Design Exploration of Quadrature Methods in Option Pricing

Anson H. T. Tse, *Student Member, IEEE*, David Thomas, *Member, IEEE*, and Wayne Luk, *Fellow, IEEE*

***Abstract*—This paper presents a novel parallel architecture for accelerating quadrature methods used for pricing complex multi-dimensional options, such as discrete barrier, Bermudan and American options. We explore different designs of the quadrature evaluation core including optimized pipelined hardware designs in reconfigurable logic and a compute unified device architecture (CUDA)-based graphics processing unit (GPU) design. A parametrizable automated system is presented for generating hardware quadrature evaluation cores with an arbitrary number of dimensions. The performance and energy consumption of field-programmable gate arrays (FPGAs), GPUs, and central processing units (CPUs) are compared across different number of dimensions and precisions. Our evaluation shows that the 100 MHz Virtex-4 xc4vlx160 FPGA design is 4.6 times faster and 25.9 times more energy efficient than a multi-threaded optimized software implementation running on a Xeon W3504 dual-core CPU. It is also 2.6 times faster and 25.4 times more energy efficient than a GPU with comparable silicon process technology.**

***Index Terms*—Compute unified device architecture (CUDA), field-programmable gate array (FPGA), graphics processing unit (GPU), multi-dimensional, option pricing, quadrature.**

I. Introduction

FINANCIAL institutions continually invent new ways to repackage and modify financial products in order to satisfy the needs of different investors. While some basic financial options 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, have no known closed-form solution.

Numerical techniques have been developed to value these complex derivative products. These techniques include binomial trees, 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 points. 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 simple 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 complex options are being revalued in real-time using live data-feeds. Many financial derivatives now involve multiple underlying assets instead of just one. As the computation complexity increases exponentially with the number of underlying assets (i.e., the number of dimensions), how to accelerate the quadrature option pricing becomes a significant problem. Energy consumption of computation is also a major concern when the computation is performed 24 hours a day, 7 days a week.

This paper explores the acceleration of quadrature computation using different computational devices including field-programmable gate arrays (FPGAs) and graphics processing units (GPUs). The main contributions of this paper are as follows:

- novel parallel hardware architecture for option pricing based on quadrature methods (see Section IV);
- techniques for multi-dimensional option pricing and a model of the computational complexity (see Section V);
- approach for generating multi-dimensional quadrature evaluation cores for FPGA and GPU (see Section VI);
- comparison of performance and energy consumption of FPGAs, GPUs, and central processing units (CPUs)6p for quadrature evaluation across different number of dimensions (see Section VII).

II. 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, because 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 binomial and trinomial option pricing models [8]. However, lattice

Manuscript received September 10, 2010; revised January 12, 2011; accepted February 18, 2011. This work was supported in part by the Croucher Foundation, by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 248976 and 257906, and by the HiPEAC NoE, by Alpha Data, by Celoxica, by nVidia, and by Xilinx.

A. H. T. Tse is with the Department of Computing, Imperial College London, London, London SW7 2AZ, U.K. (e-mail: htt08@doc.ic.ac.uk; tsehongtak@gmail.com).

D. Thomas is with the Electrical and Electronic Engineering Department, Imperial College London, London SW7 2AZ, U.K. (e-mail: dt10@imperial.ac.uk).

W. Luk is with the Department of Computing, Imperial College London, London SW7 2BZ, U.K. (e-mail: wl@doc.ic.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2011.2128354

and finite-difference methods contain two main types of errors: "distribution errors" and "nonlinearity errors" [4]. Distribution errors occur because a continuous log-normal distribution is approximated by a discrete distribution. Nonlinearity errors occur because the lattice or grid cannot cater for nonlinearity in option price for certain values of the underlying asset. Nonlinearity is common in pricing exotic options: in a discrete barrier option, at every barrier there is a nonlinearity in the option price. Quadrature methods overcome these errors and are shown to be effective in pricing path-dependent options such as discrete moving barrier options, multiply compounded options, Bermudan options, and American call options with changing strike price [4].

In earlier work, we described the reconfigurable architecture of option pricing based on quadrature methods [9]. An FPGA implementation using Virtex-4 demonstrated a speedup of 32.8 times over a software implementation running on a Pentium-4 3.6 GHz processor. In addition, the theoretical complexity of quadrature methods for pricing multi-dimensional options is explored [10]. A 3-D option priced using FPGA is 25 times faster than using a Pentium-4 processor. This paper provides a unified view of these studies; describes the proposed FPGA architecture in greater detail; compares the performance with an optimized multi-threaded CPU implementation and GPU implementations; evaluates the energy consumption with actual measured dynamic power; and discusses the tradeoff between performance and energy consumption.

III. Option Pricing and Quadrature Methods

To understand option pricing with quadrature methods, we first consider the Black and Scholes partial differential equation [11] for an option with an underlying asset following geometric Brownian motion:

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

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

The following standard transformations:

$$x=\log\left(\frac{S_t}{E}\right),\quad y=\log\left(\frac{S_{t+\Delta t}}{E}\right)$$

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

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

where

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

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

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

Equation (2) contains an integral which cannot be evaluated analytically. Although for European options they can be converted to the probability density function for the 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 (2) iteratively.

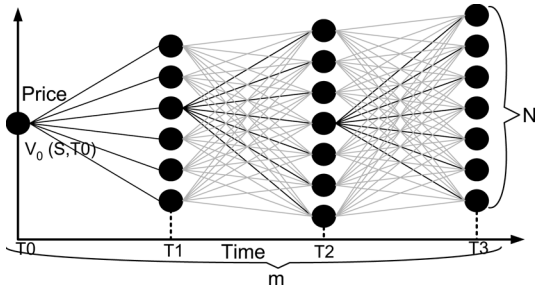

Fig. 1. Backward iteration process.

There are many different methods of numerical integral evaluation. Two of the most common methods include the trapezoidal rule and Simpson's rule [12]:

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

$$\int_a^b f(y)dy\approx\frac{\delta y}{2}\{f(a)+2f(a+\delta y)+2f(a+2\delta y)\cdots+2f(b-\delta y)+f(b)\}. \quad (6)$$

Simpson's rule: This is the most popular method for approximating integrals. It converges at a rate of $(\delta y)^4$. The approximation equation is:

$$\int_a^b f(y)dy\approx\frac{\delta y}{6}\left\{f(a)+4f\left(a+\frac{1}{2}\delta y\right)+2f(a+\delta y)+\cdots+2f(b-\delta y)+4f\left(b-\frac{1}{2}\delta y\right)+f(b)\right\}. \quad (7)$$

IV. Parallel Architecture

Using quadrature methods from (6) or (7), the option value $V(x,t)$ from (2) can be computed as

$$\begin{aligned}V(x,t)&=A(x)\int_{-\infty}^{+\infty}B(x,y)V(y,t+\Delta t)dy\\&\approx A(x)I_{oc}\sum_{i=0}^{N}I_i\cdot B(x,y_{\min}+i\delta y)\\&\quad\times V(y_{\min}+i\delta y,t+\Delta t).\end{aligned} \quad (8)$$

The sequence of integration coefficients I_i and the value of the outer integration coefficient I_{oc} depend on the type of quadrature method used. For example, the sequence of I_i is $1,2,2,\ldots,2,1$ and the value of I_{oc} is $\delta y/2$ for trapezoidal rule.

The major calculation part of $V(x,t)$ is the summation of $B(x,y_{\min}+i\delta y)V(y_{\min}+i\delta y,t+\Delta t)$ times I_i for all i. Similarly, the values of $V(y_{\min}+i\delta y,t+\Delta t)$ are computed by the summation of $B(y_{\min}+i\delta y,y_{\min}+j\delta y)V(y_{\min}+j\delta y,t+2\Delta t)$

TABLE I
PRICING EQUATIONS FOR VARIOUS TYPES OF OPTIONS

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

TABLE II
COMPUTATIONAL COMPLEXITY FOR SOME EXAMPLE OPTIONS. N DENOTES THE NUMBER OF INTEGRATION GRID POINTS AND m DENOTES THE NUMBER OF TIME STEPS

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

times I_j for all j. Therefore, the computation is a backward iterative process from the maturity time. A graphical representation of the process is illustrated in Fig. 1.

The value of δy determines the density of the integration. As the underlying asset follows a lognormal distribution and the change of price exhibits Brownian motion, the value of y fluctuates proportional to $\sqrt{\Delta t}$. As a result, we define the grid density factor $K1$ from

$$\delta y = \frac{\sqrt{\Delta t}}{K1}. \tag{9}$$

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

It is not possible to integrate a function from $-\infty$ to $+\infty$ numerically in practice. Therefore, the quadrature methods evaluate from a sufficiently small value $y_{\min}$ to a sufficiently large value $y_{\max}$. We define the grid size factor $K2$ from

$$y_{\max} = x + K2 \cdot \sigma\sqrt{\Delta t} \tag{10}$$
$$y_{\min} = x - K2 \cdot \sigma\sqrt{\Delta t}. \tag{11}$$

As a result, a large value of $K2$ leads to a large value of $y_{\max}$ and a small value of $y_{\min}$, resulting in a wide grid. $K2$ can also be viewed as the number of standard deviations from y to the original position of x after Δt.

Table I shows some of the pricing equations for different option types. The pricing equations are slightly different in terms of the integration range and the evaluation flow. For discrete barrier 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 II shows the computational 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.

A key result from Tables I and II 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 lower than $B(x,y)$ from Table II.

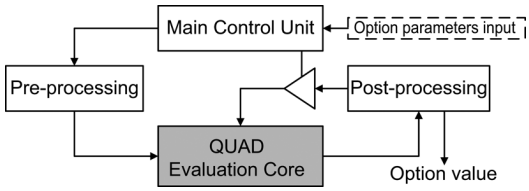

Fig. 2. System architecture of a generic option valuation system based on quadrature methods.

Our system architecture is not designed for pricing a specific option, so the flexibility to support 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 [4], [13]. 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. As a result, our system is designed to provide efficiency in hardware evaluation of the integral and flexibility for a general option pricing framework, as illustrated in Fig. 2.

The system architecture of the generic option valuation system using quadrature method is shown in Fig. 2. The architecture consists of the following components: 1) a preprocessing block; 2) one or more quadrature (QUAD) evaluation cores; 3) a post-processing block; and 4) a main control unit. Data input to the system are: $K1$, $K2$, T, So, E, r, D_c, σ, option-type, and option-specific-parameters. The option-type and option-specific-parameters provide the flexibility to support the pricing of multiple types of options. For example, we could specify the number of periods (m) and the knock-out/knock-in prices (b) for barrier options.

A typical option evaluation flow is illustrated in Fig. 3. The main control unit accepts the basic option input, selects the corresponding option evaluation equation and coordinates with the preprocessing and post-processing blocks. The preprocessing

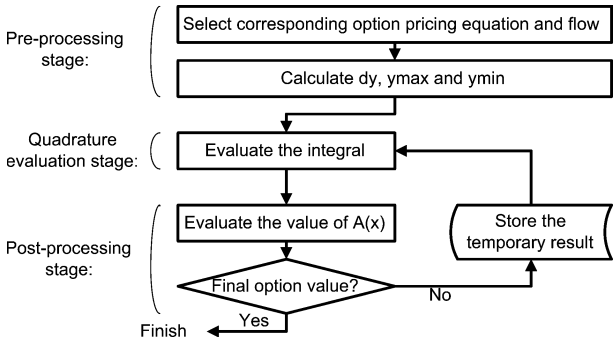

Fig. 3. Option evaluation flow.

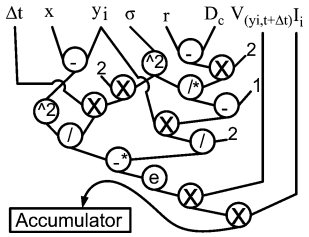

Fig. 4. Operator tree diagram for straightforward design (the operator with "$*$" denotes the operation from right to left).

block 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 evaluation cores. The QUAD evaluation cores evaluate the integral value based on (8). 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.

The QUAD evaluation core is implemented in hardware for three main reasons. First, more than one QUAD evaluation core fits 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 in Table II, the evaluation of the quadrature is the computation bottleneck, which would benefit from hardware acceleration.

The main control unit, preprocessing and post-processing blocks are implemented in software for the following reasons: 1) it increases the flexibility to support other options and 2) the evaluation in preprocessing and post-processing blocks is not the performance bottleneck; implementing them in hardware would not improve performance significantly.

The proposed architecture offers fast and parallel hardware cores for repeated numerical integrations, while supporting a versatile option evaluation platform.

A straightforward way of optimizing the QUAD evaluation core is to create a tree of pipelined operators. Fig. 4 shows an operator tree based on (2)–(7).

In Fig. 4, $\Delta t,\ x,\ y_i,\ \sigma,\ r,\ D_c,\ V(y_i, t+\Delta t)$ and I_i are fed to the evaluation tree continuously. However, the straightforward implementation consumes a large amount of hardware resources as it requires many floating-point operators. The optimized design is shown in Fig. 5 and will be used to produce implementations on both FPGA and GPUs (see Section VI).

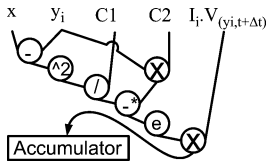

Fig. 5. Operator tree diagram for optimized design.

TABLE III
COMPARING THE ORIGINAL AND OPTIMIZED DESIGNS

	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	$3N+5$	$2N+3$

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

$$C1 = 2\sigma^2\Delta t, \quad C2 = \frac{(r-D_c)}{\sigma^2} - \frac{1}{2}. \tag{12}$$

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 precomputed in the preprocessing 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 an integration grid with N steps, the total number of parameters required is of the order $2N$, a 33% reduction from the original design which is of the order $3N$. Table III summarizes the differences between the original design and the optimized design.

V. MULTI-DIMENSIONAL QUADRATURE ANALAYSIS

To extend the design to support multiple underlying assets, we first consider the Black and Scholes partial differential equation [14] for an option with all underlying assets following geometric Brownian motion:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\sigma_i\sigma_j\rho_{ij}S_iS_j\frac{\partial^2 V}{\partial S_i \partial S_j} + \sum_{i=1}^{d}(r-D_i)S_i\frac{\partial V}{\partial S_i} - rV = 0 \tag{13}$$

with the logarithmic transformations of $x_i = \log(S_i)$ to be the chosen nodes at t and $y_i = \log(S_i)$ to be the chosen nodes at $t+\Delta t$. Let $\mathbb{R}$ be the matrix such that element $\mathbb{R}_{ij} = \rho_{ij}$. The solution is

$$V(x_1,\ldots,x_d,t) = A\int_{-\infty}^{+\infty}\cdots\int_{-\infty}^{+\infty} V(y_1,\ldots,y_d,t+\Delta t) \times B(x_1,\ldots,x_d,y_i,\ldots,y_d)dy_1\ldots dy_d \tag{14}$$

where

TABLE IV
COMPUTATION COMPLEXITY FOR SOME EXAMPLE MULTI-DIMENSIONAL OPTIONS

Option type:	Number of integration	Number of evaluation of B	Total number of floating-point operations
European options	$O(1)$	$O(N^d)$	$O(N^d(2d^2+4d+1))$
Barrier options	$O(N^d m)$	$O(N^{2d} m)$	$O(N^{2d}(2d^2+4d+1)m)$
American options	$O(N^d m)$	$O(N^{2d} m)$	$O(N^{2d}(2d^2+4d+1)m)$

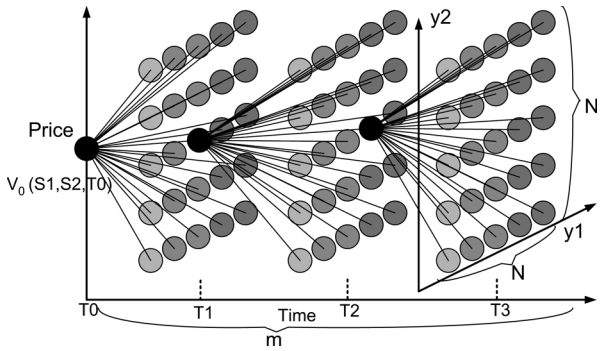

Fig. 6. Iteration process of a 2-D barrier option.

$$B(x_1,\ldots,x_d,y_i,\ldots,y_d) = \exp\left(-\frac{1}{2}\boldsymbol{\alpha}^T\mathbb{R}^{-1}\boldsymbol{\alpha}\right) \quad (15)$$

$$\alpha_i = \frac{(x_i - y_i + C1_i)}{C2_i}. \quad (16)$$

Equation (14) is the fundamental equation for multi-dimensional option pricing containing an integral which cannot be evaluated analytically. The number of dimensions for this integration is given by the total number of assets. $C1_i$ and $C2_i$ will be calculated in the preprocessing stage to improve performance

$$C1_i = \left(r - D_i - \frac{\sigma_i^2}{2}\right)\Delta t, \quad C2_i = \sigma_i(\Delta t)^{1/2}. \quad (17)$$

All quadrature methods discretize the continuous integration range into a set of grid points. The function value $f(y)$ is evaluated at these grid points and multiplied with integration coefficients. As from (7), the integration coefficients for Simpson's rule are $\{1,4,2,4,2,\ldots,2,4,1\}$. Under multi-dimensional quadrature methods, the product rule is used to determine the coefficient. The effective integration coefficients are calculated by the product of all original integration coefficients in their corresponding dimensions.

Fig. 6 shows the graphical representation of the iteration process of a 2-D barrier option pricing. We define N as the number of possible values (grid points) for y_1 and assume the number of grid points for all y_i is the same. We define d as the number of dimensions and m as the number of time intervals. For each time step, the number of integrations required is equal to the number of grid points, which is N^d. As a result, the total number of integrations required for multiple time steps American options and Barrier options is $N^d m$. The total number of evaluations of B is $N^d m \times N^d = N^{2d} m$.

Next, consider complexity analysis of our designs. The optimized number of operators required for the calculation of column matrix α is d for $(-)$ operator, d for $(+)$ operator and d for $(\div)$ operator. For matrix multiplication of $\alpha^T R^{-1}\alpha$ in (15), the number of $(\times)$ operators required is $d(d+1)$ and the number of $(+)$ operators required is $(d-1)(d+1)$. The rest of (15) requires one more $(\times)$ operator and one more exponential operator. Table V shows the summary of operator requirement for the evaluation of $B(x_1,\ldots,x_d,y_i,\ldots,y_d)$ and Table IV shows the summary of computation complexity for some example options.

TABLE V
OPERATORS COUNT FOR THE EVALUATION OF B

Operator type:	**Count**
$+$	d^2+d-1
$-$	d
$\times$	d^2+d+1
$\div$	d
exp	1
Total:	$2d^2+4d+1$

The computation time can be estimated by assuming that a 10 GFLOPs processor is used (the peak performance of a Pentium 4 3.2 GHz CPU is around 6.4 GFLOPs) and all floating point operators take the same amount of time. It can be shown that pricing a European option with 7 underlying assets takes 14.7 days with this processor at peak performance; however, it takes over 5 years with 8 assets. Hence other methods, such as using a cluster of accelerators, are required for designs beyond 7 dimensions.

VI. FPGA AND GPU DESIGNS

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.

Fig. 7 shows a pipelined QUAD evaluation core based on the design in Fig. 5. The grey boxes denote the pipeline balancing registers that are allocated automatically by *HyperStreams*. The QUAD evaluation core produces the value of $B(x,y)$ in (2) for every clock cycle. For an FPGA running at 100 MHz, the QUAD evaluation core can produce 100 M partial integral values per second.

The most challenging part of the multi-dimensional design is to support an arbitrary number of dimensions. The hardware evaluation cores are completely different for different dimensions as the underlying logic and the number of pipeline stages are different. Our approach provides a generic architecture that produces hardware designs specialised for a given dimension.

The hardware QUAD evaluation core involves three major parts. The first part is the evaluation of the vector α from (16).

TABLE VI
LOGIC UTILIZATION OF QUAD EVALUATION CORE IN DIFFERENT DIMENSIONS. ASTERISK (∗) INDICATES THAT THE PLACE AND ROUTE PROCEDURE CANNOT BE COMPLETED

	FPGA - Virtex-4 xc4vlx160							
Precision	single						double	
Dimension	1	2	3	4	5	6	1	2
DSPs	34 (35%)	51 (53%)	75 (78%)	96 (100%)	96 (100%)	96 (100%)	96 (100%)	96 (100%)
LUTs	19,713 (14%)	32,053 (23%)	43,580 (32%)	60,058 (44%)	70,909 (52%)	94,545 (69%)	53,792 (39%)	84,063 (62%)
FFs	16,605 (12%)	22,418 (16%)	30,005 (22%)	41,466 (30%)	54,569 (40%)	72,515 (53%)	39,281 (29%)	67,361 (49%)
Slices	20,200 (29%)	27,970 (41%)	38,006 (56%)	51,862 (76%)	67,582 (99%)	67,582 (99%)	51,396 (76%)	100,963 (149%)
Clock Rate	100MHz	91.2MHz	89.7MHz	91.5MHz	88.0MHz	(*)	81.9MHz	(*)

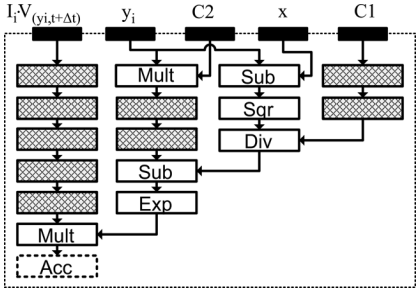

Fig. 7. Pipelined QUAD evaluation core for FPGA.

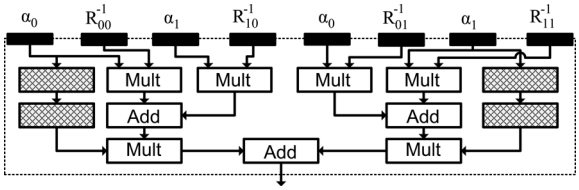

Fig. 8. Pipelined $\alpha^T R^{-1}\alpha$ design for 2-D QUAD evaluation.

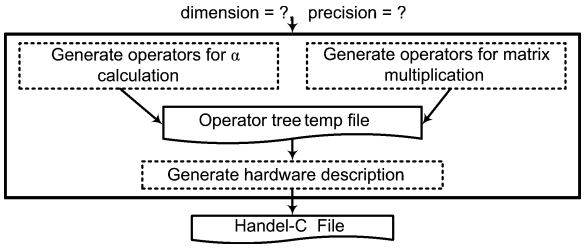

Fig. 9. Generating multi-dimensional QUAD evaluation.

The second part is the matrix multiplication of $\alpha^T R^{-1}\alpha$. The last part is the rest of the integration.

Fig. 8 shows a $\alpha^T R^{-1}\alpha$ design for 2-D QUAD evaluation. The $\alpha^T R^{-1}\alpha$ design becomes more complex for higher dimensions, with an increasing number of operators and pipeline stages. An evaluation core generator is developed to produce designs for different dimensions automatically.

The flow of the QUAD evaluation core generator is shown in Fig. 9. The generator accepts two input parameters: number of dimensions and the precision (single or double). An operator tree is generated and stored in a temporary file. Finally, the operator tree file is parsed and the corresponding *HyperStreams* and Handel-C codes are generated. The Handel-C code can then be compiled for simulation or bit-stream generation.

The operator tree generation consists of two main parts. The first part is the operator tree generation for the vector α calculation. It is generated according to (16) and replicated d times

```
void GPU_EvaluationCore()
{
   unique_thread_ID = Num of block * block_ ID + thread_ID_per_block
   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, and accumulate on local register;

   copy local register value to shared memory
   Synchronize with all other threads.
   if  (thread_ID_per_block==0)              // the first thread in each 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.
}
```

Fig. 10. CUDA pseudo code for QUAD evaluation kernel.

with respect to dimension d. Therefore, the logic resources required grow proportionally to d for the α calculation part. The second part is the operator tree generation for the matrix multiplication $\alpha^T R^{-1}\alpha$ from (15). The numbers of $(\times)$ and $(+)$ operators required for this matrix multiplication are $d(d+1)$ and $(d-1)(d+1)$, respectively. Therefore, logic resources required grow proportionally to d^2. Finally, the operator trees from the above two parts are combined with the rest of the quadrature operators.

Table VI shows the FPGA device utilization figures for the QUAD evaluation core in different dimensions and precisions. The targeted FPGA is Xilinx Virtex-4 xc4vlx160 and the designs are compiled using DK5.1 and Xilinx ISE 10.1. The result indicates that the FPGA device is fully utilized for 1 dimension under double-precision and is fully utilized for 5 dimensions under single-precision. The result also shows that for 1 dimension, multiple QUAD evaluation cores could be fitted into a single FPGA in order to exploit parallelism.

GPUs have been used for accelerating various applications [15], [16] . Our implementation on GPUs is based on CUDA API for nVidia GPUs [17].

Under CUDA, a function can be compiled into a "kernel". Each computation grid consists of a grid of thread blocks. The "kernel" is executed by all threads in parallel. Each block has a unique ID; so has each thread.

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 addition, the 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

TABLE VII
PERFORMANCE AND ENERGY CONSUMPTION COMPARISON OF DIFFERENT IMPLEMENTATION OF 1-D QUAD EVALUATION CORE. THE GEFORCE 8600 GT HAS 32 PROCESSORS, THE TESLA C1060 HAS 240 PROCESSORS AND THE XEON W3505 HAS TWO PROCESSING CORES

	FPGA		GPU			CPU
	Virtex-4 xc4vlx160		Geforce 8600GT	Tesla C1060		Xeon W3505
Technology	90nm		80nm	65nm		45nm
Release date	Sep 2004		Apr 2007	Sep 2008		Mar 2009
Arithmetic	single	double	single	single	double	double
Clock Rate	100MHz	81.9MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	3	1	-	-	-	-
Processing Speed (M values/sec)	300.0	81.9	114.4	546.5	288.7	65.3
Time for 10^9 values (s)	3.3	12.21	8.74	1.83	3.46	15.31
Acceleration (replicated cores)	**4.59x**	**1.25x**	**1.75x**	**8.37x**	**4.42x**	**1x**
APCC for 10^9 values (W)	4.18	3.3	40.55	102.00	99.00	23.60
AECC for 10^9 values (J)	13.93	40.29	354.42	186.64	342.92	361.30
Normalized energy efficiency	**25.93x**	**8.97x**	**1.02x**	**1.94x**	**1.05x**	**1x**

TABLE VIII
PERFORMANCE AND ENERGY CONSUMPTION COMPARISON OF DIFFERENT IMPLEMENTATION OF 2-D QUAD EVALUATION CORE

	FPGA	GPU			CPU
	Virtex-4 xc4vlx160	Geforce 8600GT	Tesla C1060		Xeon W3505
Arithmetic	single	single	single	double	double
Clock Rate	91.2MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	1	-	-	-	-
Processing Speed (M values/sec)	91.20	95.50	509.60	284.40	50.82
Time for 10^9 values (s)	10.96	10.47	1.96	3.52	19.68
Acceleration	**1.79x**	**1.88x**	**10.03x**	**5.60x**	**1x**
APCC for 10^9 values (W)	2.64	37.02	91.00	83.00	21.16
AECC for 10^9 values (J)	28.95	387.64	178.57	291.84	416.37
Normalized energy efficiency	**14.38x**	**1.07x**	**2.33x**	**1.43x**	**1x**

main thread then adds up all the results from all the blocks. The CUDA pseudo code for the QUAD evaluation kernel is shown in Fig. 10. The grid size and block size is set to 60 and 256, respectively. Registers per thread is 16 and the occupancy of each multiprocessor is 100%.

VII. EVALUATION AND COMPARISON

In this section, the performance and energy consumption of different implementations of QUAD evaluation core are studied. We choose the pricing of 1000 European options with grid density factor $K1 = 400\,000$ and grid size factor $K2 = 10$ as the benchmark. 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 option pricing of the order N^2m as shown in Table II or $N^{2d}m$ in multi-dimensional cases. 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 performance analysis. The performance and energy consumption analysis for the pricing of 1-underlying, 2-underlying, and 3-underlying assets European options are studied.

The FPGA and GPU implementations are compared to a reference software implementation. The reference CPU is Intel Xeon W3505 2.53 GHz dual-core processor. The software implementation is written using C language. It is optimized with multi-threading using OpenMP API and compiled using Intel compiler (icc) 11.1 with -O3 maximum speed optimization option and SSE enabled. Intel Math Kernel Library is used. The targeted FPGA is Xilinx Virtex-4 xc4vlx160 in the RCHTX card. The designs are compiled using DK5.1 and Xilinx ISE 9.2. The targeted GPU is nVidia Geforce 8600 GT with 256 MB of on board RAM and nVidia Tesla C1060 with 4 GB of on board RAM. The time measured for the GPU is the execution time of the evaluation kernel only. The time for copying the data from the main memory to the global memory of GPU is excluded. Similarly, the date transfer time for copying the data from main memory to the block RAM of FPGA is excluded. The performance figures obtained reflect the pure processing speed of the underlying devices only.

We measure the additional power consumption for computation (APCC) with a power measuring setup involving multiple equipments. A FLUKE i30 current clamp is used to measure the additional AC current in the live wire of the power cord during the computation. This current clamp has an output sensitivity of $S = 100$ mV/A in $\pm$ 1 mA resolution. The output of the clamp is measured in mV scale by a Maplin N56FU digital multi-meter (DMM), collected through a USB connection and logged with open source QtDMM software. APCC is defined as the power usage during the computation time (run-time power) minus the power usage at idle time (static power). In other words, APCC is the dynamic power consumption for that particular computation. Since the dynamic power consumption fluctuates a little, we take the average value of dynamic power to be the APCC.

The additional energy consumption for computation (AECC) is defined by the following equation:

$$\text{AECC} = \text{APCC} \times \text{Total Computational Time}. \quad (18)$$

TABLE IX
PERFORMANCE AND ENERGY CONSUMPTION COMPARISON OF DIFFERENT IMPLEMENTATION OF 3-D QUAD EVALUATION CORE

	FPGA	GPU			CPU
	Virtex-4 xc4vlx160	Geforce 8600GT	Tesla C1060		Xeon W3505
Arithmetic	single	single	single	double	double
Clock Rate	89.7MHz	1.35GHz	1.3GHz	1.3GHz	3.6GHz
Replicated cores	1	-	-	-	-
Processing Speed (M values/sec)	89.70	81.70	489.20	272.80	33.69
Time for 10^9 values (s)	11.15	12.24	2.04	3.67	29.68
Acceleration	**2.66x**	**2.43x**	**14.52x**	**8.10x**	**1x**
APCC for 10^9 values (W)	3.08	38.74	91.00	87.00	19.8
AECC for 10^9 values (J)	34.34	474.17	186.02	318.91	587.71
Normalized energy efficiency	**17.12x**	**1.24x**	**3.16x**	**1.84x**	**1x**

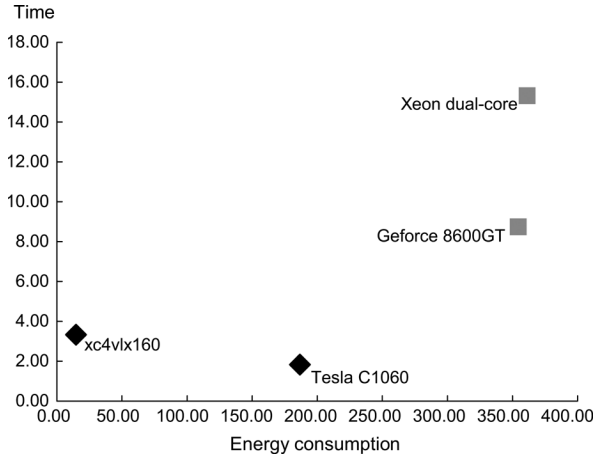

Fig. 11. Computational time and energy consumption relationship of different devices.

Therefore, AECC measures the actual additional energy consumed for that particular computation.

The summary of the performance comparison of 1-D, 2-D, and 3-D QUAD evaluation core is shown in Tables VII–IX.

From the results of Table VII for 1-D case, it can be seen that the FPGA implementation on the xc4vlx160 achieved 4.59 times acceleration using single-precision with 3 replicated QUAD cores and achieved 1.25 times acceleration using double-precision. For GPUs, a speedup of 1.75 times is achieved by Geforce 8600 GT and a speedup of 8.37 times is achieved by Tesla C1060 in single-precision. In double-precision, the Tesla C1060 has shown a 4.42 times speedup over the reference CPU, while there is no double-precision support in the Geforce 8600 GT.

It is not surprising that Tesla C1060 outperforms xc4vlx160, since Tesla C1060 is based on 65-nm fabrication technology while Virtex-4 xc4vlx160 is based on 90-nm fabrication technology. It would be fair to compare Virtex-4 FPGA with Geforce 8600 GT GPU because of similar fabrication technology. Xeon W3505 is selected to be a CPU reference because it represents the processing power of most workstations and it has a similar architecture to the latest CPU. We included a set of comparable devices—Virtex-4 FPGA, Geforce 8600 GT GPU and Xeon W3505 CPU. We estimated that Virtex-5 FPGA performs at least 4 times faster than Virtex-4 as Virtex-5 has 4 times more slices than Virtex-4 and with higher clock frequency. We found that Tesla C1060 GPU is more than 4 times faster than Geforce 8600 GT from Table VII. We also estimate that the performance of the latest Intel Core i7 CPU will be around 4 times faster than Xeon W3505 according to their number of cores and frequency ratios.

From Tables VIII and IX, it can be seen that the performance of xc4vlx160 FPGA in 2-D and 3-D cases is not as good as in 1-D case. The reason is that the xc4vlx160 FPGA is fully utilized in 1-D case with three replicated QUAD evaluation cores. However, only one QUAD evaluation core can be fitted in the xc4vlx160 FPGA in 2-D and 3-D cases and there are many unused logic resources. From this point of view, we can conclude that an algorithm with a smaller computation core is more suitable to FPGA because it is easier to replicate multiple smaller computation cores to fully utilize the resources in the FPGA. The worst scenario, like our 2-D case, involves a computation core that consumes just above 50% FPGA resources; it precludes replication so possibly wasting resources.

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 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 [18]. However, it is not applicable to quadrature methods as the range of the numerical values spreads widely from small size partial integral values to large size complete integral values.

Next, consider the energy efficiency of different devices. It is interesting to note that the xc4vlx160 FPGA demonstrates the greatest energy efficiency regardless of the technology differences. In single dimension case, xc4vlx160 is 25.9 times more energy efficient than Xeon W3505, 25.4 times more energy efficient than Geforce 8600 GT and 13.4 times more energy efficient than Tesla C1060. An interesting observation is that xc4vlx160 FPGA demonstrates similar level of energy efficiency with the 1-D case in 2-D and 3-D cases, despite the decrease of computational performance.

Fig. 11 shows a scatter plot graph of the computation time versus the energy consumption (AECC) of different devices im-

plementing the 1-D QUAD evaluation core. From this graph, the highest computational performance is achieved using Tesla C1060 GPU and the lowest energy consumption is achieved using xc4vlx160 FPGA. Therefore, Geforce 8600 GT and Xeon W3504 are considered to be inefficient for this application. Tesla C1060 and xc4vlx160 are the fastest and the most energy efficient respectively for this application.

VIII. Conclusion

This paper proposes a novel parallel architecture for hardware accelerated option pricing based on quadrature methods. Our proposal includes a highly pipelined datapath capable of supporting quadrature evaluation in parallel. We explore implementations for quadrature evaluation in FPGA and GPU technologies. A tool is developed for automatic production of hardware designs with a given number of dimensions.

The performance and energy consumption of FPGA and GPU implementations are compared against each other and compared against a multi-threaded software implementation on a CPU. The results show that FPGA implementation is 4.6 times faster than the CPU, 1.75 times faster than a GPU in comparable technology and 1.8 times slower than the latest GPU. In addition, the FPGA is up to 25 times more energy efficient than CPU and comparable GPU. The energy efficiency of FPGA against other devices in multi-dimensional cases is similar to the 1-D case.

Current and future work includes the design of a distributed quadrature evaluation framework in a heterogeneous cluster consisting of FPGAs, GPUs and CPUs for collaborative computing. The system targets the latest FPGAs such as Virtex-5, Virtex-6 and Stratix IV for improved performance and for comparison with the latest GPUs. 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, such as the solutions of electromagnetic problems [2], calculations involving photon distribution [3] and modeling of credit risk [1].

References

[1] M. H. A. Davis and J. C. Esparragoza-Rodriguez, "Large portfolio credit risk modeling," *Int. J. Theor. Appl. 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," *J. Computat. Phys.*, 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," in *Proc. IEEE Nucl. Sci. Symp. Conf.*, 2007, pp. 3548–3554.

[4] A. D. Andricopoulos, M. Widdicks, P. W. Duck, and D. P. Newton, "Universal option valuation using quadrature methods," *J. Financial Econom.*, vol. 67, no. 3, pp. 447–471, Mar. 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. Field-Program. Technol.*, 2005, pp. 215–224.

[6] D. Thomas, J. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *Proc. Int. Conf. Appl.-Spec. Syst., Arch., Processors*, 2007, pp. 168–173.

[7] G. Morris and M. Aubury, "Design space exploration of the European option benchmark using hyperstreams," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2007, pp. 5–10.

[8] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for tree-based pricing models," *ACM Trans. Reconfig. Tech. Syst.*, vol. 2, no. 4, 2009, Article No. 21.

[9] A. H. T. Tse, D. B. Thomas, and W. Luk, "Accelerating quadrature methods for option valuation," in *Proc. IEEE Symp. FPGAs for Custom Comput. Mach.*, 2009, pp. 29–36.

[10] A. H. T. Tse, D. B. Thomas, and W. Luk, "Option pricing with multi-dimensional quadrature architectures," in *Proc. Int. Conf. Field-Program. Technol.*, 2009, pp. 427–430.

[11] F. Black and M. S. Scholes, "The pricing of options and corporate liabilities," *J. Political Economy*, vol. 81, no. 3, pp. 637–54, May/Jun. 1973.

[12] E. Sueli and D. F. Mayers, *An Introduction to Numerical Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2006.

[13] G. Fusai and M. C. Recchioni, "Analysis of quadrature methods for pricing discrete barrier options," *J. Econom. Dyn. Control*, vol. 31, no. 3, pp. 826–860, Mar. 2007.

[14] A. D. Andricopoulos, M. Widdicks, D. P. Newton, and P. W. Duck, "Extending quadrature methods to value multi-asset and complex path dependent options," *J. Financial Econom.*, vol. 83, no. 2, pp. 471–499, 2007.

[15] L. Pan, L. Gu, and J. Xu, "Implementation of medical image segmentation in cuda," in *Proc. Int. Conf. Technol. Appl. Biomed.*, 2008, pp. 82–85.

[16] H. Jang, A. Park, and K. Jung, "Neural network implementation using cuda and openmp," in *Proc. Digit. Image Comput. Techn. Appl.*, 2008, pp. 151–161.

[17] NVidia, Santa Clara, CA, "Nvidia CUDA programming guide," 2008.

[18] G. A. Constantinides, "Word-length optimization for differentiable nonlinear systems," *ACM Trans. Des. Autom. Elect. Syst.*, vol. 11, no. 1, pp. 26–43, 2006.

Anson H.T. Tse (S'08) received the B.Eng. and M.Sc. degrees from the Chinese University of Hong Kong, Hong Kong, in 2005 and 2008, respectively. He is currently pursuing the Ph.D. degree from the Department of Computing, Imperial College London, London, U.K.

His research interests include reconfigurable computing, high performance computing, distributed computing, and computational finance.

Mr. Tse was a recipient of a Croucher Foundation Scholarship.

David Thomas (M'06) received the M.Eng. and Ph.D. degrees in computer science from Imperial College London, London, U.K., in 2001 and 2006, respectively,

Since 2010, he has been a Lecturer with the Electrical and Electronic Engineering Department, Imperial College London. His research interests include hardware-accelerated cluster computing, FPGA-based Monte Carlo simulation, algorithms and architectures for random number generation, and financial computing.

Wayne Luk (F'09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K.

He is a Professor of computer engineering with Imperial College London, London, U.K., and a Visiting Professor with Stanford University, Stanford, CA, and with Queens University Belfast, Belfast, U.K. His research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.