Accelerating Quadrature Methods for Option Valuation

Anson H.T. Tse, David B. Thomas, Wayne Luk

Department of Computing
Imperial College London
London, England
{htt08,dt10,wl}@doc.ic.ac.uk

Abstract

This paper presents an architecture for FPGA acceleration of quadrature methods used for pricing complex options, such as discrete barrier, Bermudan, and American options. The architecture can be optimized for speed and power consumption by exploiting pipelining and parallelism to produce efficient implementations in reconfigurable logic. An optimised implementation using Graphics Processing Units (GPUs) is also developed, to provide a performance and efficiency comparison with an FPGA accelerator. Our 100MHz FPGA implementation demonstrates a 32.8 times speedup over a software implementation running on a Pentium 4 3.6GHz processor, and is 8.3 times more power efficient than a Tesla C1060 GPU with 240 processors at 1.3GHz.

1. Introduction

The financial derivatives trade sees constant innovation and development, with new types of options introduced each year, offering increasingly sophisticated features and complex settlement terms. Although the basic European option can be priced with a closed-form solution, many other derivatives with knock-out/knock-in features (e.g. Accumulator, Decumulator, and Barrier Options), changing strike prices, or discrete settlement days, cannot be priced easily.

Numerical techniques have been developed to value these complex derivative products accurately and efficiently. These techniques include lattice (binomial and trinomial trees), finite-difference, Monte Carlo and quadrature methods.

Quadrature methods have been applied in different areas including modeling credit risk [1], solving electromagnetic problems [2] and calculating photon distribution [3]. It is a powerful way of pricing path-dependent options where the path is monitored in discrete time. A lookback discrete barrier option priced using quadrature methods is more than 1000 times faster than using the trinomial method, while achieving a more accurate result [4].

Using quadrature methods to price a single option is fast and can typically be performed in milliseconds on desktop computers. However, quadrature methods can become a computational bottleneck when a huge number of options are being revalued in real-time using live data-feeds. This paper shows how to accelerate the quadrature computation using Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). The main contributions of this paper are:

- A parallel hardware architecture for option pricing based on quadrature methods (Section 4).
- Implementation of quadrature pricing on FPGA and GPUs (Section 5).
- Evaluation and comparison of the FPGA and GPU implementations against each other, and against a software implementation (Section 6).

2. Related work

Previous work on hardware acceleration of financial simulation has been focused mainly on Monte Carlo methods. A pipelined datapath architecture and an on-chip instruction processor have been reported for speeding up the Brace, Gatarek and Musiela (BGM) interest rate model for pricing interest rate derivatives [5]. An automated methodology has been developed which produces optimized pipelined designs with thread-level parallelism based on high-level mathematical descriptions of financial simulation [6]. A stream-oriented FPGA-based accelerator with higher performance than GPUs and Cell processors has been proposed for evaluating European options [7]. However, the computational cost of the Monte Carlo approach is high, as in order to improve the accuracy of the result by a factor of n, the sample size has to be multiplied by n^2 times.

A pipelined hardware architecture has been developed for the binomial and trinomial option models [8]. However, lattice and finite-difference methods contain two main types of error: "distribution error" and "non-linearity error". Distribution error occurs because a continuous log-normal distribution is approximated by a discrete distribution. Non-linearity error occurs because the lattice or grid

cannot cater for non-linearity in option price for certain values of the underlying asset. Non-linearity in option pricing is frequent for exotic options: for example in a discrete barrier option, at every barrier there is a non-linearity in the option price.

A numerical approach for option pricing based on quadrature methods has been proposed, which overcomes these errors [4], and demonstrates accurate and fast calculation. The paper illustrates the effectiveness of quadrature methods in pricing path-dependent options, e.g. discrete moving barrier options, multiply compounded options, Bermudan options and American call options with changing strike price. Our approach involves repeated evaluations of a complex function which can be mapped to hardware efficiently in a fully pipelined and parallel architecture.

3. Option pricing and quadrature methods

To understand the option pricing with quadrature methods, we first consider the Black and Scholes partial differential equation [9] for an option with underlying asset following geometric Brownian motion:

$$\frac{\partial V}{\partial t}+\frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}+(r-D_c)S\frac{\partial V}{\partial S}-rV=0 \qquad (1)$$

where $V(S,t)$ denotes the price of the option, S denotes the value of the underlying asset, t denotes time, T denotes the time to maturity, r denotes risk-free interest rate, σ denotes volatility of the underlying asset, E denotes exercise price and D_c denotes continuous dividend yield.

The following standard transformations

$$x=\log(S_t/E)$$

$$y=\log(S_{t+\Delta t}/E)$$

give us the solution of $V(x,t)$ as:

$$V(x,t)=A(x)\int_{-\infty}^{+\infty}B(x,y)V(y,t+\Delta t)dy \qquad (2)$$

where

$$A(x)=\frac{1}{\sqrt{2\sigma^2\pi\Delta t}}e^{(-kx/2)-(\sigma^2k^2\Delta t/8)-r\Delta t} \qquad (3)$$

$$B(x,y)=e^{-((x-y)^2/2\sigma^2\Delta t))+(ky/2)} \qquad (4)$$

$$k=\frac{2(r-D_c)}{\sigma^2}-1 \qquad (5)$$

Eq. (2) contains an integral which cannot be evaluated analytically. Although for European options they can be converted to the probability density function for normal distribution, for more complicated options numerical techniques are required to evaluate the integrals. For the evaluation of other complicated options such as discrete barrier options and American options, the valuation problem can be arranged to exploit consecutive time intervals and apply Eq. (2) iteratively.

There are many different methods of numerical integral evaluation. Two of the most common methods include the trapezoidal rule and Simpson's rule [10]:

Trapezoidal rule: The trapezoidal rule is the simplest quadrature method but is the slowest to converge. It converges at a rate of $(\delta y)^2$. The approximation equation is:

$$\int_{a1}^{a2}f(y)dy\approx\frac{\delta y}{2}\{f(a1)+2f(a1+\delta y)+2f(a1+2\delta y)\cdots +2f(a2-\delta y)+f(a2)\} \qquad (6)$$

Simpson's rule: The Simpson's rule is the most popular method for approximating integrals. It converges at a rate of $(\delta y)^4$ which means a doubling of the number of steps reduces the error by a factor of sixteen. The approximation equation is:

$$\int_{a1}^{a2}f(y)dy\approx\frac{\delta y}{6}\{f(a1)+4f(a1+\frac{1}{2}\delta y)+2f(a1+\delta y)+\cdots+2f(a2-\delta y)+4f(a2-\frac{1}{2}\delta y)+f(a2)\} \qquad (7)$$

4. Parallel Architecture

Our system architecture is not designed for pricing a specific option, so the implementation of all kinds of options must be considered. It has been shown that most of the equity options can be expressed in integral forms and solved by quadrature methods including: European options, discrete barrier options, moving discrete barrier options, Bermudan put options, American call options and look-back options [4, 11]. However, the quadrature evaluation procedures are slightly different for different types of options. Different types of options have different discontinuities, which lead to different integral boundaries. Some options contain option specific parameters: for example, the knock-out prices and number of periods are required for discrete barrier options. Although European options can be priced with a single quadrature step, most of the other options need to be evaluated iteratively from the price in period of m to $m-1$. Therefore using different number of quadrature steps is required.

Table 1 shows some of the pricing equations for different option types as illustration. We can see from the table that the pricing equations are different in terms of the integration range and the operation flow. For discrete barrier

Option type:	Pricing equation:
European options	$V(x,t) \approx A(x) \int_0^{N^+ \delta y} B(x,y)V(y,t+\Delta t)dy$
Discrete barrier call options	$C_m(x,T_{m-1}) \approx A(x) \int_{b_m}^{y_{max_m}} B(x,y)C_{m+1}(y,T_m)dy$
Bermudan put options	$P_m(x,T_{m-1}) \approx A(x) \int_{b_m}^{y_{max_m}} B(x,y)P_m(y \geq b_m,T_m)dy + Ee^{-r\Delta t_m}N(-d_2) - Ee^{x-D_c\Delta t_m}N(-d_1)$
American call options	$C_m(x,T_{m-1}) \approx A(x) \int_{y_{min_m}}^{b_m} B(x,y)C_m(y \leq b_m,T_m)dy + E_M e^{x-D_c\Delta t_m}N(d_1) - E_m e^{-r\Delta t_m}N(d_2)$

Table 1: The pricing equations for some example options.

Option type:	Number of integration	Number of evaluation of B(x,y)	Number of evaluation of A(x)
European options	1	N	1
Discrete barrier call options	Nm	N^2m	Nm
Bermudan put options	Nm	N^2m	Nm
American call options	Nm	N^2m	Nm

Table 2: The computation complexity for some example options. N denotes the number of integration grid points and m denotes the number of time steps.

options, the result of C_{m+1} is required for the evaluation of C_m. The final option value C_0 has to be evaluated iteratively. Table 2 shows the computation complexity for different types of options. The computation complexity depends on the evaluation flow, the number of integration grid points N and the number of time steps m.

4.1. System architecture

An interesting finding in Table 1 and Table 2 is that all option pricing equations require the evaluation of a similar integral on $B(x,y)V(y,t+\Delta t)$. Using quadrature methods requires the evaluation of the function $B(x,y)$ intensively, which is the computation bottleneck. Although function $A(x)$ is also required to be evaluated repeatedly, the computation complexity for $A(x)$ is much lower compared to $B(x,y)$ with an order of N. As a result, our hardware acceleration is focused on the effective evaluation of the integral and the flexibility for a general option pricing framework, as illustrated in Figure 1.

The system architecture of the generic option valuation system using quadrature method is shown in Figure 1. The architecture consists of the following components: (a) a pre-processing block, (b) one or more QUAD evaluation core(s), (c) a post-processing block, and (d) a main control unit. Data input for the system is, $K1, K2, T, So, E, r, D_c, \sigma$, option-type and option-specific-parameters. The option-type and option-specific-parameters provide the flexibility to support the pricing of other options. For example, we could specify the number of periods (m) and the knock-out/knock-in prices (b) for barrier options.

The parameter $K1$ denotes the grid density factor. As the underlying asset follows a lognormal distribution and the change of price exhibits Brownian motion, the value of

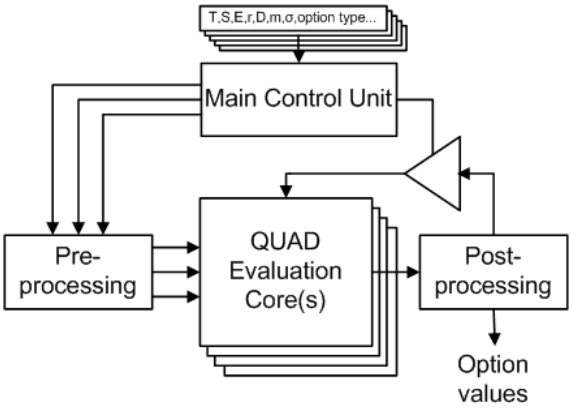

Figure 1: System architecture of the generic option valuation system.

y fluctuates proportional to $\sqrt{\Delta t}$. As a result, $K1$ is defined as:

$$\delta y = \frac{\sqrt{\Delta t}}{K1} \tag{8}$$

Therefore, increasing the value of $K1$ leads to a smaller value of δy and a denser grid.

The parameter $K2$ denotes the grid size factor. It is not possible to integrate a function from $-\infty$ to $+\infty$ numerically. Our quadrature method will evaluate from y_{min} to y_{max} where:

$$y_{max} = x + K2 \cdot \sigma\sqrt{\Delta t} \tag{9}$$

$$y_{min} = x - K2 \cdot \sigma\sqrt{\Delta t} \tag{10}$$

As a result, a large value of $K2$ leads to a large/small value of y_{max}/y_{min} and a wider grid. $K2$ can also be viewed as

the number of standard deviations from y to the original position of x after Δt.

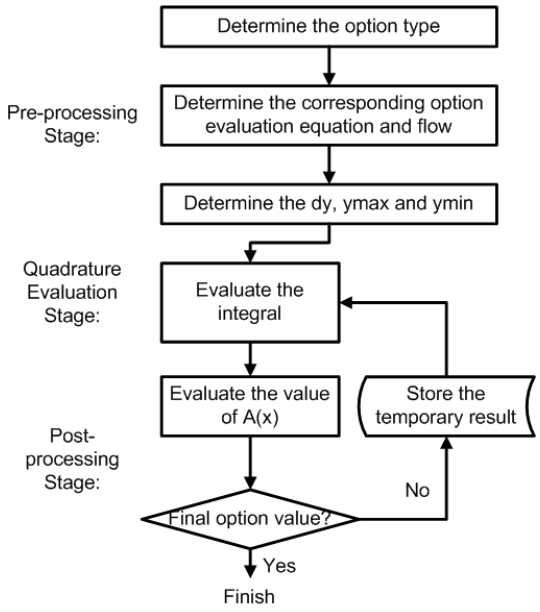

Figure 2: The option evaluation flow.

A typical option evaluation flow is illustrated in Figure 2. The main control unit accepts the basic option input and coordinates with the pre-processing and post-processing block. The pre-processing block determines the option type, selects the corresponding option evaluation equation and computes the non-repeated values such as δy, y_{max} and y_{min}. It then generates the set of y_i, V_i and x for the QUAD(quadrature) evaluation core(s). The QUAD evaluation core(s) evaluate the integral value as Eq. (2) from the pre-processing block and output the result to the post-processing block. The post-processing block combines the integral value with the value of $A(x)$ and produces the value of $V(x,t)$. The main control unit then decides whether $V(x,t)$ is the final solution or a temporary result for the next iteration.

4.2. Architecture Style

The C-Slow approach [12] has been used in many FPGA based simulations and is particularly appropriate in this system in order to achieve high clock rates while still maintaining high throughput. C-Slow operation can be achieved by modelling multiple QUAD evaluation cores in parallel. We continuously provide the pipeline with parameters to evaluate other integral values while we are waiting for the results required for the current integral. The main controller manages the overall timing of the system and ensures that the intermediate values are stored and retrieved correctly.

The QUAD evaluation core is implemented in hardware for 3 main reasons. First, more than one QUAD evaluation core can be fitted on a single FPGA. Therefore, several quadratures can be evaluated simultaneously to exploit parallelism. Second, the evaluation of the function $B(x,y)$ could be implemented in pipelined hardware which is fast and efficient. The value of $B(x,y)$ can be obtained in every clock cycle. Third, as shown previously in Table 2, the evaluation of the quadrature is the computation bottleneck, which would benefit from hardware acceleration.

The main control unit, pre-processing and post-processing blocks are implemented in software for the following reasons. First, this increases the flexibility to support other options. Second, the evaluation in pre-processing and post-processing blocks is not the performance bottleneck. Implementing them on the hardware would not improve the performance significantly.

The proposed software-hardware architecture offers fast and parallel hardware evaluation cores for the repeated numerical integrations, and provides flexibility to a versatile option evaluation platform.

4.3. Optimising QUAD evaluation core

An effective implementation of the QUAD evaluation core is to create a tree of pipelined operators. Figure 3 shows an operator tree based on Eq. 2 to Eq. 7.

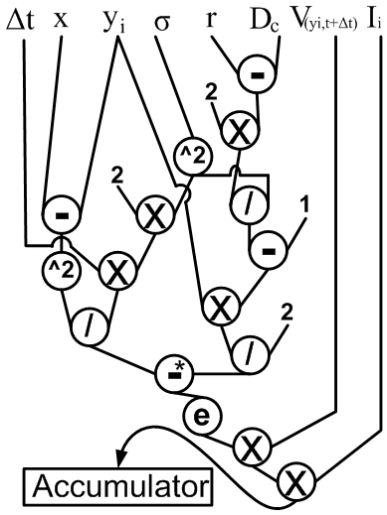

Figure 3: An operator tree diagram for straight forward design (The operator with '*' denotes the operation from right to left).

In Figure 3, $\Delta t, x, y_i, \sigma, r, D_c, V(y_i, t+\Delta t)$ and I_i are fed to the evaluation tree continuously. I_i denotes the integration coefficients in the quadrature method. Using the trapezoidal rule or Simpson's rule, I_i will be in a sequence of

$1,2,2,\ldots,2,1$ or $1,4,2,4,2,\ldots,2,4,1$ respectively as the coefficients in Eq. 6 and Eq. 7. The accumulated value should be multiplied in post-processing with $\frac{\delta y}{2}$ for trapezoidal rule, or with $\frac{\delta y}{6}$ for Simpson's rule. Therefore, the type of quadrature rule can be specified outside the QUAD evaluation cores.

However, the straight forward implementation consumes a large amount of hardware resources as it requires many floating-point operators. The optimized design is shown in Figure 4, and will be used to produce implementations on both FPGA (Section 5.1) and GPUs (Section 5.2).

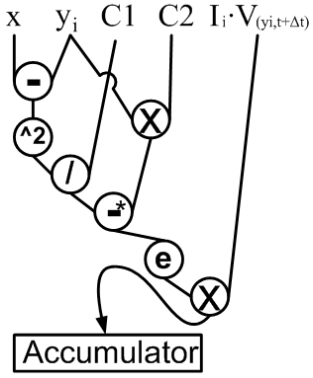

Figure 4: An operator tree diagram for optimized design (The operator with '*' denotes the operation from right to left).

The optimized quadrature operator tree takes the following data input: $x, y_i, C1, C2, I_i$ and $V(y_i, t+\Delta t)$. We define:

$$C1 = 2\sigma^2 \Delta t \tag{11}$$

$$C2 = \frac{\frac{2(r-D_c)}{\sigma^2} - 1}{2} \tag{12}$$

The operator tree is optimized by identifying the non-changing nodes during the pipelined evaluation. The values of $C1$ and $C2$ are fixed for the values of $y_i, I_i, V_(y_i, t+\Delta t), i \in [0, N]$. Therefore, $C1$ and $C2$ can be pre-computed in the pre-processing stage and passed to QUAD evaluation cores. The hardware size is therefore reduced significantly and the number of parameters is also reduced. The parameters of I_i and $V_(y_i, t+\Delta t)$ are passed to the QUAD evaluation cores together. For a grid integration with N steps, the total number of parameters required is of the order $2N$, which is a 33% reduction from the original design which is of the order $3N$. Table 3 summarizes the differences between the original design and the optimized design.

	Original	Optimized
Number of $exp(x)$ operators	1	1
Number of $\times$ operators	8	3
Number of $\div$ operators	3	1
Number of $-$ operators	4	2
Number of input parameters	$3N + 5$	$2N + 3$

Table 3: Comparison between the original design and the optimized design. N denotes the number of integration grid points.

5. Implementation

5.1. QUAD evaluation core on FPGA

Our FPGA implementation of the QUAD evaluation cores is based on *HyperStreams* and the Handel-C programming language. *HyperStreams* is a high-level abstraction language and library [7]. It can produce a fully-pipelined hardware implementation with automatic optimization of operator latency at compile time. This feature is useful when implementing a complex algorithm core.

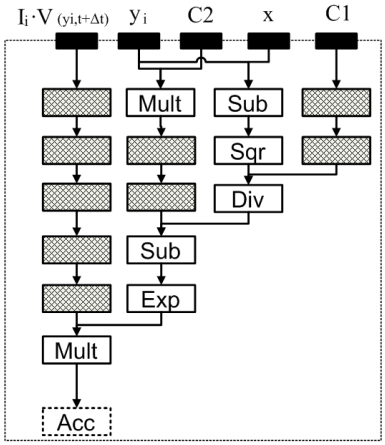

Figure 5: A fully pipelined QUAD evaluation core on FPGA.

Figure 5 shows the diagram of a fully-pipelined QUAD evaluation core based on the design in Figure 4. The grey boxes denote the pipeline balancing registers that are allocated automatically by *HyperStreams*. The QUAD evaluation core therefore produces the value of $B(x,y)$ in Eq. (2) for every clock cycle. From Figure 5, we can see that it requires initially 7 clock cycles for the partial integral value to reach the accumulator. As the evaluation is pipelined, we can have a partial integral value for every clock cycle after. For an FPGA running at 100MHz, the QUAD evaluation

core can produce 100M partial integral values per second.

Bandwidth is one of the major concerns in a hardware accelerated system. For an FPGA running at 100MHz, 100M partial integral values can be obtained per second which require a bandwidth of ≈800MBytes/second for input parameters. The Celoxica RCHTX-XV4 supports a high speed HTX interface for data transfer up to 3.2GBytes/second which is sufficient for the I/O requirement [13]. We also assume that the requests for quadrature evaluation are received concurrently. This assumption allows us to use high-latency pipelined functional units to achieve high clock rate while still achieving high throughput..

5.2. QUAD evaluation core on GPU

Graphics Processing Units (GPUs) have been used for acceleration in various applications [14] [15]. Our implementation on GPUs is based on Compute Unified Device Architecture (CUDA) API provided for nVidia GPUs [16]. A typical CUDA co-processing flow is shown in Figure 6.

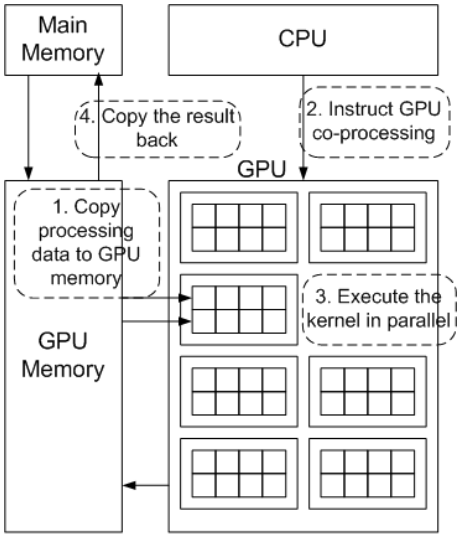

Figure 6: A typical CUDA co-processing flow.

Under the CUDA environment, a function can be compiled into a “kernel”. The execution of CUDA is organized as a computation grid as shown in Figure 7. Each computation grid consists of a grid of thread blocks. The “kernel” is executed by all threads in parallel. Each block and thread has a unique block ID and thread ID.

The QUAD evaluation core is implemented in CUDA to exploit parallelism. Similar to the implementation on FPGA, we implement the evaluation core in CUDA based on the optimized operator tree in Figure 4. In addition, the

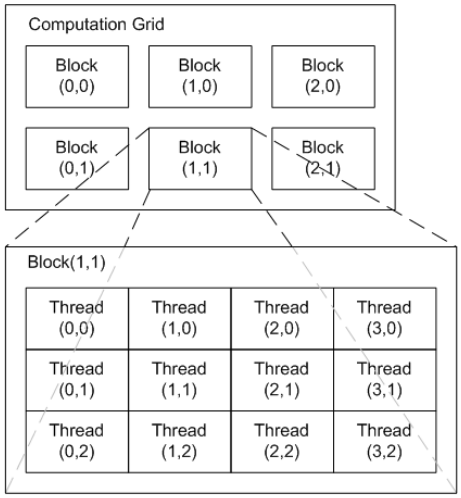

Figure 7: Diagram for the CUDA computation grid.

whole integration is segmented to support different blocks and threads in the CUDA environment. Each thread would evaluate a set of partial integrals and accumulate the result. The first thread in each block then adds up the results from all the threads within the same block. The main thread then adds up all the results from all the blocks. The CUDA pseudo code for the QUAD evaluation kernel is shown in Figure 8.

```
void GPU_EvaluationCore()
{
   unique_thread_ID = Num of block * block_ ID + thread_ID
   THREAD_COUNT  = Num of thread in a block * Num of block in a grid

   for (i = unique_thread_ID ; i <N; i += THREAD_COUNT)
   {
      evaluate partial integral on yi and Vi
      accumulate the value on local register;
   }
   copy local register value to share memory
   Synchronize with all other threads.
   if  (thread_ID==0)                  // the first thread in the block
   {
      Sum up all partial integrals from all threads in the same block.
   }
   Synchronize with all other threads.
   if (unique_thread_ID==0)                  // the main thread
   {
      Sum up all partial integrals from all the first threads in all blocks.
   }
}
```

Figure 8: The CUDA pseudo code for the QUAD evaluation core kernel.

6. Evaluation and comparison

In this section, the performance of different implementations of QUAD evaluation core is studied. We choose

	FPGA		GPU			PC
	Virtex-4 xc4vlx160		Geforce 8600GT	Tesla C1060		Pentium 4
	single	double	single	single	double	double
Slices	20,200 (29%)	51,396 (76%)	-	-	-	-
LUTs	19,713 (14%)	53,792 (39%)	-	-	-	-
FFs	16,605 (12%)	39,281 (29%)	-	-	-	-
DSPs	34 (35%)	96 (100%)	-	-	-	-
Clock Rate	100MHz	81.9MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Processing Speed (M values/sec)	100.0	81.9	114.4	546.5	288.7	9.1
Acceleration (1 core)	10.9x	9x	-	-	-	1x
Replication (core/chip)	3	1	-	-	-	1
Acceleration (replicated cores)	32.8x	9x	12.5x	59.8x	31.6x	1x
Max Power	4.4W	8.8W	43W	200W	200W	115W
Processing/Max Power (M values/J)	22.5	9.3	2.7	2.7	1.4	0.1

Table 4: Performance comparison of different implementation of QUAD evaluation core. The Geforce 8600GT has 32 processors and the Tesla C1060 has 240 processors.

the pricing of European option with grid density factor $K1 = 400,000$ and grid size factor $K2 = 10$. The typical $K1$ value of 400 produces highly accurate results, but the reason for choosing a much larger value is to facilitate performance analysis of the QUAD evaluation cores with a longer evaluation time. No matter what values of $K1$ or $K2$, the QUAD evaluation cores are still responsible for the computation bottleneck of the option pricing of the order N^2m as shown in Table 2. Simpson's rule is preferable to the trapezoidal rule in our system as the error terms of Simpson's rule decrease at a rate of $(\delta y)^4$ which produces more accurate results with the same hardware complexity. Therefore, Simpson's Rule is adopted for the performance analysis. The performance of both 32-bit single precision and 64-bit double precision are investigated.

The accelerations of FPGA implementation and GPUs implementations are in comparison to a reference software implementation. The reference PC is a 3.6GHz Pentium 4 processor with 1GB RAM and Ubuntu 8.04.1 operating system. The software implementation is written using C language and compiled with maximum speed optimization options.

The targeted FPGA is Xilinx Virtex-4 xc4vlx160 as this is the part found in the RCHTX card. The designs are compiled using DK5.1 and Xilinx ISE 9.2. The targeted GPU is nVidia Geforce 8600GT with 256MB of on board RAM and nVidia Tesla C1060 with 4GB of on board RAM. The summary of the performance comparison is shown in Table 4.

FPGA device utilization figures are shown in Table 4. The result indicates that the FPGA device is fully occupied under double-precision but not under single-precision, so a performance improvement can be achieved by replicating the evaluation core under single-precision.

From the results, it can be seen that the FPGA implementation on the xc4vlx160 achieved 10.9 times acceleration using single-precision, and achieved 9 times of acceleration using double-precision, with 1 QUAD evaluation core. As 3 cores can be replicated on xc4vlx160 using single-precision, 32.8 times acceleration can be achieved in the single-precision case.

For the performance of GPUs, a speedup of 12.5 times is achieved by Geforce 8600GT and a speedup of 59.8 times is achieved by Tesla C1060 in single-precision. In double-precision, the Tesla C1060 has shown a 31.6 times speedup over the reference PC, while there is no double-precision support in the Geforce 8600GT.

It can be seen that xc4vlx160 outperforms Geforce 8600GT by 2.6 times in single precision case. It is not surprising that Tesla C1060 GPU outperforms xc4vlx160 by 1.8 times, as Tesla C1060 is the latest GPU based on 65nm fabrication technology while Virtex-4 xc4vlx160 is an older model of FPGA based on 90nm fabrication technology. It would be fair to compare Tesla C1060 with the latest FPGAs such as Virtex-5 from Xilinx or Stratix IV from Altera, which would be addressed in our future work.

It is interesting to note that the xc4vlx160 FPGA demonstrates better performance than the GPUs when power

consumption is taken into account. It can be seen that xc4vlx160 can evaluate 8.3 times more than both Geforce 8600GT and Tesla C1060 per unit of power usage under single-precision. In double-precision, xc4vlx160 evaluates 6.6 times more than Tesla C1060 and 93 times more than Pentium 4 per unit of power usage. An interesting observation is that Tesla C1060 is 4.7 times faster than Geforce 8600GT and consumes 4.7 times more power as well. Therefore Tesla C1060 has the same processing/power ratio as Geforce 8600GT.

Further performance improvement for FPGAs can be achieved using more up-to-date devices such as Virtex-5. As Virtex-5 has 4 times more slices than Virtex-4 and with higher basic clock frequency, at least 4 times more speedup can be achieved without further optimization. Also, although complex algorithms can be implemented easily in FPGAs with *HyperStreams*, maximum performance and utilization of FPGA resources is not guaranteed, as there is a tradeoff when using *HyperStreams* between the development time and the amount of acceleration that can be achieved. However, our *HyperStreams* implementation still provides a satisfactory result with significant acceleration over the software implementations. Therefore, *HyperStreams* is useful for producing prototypes rapidly to explore the design space. Further optimization can be applied after a promising architecture is found.

Fixed-point implementations with sophisticated techniques such as word-length optimization usually enable FPGA to achieve the best performance [17]. However, it is not applicable to quadrature methods as the range of the numerical values ranges widely from small size partial integral values to large size complete integral values.

7. Conclusion

This paper explores a novel architecture for hardware accelerated option pricing models based on quadrature methods. The proposed system involves a parallel architecture for general option pricing. This includes a highly pipelined datapath capable of supporting quadrature evaluation in parallel. We have implemented our designs on FPGA and GPUs. Our FPGA implementation can generally run 32.8 times faster than a Pentium 4 3.6GHz processor, 2.6 times faster than a GPU in comparable technology and 1.8 times slower than the latest GPU. From the power consumption perspective, the FPGA is up to 8.3 times more power efficient than GPUs and 93 times more power efficient than the CPU.

Current and future work includes exploring designs on the latest FPGAs such as Virtex-5 and Stratix IV, for a more sophisticated comparison with the latest GPUs. The automation of producing efficient FPGA and GPU implementations will also be investigated. Additionally, we intend to extend our work to cover the development of optimized hardware designs based on quadrature methods for a wide variety of applications, including the solutions of electromagnetic problems [2] and calculations involving photon distribution [3].

Acknowledgements. The support of the Croucher Foundation, UK EPSRC, AlphaData, Celoxica and Xilinx is gratefully acknowledged.

References

[1] M. H. A. Davis and J. C. Esparragoza-Rodriguez, "Large portfolio credit risk modeling," *International Journal of Theoretical and Applied Finance*, vol. 10, no. 04, pp. 653–678, 2007.

[2] A. Masserey, J. Rappaz, R. Rozsnyo, and M. Swierkosz, "Numerical integration of the three-dimensional green kernel for an electromagnetic problem," *Journal of Computational Physics*, vol. 205, no. 1, pp. 48 – 71, 2005.

[3] T. Humphries, A. Celler, and M. Trammer, "Improved numerical integration for analytical photon distribution calculation in spect," *Nuclear Science Symposium Conference IEEE*, vol. 5, pp. 3548–3554, 2007.

[4] A. D. Andricopoulos, M. Widdicks, P. W. Duck, and D. P. Newton, "Universal option valuation using quadrature methods," *Journal of Financial Economics*, vol. 67, no. 3, pp. 447–471, March 2003.

[5] G. Zhang, P. Leong, C. Ho, K. Tsoi., D.-U. Lee, C. Cheung, R. Cheung, and W. Luk, "Reconfigurable acceleration for Monte-Carlo based financial simulation," in *Proc. Int. Conf. on Field-ProgrammableTechnology*. IEEE, 2005, pp. 215–224.

[6] D. Thomas, J. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *Int. Conf. on Application-Specific Systems, Architectures and Processors*. IEEE, 2007.

[7] G. Morris and M. Aubury, "Design space exploration of the European option benchmark using hyperstreams," in *Proc. Int. Conf. on Field Programmable Logic and Applications*. IEEE, 2007.

[8] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for binomial-tree pricing models," in *Proceedings of the 4th international workshop on Applied Reconfigurable Computing*. LNCS 4943. Springer-Verlag, 2008, pp. 245–255.

[9] F. Black and M. S. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–54, May-June 1973.

[10] E. Sueli and D. F. Mayers, *An Introduction to Numerical Analysis*. Cambridge University Press, 2006.

[11] G. Fusai and M. C. Recchioni, "Analysis of quadrature methods for pricing discrete barrier options," *Journal of Economic Dynamics and Control*, vol. 31, no. 3, pp. 826–860, March 2007.

[12] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement c-slow retiming for the Xilinx Virtex FPGA," in *FPGA '03: Proceedings of international symposium on Field programmable gate arrays*. ACM, 2003, pp. 185–194.

[13] Celoxica, "RCHTX-XV4 datasheet," 2006.

[14] L. Pan, L. Gu, and J. Xu, "Implementation of medical image segmentation in cuda," *Proc. Int. Conf. on Technology and Applications in Biomedicine*, pp. 82–85, May 2008.

[15] H. Jang, A. Park, and K. Jung, "Neural network implementation using cuda and openmp," *Computing: Techniques and Applications, 2008. DICTA '08.Digital Image*, pp. 155–161, Dec. 2008.

[16] nVidia, "Nvidia CUDA programming guide," 2008.

[17] G. A. Constantinides, "Word-length optimization for differentiable nonlinear systems," *ACM Trans. on Design Automation of Elect. Sys.*, vol. 11, no. 1, pp. 26–43, 2006.