GPU Acceleration of a Basket Option Pricing Engine

Sean Trainor and Danny Crookes

***Abstract*— One of the most important methods for pricing complex derivatives is Monte Carlo simulation. However, this method requires a large amount of computing resources for accurate estimates. Since Monte Carlo simulations used in derivatives pricing are often parallelisable, one way to reduce the computing time is to use GPUs, which allow many copies of the same process to be run in parallel with different data.**

This paper first presents a GPU implementation of a Basket Option pricing engine and an analysis of the timing and memory resources for different algorithm parameters. The results show that, on an NVidia GTX670 card using NVidia's proprietary CUDA programming platform, a speedup of over 250 can sometimes be achieved compared with a sequential C implementation.

To produce more portable code which will run on a range of different parallel architectures, OpenCL is becoming popular. However, users can be reluctant to adopt OpenCL in case they lose performance compared with an architecture-specific programming platform such as NVidia's CUDA, particularly when high speed-ups are at stake. This paper secondly reports experiments which show that, all things being equal, the performance of OpenCL and CUDA implementations are within approximately 10% of each other, with the OpenCL implementation sometimes being the faster. This suggests that OpenCL is a viable programming platform for GPU acceleration of pricing engines, even when aiming for high speed-up factors of 100-300.

***Index Terms*— Acceleration, GPU, OpenCL, Options Pricing**

I. Introduction

One of the most important methods for pricing complex derivatives, and sometimes the only practical one, is Monte Carlo simulation. This method is commonly used, but it requires a large number of simulations to achieve an accurate price estimate. This in turn requires a large amount of computing resources. Since Monte Carlo simulations used in derivatives pricing are often parallelisable, one way to reduce the computing time is to use multi-core CPUs or many-core GPUs, which allow many copies of the same process to be run in parallel with different data. The technology currently employed by many financial institutions is grid computing, and this often involves sending data out to a central computing grid, which can cause security issues. These grids also consume a lot of energy, in comparison to that consumed by GPUs, for the same amount of computation.

In order to aid developers, manufacturers of GPUs have developed programming languages for their products which allow them to be used for more general purpose computing. GPUs consist of many hardware managed single instruction multiple data (SIMD) units, each of which can have multiple threads running at once. These multithreaded processors were originally designed to process large numbers of pixels very efficiently for graphics processing applications, and so will work well when data can be broken up into smaller pieces and these pieces processed independently.

This paper focuses on accelerating a Basket Option derivative pricing engine using an NVidia GPU (the GTX670). For programming NVidia GPUs, one proprietary programming language is NVidia's CUDA (released in 2006). Many authors (see e.g. [1], [2], [3]) have shown that using the CUDA programming platform on NVidia GPUs is highly effective for accelerating option pricing and risk calculation. This paper firstly describes the implementation of a Basket Option derivative pricing engine on an NVidia GTX670 GPU, and reports a speed-up of over 200 compared to a single-core CPU implementation. However, using CUDA restricts developers to writing code that only operates on NVidia GPUs, and with the constant changing and updating of the set of parallel architectures available today, algorithm developers would like their code to be able run on a range of different current and future architectures. One potentially portable cross-platform programming language, which may allow them to achieve this goal, is OpenCL. This paper also investigates whether the very high speed-up of the CUDA implementation can be retained when the pricing engine is coded in OpenCL, even though it is not architecture-specific.

For a financial company considering investing in GPU implementation, the following questions arise:

- What speedups can a GPU offer?
- Which programming platform gives greater speedups?
- If code portability is required (through the use of OpenCL), what are the performance costs, if any?

This paper presents a GPU implementation of a basket option pricing engine and a comparison of OpenCL and CUDA for implementing it. We compare a combination of three kernel functions which make up this pricer: the first generates correlated samples from normally distributed random numbers, the second generates the price paths and the third performs a binary reduction.

Manuscript received March 19, 2013; revised April 11, 2013

Sean Trainor is with The Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, Belfast BT3 9DT, UK (e-mail: strainor05@qub.ac.uk).

Danny Crookes is with The Institute of Electronics, Communications and Information Technology (ECIT), Queen's University Belfast, Belfast BT3 9DT, UK (e-mail: d.crookes@qub.ac.uk).

II. RELATED WORK

In the area of derivatives pricing and risk calculation, there is always a demand for increased computational capacity. An important, and often indispensable, tool for these types of calculations is Monte Carlo simulation. However, a large number of price path simulations need to be carried out in order to obtain an accurate enough answer.

Situations in which Monte Carlo methods are most useful usually involve at least one of the following (see [4]): complex asset price dynamics, path dependence of the derivatives, or hedging strategies depending on more than two or three underlying assets. Basket options fall into at least the last of these categories, and so it is appropriate to use Monte Carlo methods to price them.

The use of GPUs for accelerating Monte Carlo simulations has been investigated. Bernemann et al [1] give an example of an exotic option pricing system that uses GPUs, although only path generation is implemented on the GPUs (and a speedup factor of around 100x for 1 GPU over 1 CPU is achieved), and it is stated that, for products that require payoffs to be calculated frequently, it may be better also to implement the payoffs on the GPU. Their paper describes the pricing of basket options (options on combinations of equity underlyings) with up to 30 underlyings, and it is evident that, for a large number of underlyings, the GPU does not have enough memory.

In a study by Dixon et al [2], the importance of speeding up risk calculations is stressed. Their approach is to use a quadratic approximation to the loss function of a portfolio, and to use low-discrepancy sequences to increase the convergence rate of the Monte Carlo simulation. Very large speedups are achieved (up to 148x), which shows how suitable GPUs are in this situation.

In [3], Solomon et al. implement a trinomial lattice pricer for European options and a binomial lattice pricer for American lookback options. They find that, for large numbers of time-steps (between 1000 and 30,000), the GPU gives speedups which increase, almost linearly, up to 101x (for 30,000 steps). This shows that GPUs can also be useful when using other types of options pricing methods. It is also found that “above 30,000 steps, the accuracy is no longer able to maintain reliable precision with the CPU version”.

More recently, a new open programming language, OpenCL (Open Computing Language), has been developed (first released in 2009). It allows programmers to write code that can be executed on many different compute devices with very minor modification. This includes devices such as: multicore CPUs, Cell Broadband Engines, FPGAs [5] and GPUs, including NVidia GPUs. When using OpenCL, programmers write code for an abstract memory model rather than for that of a specific architecture, and this allows for portability.

In a comparison between OpenCL and CUDA for Monte Carlo simulations of a Quantum system, Karimi et al. [6] find that CUDA performed better when transferring data to and from the GPU and they did not see any considerable change in OpenCL’s relative data transfer performance as more data were transferred. CUDA’s kernel execution was also consistently faster than OpenCL’s, despite the two implementations running nearly identical code.

A more comprehensive comparison between CUDA (version 3.2) and OpenCL is carried out by Fang et al. [7]. They find that CUDA performs at most 30% better than OpenCL and they also show that this difference is due to “unfair comparisons”. In total, 14 benchmarks are used, including matrix multiplication, Fast Fourier Transform, molecular dynamics and sorting. By changing code and analysing PTX (Parallel Thread Execution) codes (low level NVidia GPU code) it is shown that “the performance gaps between OpenCL and CUDA are due to programming model differences, different optimisations on native kernels, architecture related differences, and compiler differences”. They also found that OpenCL outperformed CUDA in achieved peak bandwidth by 8.5% on GTX280 and 2.4% on GTX480 in a bandwidth test.

Komatsu et al. [8] compare CUDA 3.0 and OpenCL 1.0 for several kernel functions, including matrix multiplication. They find that manual optimisation removes any significant performance difference. The use of OpenCL to make comparisons between NVidia and AMD GPUs is also investigated, and it is found that the optimisation option of the OpenCL C compiler and the work-group size need to be adjusted for each GPU to obtain the best performance.

III. THE OPTION PRICING MODEL

Following Hull [9], the case of a basket option whose payoff depends on n variables S_i $(0 \leq i < n)$ will be investigated. Attention will be restricted to the case of constant volatilities σ_i and constant risk-free interest rates r_i. Letting z_i be an element of a sample (a vector) from a multivariate normal distribution, then discretising the price paths into subintervals of length Δt, we obtain the following price paths:

$$S_i(t+\Delta t) - S_i(t) = r_i S_i(t)\Delta t + \sigma_i S_i(t) z_i \sqrt{\Delta t}\,.$$

It is usually more accurate and more convenient to simulate $ln\,S_i$ rather than S_i. Ito’s Lemma allows us to change variables from S_i to $ln\,S_i$, and so to express the path followed by S_i as:

$$S_i(t+\Delta t) = S_i(t)\exp\left[\left(r_i - \frac{\sigma_i^2}{2}\right)\Delta t + \sigma_i z_i \sqrt{\Delta t}\right],$$

and it is processes of this form that we simulate.

The price of the option will be calculated by taking the average over all simulation paths (trials) for each asset, discounting back to the start time, and multiplying each average by its respective weighting. The average over these results will be taken to give the price estimate. This can be expressed mathematically as:

$$Price = \frac{1}{numAssets}$$

$$\times$$

$$\sum_{i=0}^{numAssets-1} weight_i \times pathPayoff_i \times e^{-rate_i \times holdingTime}$$

with

$$pathPayoff_i = \frac{1}{numPaths} \sum_{j=0}^{numPaths-1} pathPayoff_{ij}(holdingTime)$$

where $pathPayoff_{ij}(holdingTime)$ is the estimate of the value of asset i, on simulation j for that asset, at the end of the holding time of the option.

IV. METHODOLOGY AND EXPERIMENTAL SETUP

In these experiments (for both the basket option pricing and the CUDA/OpenCL comparison) an NVidia GTX670 GPU was used and compared against an Intel Xeon 3.3GHz CPU. This GPU has 1344 cores, with each running at 980MHz. We compare OpenCL version 1.2 (see [10]) and CUDA version 4.0.

At the heart of the Monte Carlo implementation is the generation of random numbers. For our experiments, we used the CUDA-based CURAND library (see [11]) to generate large blocks of random numbers (the generators in this library can be used on both the host and the GPU(s)). The random numbers needed for the path simulations were standard normal deviates, and these are produced by the library functions by applying a Box-Muller transform to uniformly distributed random numbers. The uniform deviates used were produced by a XORWOW generator from the CURAND library. This generator parallelises very well, and this contributes positively to the speedups obtained.

Although it is beneficial, in terms of execution time, to generate as many random numbers as possible at a time, it was found that, for some parameter settings, the GPU card does not have enough memory to hold all the random numbers at once. Therefore, we implement the algorithm using batches of random numbers, and accumulate intermediate totals.

The following pseudo-code shows the steps involved in the pricing of the basket option, here: n_A is the number of underlying assets, n_P is the total number of paths generated, n_B is the number of batches of paths generated, n_S is the number of time steps per path, r_A is the risk-free interest rate, σ_A is the volatility and z_A is the *A*th element from a sample from the required multivariate normal distribution.

A. Pseudo-code for the Basket Option Pricer

```
Initiate input values
For each batch B do
  R= Block of $(n_A * \frac{n_P}{n_B} * n_S)$ random numbers
  Generate correlated samples from the required
  distribution using R values.
  For each asset A do
    $total1 = 0$
    For each path $P$ do
      U = startprice$_A$
      For each step S do
        $U = U\exp\left[\left(r_A - \frac{\sigma_A^2}{2}\right)\Delta t + \sigma_A z_A\sqrt{\Delta t}\right]$
      $total1 += U$
    $total1 = total1 \times \frac{n_B}{n_P}$
    $total2_A += total1$
For each asset A do
    $answer += total2_A \times e^{-rate_A \times holdingTime}$
$answer = \frac{answer}{n_A}$
```

B. Obtaining Correlated Samples from Multivariate Normal Distributions

In order to obtain random samples from a multivariate normal distribution which are correlated by the instantaneous correlations between the S_i variables (when the correlation matrix is symmetric and positive definite), a Cholesky factorisation (based on that in [12]) of the correlation matrix is performed to obtain a lower triangular matrix. By multiplying this factorised matrix by a vector of random univariate standard normal deviates, a vector of correlated standard normal deviates is obtained, and this vector is a sample from the desired multivariate normal distribution. A vector of this type must be obtained to simulate each time-step taken by all assets in the basket on each path.

C. Binary Reduction on the GPU

To parallelise the summation process which occurs in the algorithm, we use a standard binary reduction algorithm on the GPU to calculate an intermediate sum for each thread block. These sums are transferred to the host for final summation to give the final result. The binary reduction algorithm also improves the precision of the summation result, compared with a linear summation.

D. CUDA vs. C Performance Comparison

For comparison purposes, we standardised on a fixed number of steps (n_S = 10), and recorded timings (averaged over three runs) for three different numbers of paths (2^{15}, 2^{20} and 2^{25}) and for three different basket sizes (8, 16 and 32 assets). For the larger numbers of paths and assets, the processing had to be done in batches, because of memory limitations. The memory per batch is of the order of $n_A * \frac{n_P}{n_B} * n_S$ which puts a limitation on the number of paths per batch (and therefore on the total number of threads per batch).

V. RESULTS AND ANALYSIS

Table I: *CUDA timings and speedup over a C implementation*

Assets	*Total Paths*	*Batches*	*Paths/ Batches*	*CUDA time (ms)*	*Speedup vs. C*
8	2^{15}	1	2^{15}	1.3	158
8	2^{20}	1	2^{20}	31	251
8	2^{25}	32	2^{20}	888	284
16	2^{15}	1	2^{15}	3.8	118
16	2^{20}	2	2^{19}	88	184
16	2^{25}	64	2^{19}	2698	194
32	2^{15}	1	2^{15}	11.2	95
32	2^{20}	4	2^{18}	294	128
32	2^{25}	128	2^{18}	9222	130

A. Performance Analysis

For accurate results, it is best to focus on Monte Carlo simulations with at least 2^{20} paths. Thus the results show that large speedups of between 128 and 284 can be achieved on a single GPU. It can be seen that the speedup drops when the number of assets is increased. With larger numbers of assets, this is partly because memory limitations reduce the number of threads (paths), with some loss of utilisation. A more important factor, however, is the data access patterns to the correlation matrix, an n_A x n_A matrix held (in our implementation) in shared memory. Optimisations to reduce this bottleneck are possible.

B. CUDA vs. OpenCL Performance comparison

We implemented an OpenCL version of the basket option pricer and compared its performance with the CUDA implementation. OpenCL and CUDA share many similarities: both models have the idea of a host (usually a CPU), both require memory to be allocated on the device(s), and both require explicit transfers of data to and from the device(s). Although these languages share many similarities, the CURAND library is not available in OpenCL. Because we did not have an OpenCL implementation of the CURAND library, we do not include the random number generation times, and use a simple Gaussian generator on the host to obtain the random numbers. This is sufficient to measure data transfer times from host to device and vice versa, and to measure the pricing speeds.

C. Analysis of CUDA vs. OpenCL

Table II shows that OpenCL is faster at transferring data to the GPU, with the difference decreasing slightly with increasing amounts of data to transfer. The pricing column shows that, as the amount of computation increases, the difference in speed between the two versions decreases. Overall, the difference between each of the two versions is only of the order of 10% at most, and for larger amounts of computation, OpenCL is actually faster than CUDA. It should be noted again, however, that these timings do not include the generation of random numbers. This actually highlights another practical consideration to be taken into account in relying on OpenCL as a portable programming platform: the availability of appropriate library implementations for the specific architecture being used. Although the programming notation may be portable, each hardware platform needs its own implementation of the suite of commonly used libraries.

VI. CONCLUSIONS

This paper has explored several issues associated with the use of a GPU to accelerate a Monte Carlo-based Basket Option pricing engine. The study has led to several conclusions:

(i) Very large speedups can be obtained (over 250) from a single NVidia GTX670 GPU for realistic problem sizes. These are significant speedups, and are for a solution to the whole problem.

Table II: *CUDA vs. OpenCL timings and comparison (If the final column < 1.0, OpenCL is faster than CUDA)*

Assets	*Total Paths*	*Transfer time (ms)*	*Pricing time (ms)*	*Total Time*	*Open CL/ CUDA*
CUDA		3.9	1.1	5.0	
8	2^{15}				1.14
OpenCL		4.0	1.7	5.7	
CUDA		80.2	24.0	104.2	
8	2^{20}				1.01
OpenCL		80.1	25.0	105.0	
CUDA		2615.5	769.9	3385.4	
8	2^{25}				0.97
OpenCL		2559.9	735.4	3295.3	
CUDA		7.4	3.1	10.5	
16	2^{15}				1.06
OpenCL		7.7	3.5	11.2	
CUDA		178.9	78.0	256.9	
16	2^{20}				0.90
OpenCL		159.9	72.0	231.9	
CUDA		5166.4	2475.4	7641.8	
16	2^{25}				0.97
OpenCL		5140.3	2265.1	7405.4	
CUDA		11.6	10.0	21.6	
32	2^{15}				0.92
OpenCL		10.3	9.6	19.9	
CUDA		324.3	278.3	602.6	
32	2^{20}				0.93
OpenCL		316.5	246.8	563.3	
CUDA		10141.1	8365.1	18506.2	
32	2^{25}				0.97
OpenCL		10121.3	7858.3	17979.6	

(ii) The speedup over a standard C implementation reduces (by approximately half, to around 130) when we quadruple the number of assets (from 8 to 32). One reason for this is the data access patterns to the shared correlation matrix, which introduces an overhead of order (number of assets)^2. Further optimisations could reduce this bottleneck, but would not necessarily be portable across different hardware platforms.

(iii) An OpenCL implementation was developed and compared with the CUDA version. This showed that OpenCL for larger problem sizes was actually a little faster than the CUDA implementation, though both were within 10% of the other. It is noted that the CUDA implementation did not explore some very architecture-specific optimisations.

(iv) A practical consideration when considering the use of OpenCL is whether or not key libraries are available in OpenCL for the specific architecture in question (e.g. for random number generation). Although the notation of OpenCL may be portable, the library implementations are normally not, even if they are available.

Ongoing research is investigating the performance of the Basket Option pricing engine on a multi-core CPU using the SSE vector processing instruction set.

REFERENCES

[1] A. Bernemann, R. Schreyer and K. Spanderen, “Accelerating Exotic Option Pricing and Model Calibration Using GPUs” (February 2011). Available at SSRN: http://ssrn.com/abstract=1753596 or http://dx.doi.org/10.2139/ssrn.1753596
[2] M. Dixon, J. Chong, K. Keutzer, “Acceleration of Market Value-at-Risk Estimation,” Proceedings of the 2nd Workshop on High Performance Computational Finance (WHPCF '09), 2009.
[3] S. Solomon, K. R. Thulasiram, P. Thulasiraman, "Option Pricing on the GPU," hpcc-icess, 2010 IEEE 12th International Conference on High Performance Computing and Communications, 2010. pp.289-296.
[4] P. Glasserman, Monte Carlo Methods in Financial Engineering, Springer, 2004.
[5] OpenCL for Altera FPGAs: Accelerating Performance and Design Productivity http://www.altera.com/products/software/opencl/opencl-index.html
[6] K. Karimi, N. G. Dickson and F. Hamze, 2010, “A Performance Comparison of CUDA and OpenCL,” Available at http://arxiv.org/abs/1005.2581.
[7] J. Fang, A. L. Varbanescu and H. Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL,” Proc. 2011 International Conference on Parallel Processing, Delft, Sept. 2011. pp. 216 – 225.
[8] K. Komatsu, K Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. "Evaluating performance and portability of OpenCL programs." In The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010), 2010.
[9] J. C. Hull, Options, Futures and Other Derivatives, 7th Edition, Pearson Education, 2010.
[10] The OpenCL Specification http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[11] CUDA Toolkit 4.0 CURAND Guide.
[12] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, Numerical Recipes in C, Second Edition, Cambridge University Press, 1992.