \longleftarrow$ `GetParam(solutionID,shortTermStart,shortTermLength,grouping)` ;
    $ma\_long\_term \longleftarrow$ `GetParam(solutionID,longTermStart,longTermLength,grouping)` ;
end
GetParam:
    $param \longleftarrow \frac{\text{solutionID}}{\text{grouping}} \bmod length + start$;
    $grouping \longleftarrow \text{grouping} * length$;
    return $(param)$;
end

This method is found to be an effective way to encode candidate solutions for a number of reasons including :

- It utilizes the inbuilt CUDA indexing functionality and does not consume additional register resources
- It does not require a matrix of candidate solutions to be maintained in device global memory. As an example, the trivial 2715 shown in table I would consume 42K of Global memory. Large optimisations numbering millions of candidate solutions would quickly consume global memory resources.
- A single integer can represent an entire set of parameters. This is useful for efficiently returning the top performing solutions to the host and maintaining collections of top performers.

C. Technical Indicator Algorithms

"Sliding Window" algorithms for calculating technical indicators are developed for this prototype in order to mitigate the thread divergence that would be caused by standard two-pass algorithms on CUDA. The model parameters refer, in most cases, to the number of days with which to calculate a value (e.g., average closing price). Each kernel iterates over the set of underlying data, maintaining various sliding windows comprising of days, the sizes of which are determined by the parameters.

To achieve this, a modification to a version of Welford's algorithm has been devised for this Prototype. The original algorithm, as described by [4], normally calculates the Mean, Variance and Standard Deviation in one pass for a stream of incoming data. In this Prototype, is required to calculate these primitives in a sliding window; data at the tail-end of the window must be discarded as new data arrives. Thus, the Mean and Variance must be adjusted accordingly.

Algorithm III.2 outlines the full modification to Welford's equation which is used in the Prototype to calculate the mean and standard deviations for volatility and Moving Average crossovers.

Algorithm III.2: One Pass approach. Modified version of Welford's algorithm to calculate mean M, variance S and standard deviation s for day k in a window of w days.

if $k - w >= 0$ **then**
    $M_k = \frac{1}{k-1}((M_k * w) - x_{k-w})$
    $S_k = S_{k-1} - (x_{k-w} - M_{k-1})(x_{k-w} - M_k)$
end
$M_k = M_{k-1} + \frac{1}{k}(x_k - M_{k-1})$
$S_k = S_{k-1} + (x_k - M_{k-1})(x_k - M_k)$
$s \longleftarrow \sqrt{\frac{S}{w-1}}$

D. Data Structures

The following data-structure defines the underlying sample data for the instruments:

```
typedef float4 EODDATA;  //.w=open .x=high .y=low .z=close

//Daily data element
typedef struct _EOD {
    unsigned int tradedate;
    EODDATA OpenHighLowClose;
} EOD;
```

An array of `EOD` structures are used to store each instrument's data in the host PC's memory. However, for the device, it is not strictly necessary to store the trade date; it suffices to store a chronological 128-bit `EODDATA` vector array, as shown in figure 4. CUDA attempts to read from global memory in a coalesced fashion in blocks 32, 64 or 128 bytes. Employing the `EODDATA` structure rather than the date-inclusive EOD structure ensures a better ratio of L2 cache-hits when threads are reading this data from device global memory.

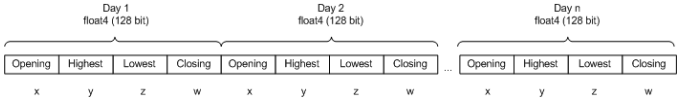

Fig. 4. `EODDATA` structure. Underlying instrument data is packed into `float4` vectors for storage in device global memory.

Table II outlines the key data structures used in the Prototype and the different types of device memory in which they are stored.

IV. Results and Conclusions

Testing of the Prototype was performed on two NVIDIA GPUs alongside sequential and parallel versions of the optimiser on the CPU, as outlined in table III. In the CUDA

Structure	Device Memory
Daily Instrument Data	Global DRAM
Parameter Range	Constant
Per-thread parameters	Shared

TABLE II
DATA STRUCTURES AND CORRESPONDING DEVICE MEMORY AREAS.

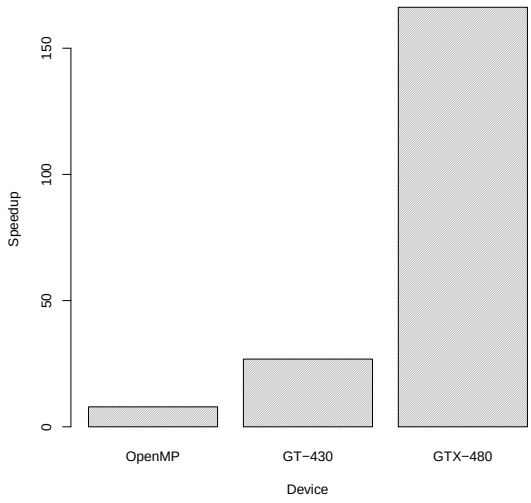

Fig. 5. Speedups observed for a Moving Average Crossover combined with a Channel Breakout model (84,165 solutions).

implementation, the optimisation algorithm is implemented as both a host and device module, thereby ensuring that the same instructions are executed on the CPU as on the GPU. The CPU serial version can thus be considered a suitable baseline with which to compare parallel performance. A parallel version of the Prototype optimiser has also been developed using **OpenMP** to allow comparisons between CUDA and native CPU parallel processing. The Prototype testing is performed on one underlying instrument (BUND). This is sufficient to measure the performance analytics.

The optimal GPU implementation algorithms are found to be the modified Welford's algorithm for the Moving Average Crossover and Volatility Ratio indicators, and a two-pass method for the Channel Breakout indicator. A single-pass double-ended queue was also tested in order to evaluate the minima and maxima for the Channel Breakout in a sliding window, as described by [5]; this was found not to be successful on the GPU as new nodes are scattererd throughout global memory leading to high degrees of latency.

The speedups and performance metrics observed for a trading model comprising a Moving Average Crossover and Channel Breakout are outlined in figure 5 and table IV respectively. The parameter ranges used for the model yield a solution space of 84,165 possible parameter combinations.

The speedup shown in table IV is calculated as the ratio of serial and parallel execution times; $\frac{T_S}{T_P}$, where T_S is the serial runtime and T_P the parallel runtime. As expected, the results observed for the OpenMP version and CUDA devices are bounded by the number of processors available. Although encouraging speedups are obtained by OpenMP in this instance, the CUDA devices considerably outperform it in terms of speed.

Cost is calculated using pT_P and Efficiency (E) is calculated using $\frac{T_S}{pT_P}$. The OpenMP version is practically cost-optimal, as its efficiency is close to $O(1)$. The 8 CPU cores exhibit an almost linear increase in speedup in this case [1]. The calculated efficiencies of the GPUs are considerably less than that of OpenMP. This may be attributed to the different architecture of the streaming multiprocessors on the GPU compared to the CPU cores; it should be borne in mind that with smaller cache sizes and higher latency requests to DRAM on the GPU, it is not comparing like-with-like when calculating the efficiency function.

Total overhead (T_O) is calculated by $pT_P - T_S$. OpenMP exhibits very little overhead at 8 CPU processors. Normally, the overhead should increase as the number of processors p increases. However, it is observed here that the smaller GT-430 exhibits a greater cost and overhead. The main reason for this is identified as Memory Bandwidth; the GTX-480 has several hardware improvements over the GT-430, most notably its 177.4 GB/sec global memory bandwidth compared with the 28.8 GB/sec found on the GT-430 (see table III). These feature is likely to contribute to the lower cost and overhead enjoyed by the GTX-480 when executing this Prototype.

Table V shows the performance metrics observed during backtesting the optimal implementation of a Moving Average Crossover combined with a Volatility Ratio indicator. The Volatility Ratio has 3 parameters, thereby increasing the solution space exponentially. The the test range yields a solution space of almost 12.3 million permutations. The speedups are illustrated in figure 6.

Broadly similar results as for the Moving Average Crossover and Channel Breakout combination are achieved; the speedup attained by OpenMP is the most efficient per parallel core, at 89%. Despite lower efficiency per CUDA core, the GPUs significantly outperform the OpenMP implementation. Wallclock speeds encountered range from 2.5 hours on a single CPU, 22 minutes using OpenMP to 5.5 minutes on the GT-430 GPU and just 48 seconds on the GTX-480 GPU. Clearly, achieving evaluation speeds such as these with a five-paramater model are of great benefit to the analyst; complicated backtesting simulations can be modified and re-submitted in far less time than previously experienced.

The key factors contributing to the optimal performance are identified as:

- **Divergence Management** due to use of the modified Welford's algorithm
- **Memory Management**; with the underlying instrument data stored as `float4` vectors, and the L1 cache disabled for this Prototype, [6] set out how global memory reads

[1] Adding additional CPU cores would eventually result in non-linear speedup according to Amdahl's Law

Device	Cores	Memory	Clock Speed	Memory Bandwidth
CPU: Intel Xeon E5420 Serial	1	4096 MB	2.5 GHz	102 GB/sec
CPU: Intel Xeon E5420 OpenMP	8	4096 MB	2.5 GHz	102 GB/sec
NVIDIA GT-430 GPU	96	1024 MB	1400 MHz	28.8 GB/sec
NVIDIA GTX-480 GPU	480	1536 MB	1401 MHz	177.4 GB/sec

TABLE III
TEST ENVIRONMENT DEVICES.

Device	Cores	Time(ms)	Speedup	Cost(ms)	Efficiency	Overhead
CPU	1	33950		33950		
OpenMP	8	4296	7.9	34368	0.988	418
GT-430	96	1266	26.82	121536	0.279	87586
GTX-480	480	205	165.61	98400	0.35	64450

TABLE IV
PERFORMANCE ANALYTICS FOR THE OPTIMAL IMPLEMENTATION OF MOVING AVERAGE CROSSOVER AND CHANNEL BREAKOUT, WITH 84,165 SOLUTIONS BACKTESTED AGAINST TWENTY YEARS OF ONE UNDERLYING INSTRUMENT (BUND)

Device	Cores	Time(ms)	Speedup	Cost(ms)	Efficiency	Overhead
CPU	1	9552022		9552022		
OpenMP	8	1335480	7.2	10683840	0.89	1131818
GT-430	96	337944	28.3	32442624	0.30	22890602
GTX-480	480	48081	198.7	23078880	0.41	13526858

TABLE V
PERFORMANCE ANALYTICS FOR THE OPTIMAL IMPLEMENTATION OF MOVING AVERAGE CROSSOVER AND VOLATILITY RATIO, WITH 12.3 MILLION SOLUTIONS BACKTESTED AGAINST TWENTY YEARS OF ONE UNDERLYING INSTRUMENT (BUND)

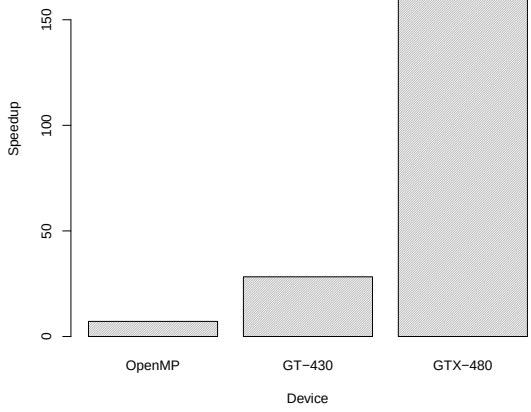

Fig. 6. Speedups observed for a Moving Average Crossover combined with a Volatility Ratio model (12.3 million solutions).

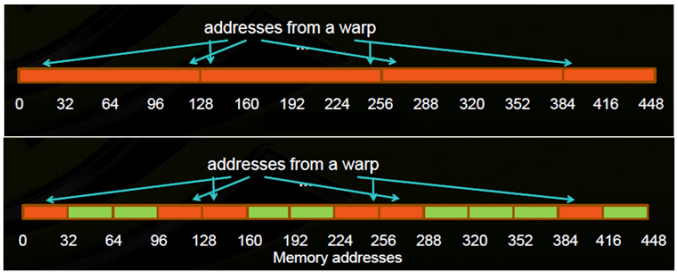

Fig. 7. Fermi offers two modes of scattered global memory access. The top example shows L1 and L2 cache with 128 byte granularity whilst the bottom example shows L2 cache only with 32 byte granularity. The bottom figure represents a more efficient bus utilisation for this implementation.

are cached with a 32-byte granularity, yielding better bus utilisation (see figure 7). With the instrument data packed into `float4` vectors in global memory as outlined in section III, the possibility of any one day's data spanning over two cache-lines is reduced. The 128-bit vector size is a factor of the 32-byte granularity, thus ensuring that the instrument data reads will be captured efficiently in cache loads.

The order of **parameter distribution** across the blocks also plays an important role in this optimal implementation. The Channel Breakout indicator is deployed here using a two-pass approach. A one pass approach using linked lists was attempted; however, this was found to perform poorly on the GPU as `malloc`ed nodes are scattered throughout global memory and the framework does not allow dynamic allocation of Shared memory. As two passes must be made over the data in order to record the maximum and minimum prices within a thread-specific range, the possibility of divergence is raised. However, because the Channel Breakout range parameter is grouped last in the parameter deduction algorithm, it is ensured that intra-block changes to this parameter occur at a higher level than changes to the Moving Average Crossover parameters; i.e., all threads in a block are likely to be processing the same value for the Channel Breakout range. When the

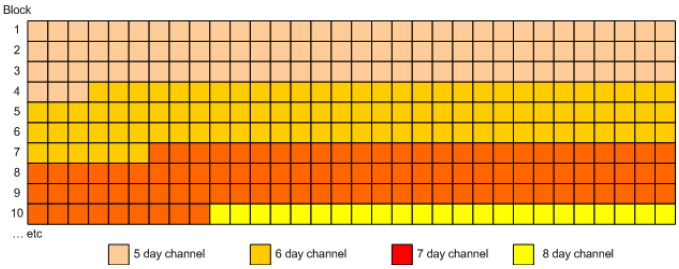

Fig. 8. Distribution of Channel Breakout parameters occurs at a lower level than other parameters. Within each coloured block all other parameters are processed. Divergent blocks (4,7,10) occur relatively infrequently.

range changes, it can never occur more than once in a block, thereby keeping this divergence to a minimum. Furthermore, the divergence occurring in this parameter is only by a factor ensuring that two paths will exist within a warp rather than one. Figure 8 illustrates how the changing values of the Channel Breakout parameter occur less frequently with respect to parameters grouped in higher levels. The range of Moving Average Crossover parameters are processed within the coloured blocks.

A. Conclusions

This work explores the feasibility of optimising the parameters for homogeneous technical analysis models using parallel architectures, in particular NVIDIA's CUDA architecture and development suite. Three popular technical indicators are selected; a Moving Average Crossover, Channel Breakout and Volatility Ratio. The purpose of the research is to investigate optimal ways to parallelise the backtesting simulations for a user defined range of parameters. The threads yielding the highest cumulative P&L for each instrument and their corresponding parameters are useful to traders as these parameters are deemed to perform best over the patterned cycles of price fluctuations encountered by the underlying stocks.

Several areas of technical interest are identified which need to be taken into consideration when developing a Prototype optimiser; Representation of individual solutions in memory, distribution of solutions across the parallel architecture, and efficient algorithm design provide some of the most important elements to be considered. Efficient sliding window algorithms to calculate primitives were researched in order to devise a solution that is non-divergent. Parallel distributions of optimisations and genetic algorithms were investigated and provided ideas and inspiration for the Prototype developed here.

The Prototype itself is designed around a Thread-centric kernel distribution similar to work researched in the literature but previously shelved in favour of a Block-centric approach. Because the trading model solution space is more homogeneous than that found in a Genetic Programming implementation, it is possible to parallelise the solutions on a per-thread basis. The most important factors contributing to this implementation are:

- Avoidance of thread divergence within blocks (or warps) using a modification to Welford's one-pass algorithm
- An algorithm which deduces a thread's unique set of parameters using its own intrinsic thread-ID, and a data structure containing the general set of parameter ranges specified by the user. This component is crucial as it removes the requirement to create an array of structures in global memory which maps each solution to a thread.
- The ordering of parameters is investigated. It is found that in models where a sliding window technique is not feasible (such as Channel Breakout), the parameters can be ordered in such a way so as to minimise the divergence encountered to non-noticeable levels.

This optimal implementation achieves speedups ranging from 8 for an OpenMP implementation to 200 for a top of the range NVIDIA CUDA graphics adaptor. Alternative implementations (such as a customised priority queue for Minima and Maxima, and a two-pass approach for Mean and Standard Deviation) were tested and found to compare unfavourably to a significant degree with the optimal implementation.

This research concludes that parallelising technical analysis simulations in order to find global optimal parameters is a viable exercise. Cost effective cutting edge technologies, particularly CUDA, make it possible to contemplate the optimisation of parameters covering millions of solutions executing in seconds rather than the many hours previously seen in commercial packages.

Future challenges in this area could be to expose the optimiser as a service whereby traders could upload sample data and parameter ranges to a remote server which returns the optimised solutions. There is also considerable scope for further research in devising heterogeneous solution sets, with the possibility of different technical indicators being evaluated in parallel, rather than just the parameters of a homogeneous model. This could possibly be achieved by distributing the differing solutions across different blocks. The key task will be to derive the model from the thread-ID in a similar way to that implemented in this Prototype.

ACKNOWLEDGMENT

With thanks to Mr. James Murray, Structured Products, Allied Irish Banks Wholesale Treasury, Dublin.

REFERENCES

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1373105.1373197

[2] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[3] D. Robilliard, V. Marion-Poty, and C. Fonlupt, "Genetic programming on graphics processing units," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 447–471, December 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1666141.1666145

[4] D. E. Knuth, *The Art of Computer Programming, Volume 2*, 3rd ed. Addison-Wesley, 1998.

[5] Y. Jiao, "Maintaining stream statistics over multiscale sliding windows," *ACM Trans. Database Syst.*, vol. 31, pp. 1305–1334, December 2006. [Online]. Available: http://doi.acm.org/10.1145/1189769.1189773

[6] NVIDIA, *NVIDIA CUDA C Programming Guide*. NVIDIA, 2011.