An Environment for Rapid Derivatives Design and Experimentation

Danny Crookes, *Senior Member, IEEE*, Sean Trainor, and Richard Jiang

***Abstract*—In the highly competitive world of modern finance, new derivatives are continually required to take advantage of changes in financial markets, and to hedge businesses against new risks. The research described in this paper aims to accelerate the development and pricing of new derivatives in two different ways. First, new derivatives can be specified mathematically within a general framework, enabling new mathematical formulae to be specified rather than just new parameter settings. This Generic Pricing Engine (GPE) is expressively powerful enough to specify a wide range of standard pricing engines. Second, the associated price simulation using the Monte Carlo method is accelerated using GPU or multicore hardware. The parallel implementation (in OpenCL) is automatically derived from the mathematical description of the derivative. As a test, for a Basket Option Pricing Engine (BOPE) generated using the GPE, on the largest problem size, an NVidia GPU runs the generated pricing engine at 45 times the speed of a sequential, specific hand-coded implementation of the same BOPE. Thus, a user can more rapidly devise, simulate, and experiment with new derivatives without actual programming**

***Index Terms*—Financial trading, high performance DSP, pricing engines.**

I. Introduction

DERIVATIVES pricing is the preserve of the Quantitative Analyst. In the highly competitive world of modern finance, new derivatives are continually required to take advantage of changesin financial markets, and to hedge businesses against new risks [1]. In order to obtain the greatest profit, or minimize risk exposure, first mover advantage is desired. The faster a company can attain beneficial positions through derivatives, the greater the benefit it can obtain in market cascades before all profit potential is taken away (or risk is increased too greatly) by other market participants copying the first movers [2]. Therefore the ability to model new market conditions quickly will put a financial company at an advantage.

There is a range of standard pricing engines which are used to model and simulate different scenarios. Options are a common special case of derivative, being based on the value of a stock. While this paper and our environment is applicable to other types of derivative (such as futures), most of the specific examples used are options. Options can be divided into two broad categories. Fixed exercise options (known as European options), allow the option holder to buy (for a call option) or sell (for a put option) the underlying asset(s) only at the expiration date of the option contract. Variable exercise options are known as American options when the option can be exercised at any time between the purchase time and expiry. Variable exercise options are known as Bermudan options when the option can be exercised at a fixed set of dates given in the option contract. In practice, American option prices are usually modelled by assuming a finite number of possible exercise dates. In this work we concentrate on fixed exercise options.

A. European Options

A European Call Option is an option on a single asset, and is non-path-dependent, which means that its payoff does not depend on values attained by the asset between the starting time and the time of expiry, but depends only on the final value. The price path is described by the Black-Scholes stochastic differential equation (SDE) [3]:

$$\frac{dS\left(t\right)}{S\left(t\right)} = rdt + \sigma dW\left(t\right) \tag{1}$$

where $S(t)$ is the stock price at time t, r is the interest rate, σ is the volatility of the stock price, and W is a standard Brownian motion. The term $\sigma dW(t)$ is known as the 'diffusion' term. This term models how the paths will spread over time, and the fact that it is a Brownian motion means that the changes in this term over a small time interval Δt are normally distributed with mean 0 and variance Δt.

The payoff at the time of expiry (T) of the option is:

$$\max\ \left\{0,\ \ S\left(T\right) - K\right\} \tag{2}$$

for a call option, and, for a put option:

$$\max\ \left\{0,\ \ K - S\left(T\right)\right\} \tag{3}$$

K is the strike price, which is a guaranteed price at which the asset can be bought (for a call option) or sold (for a put option) at time T. Eq. (2) and Eq. (3) make clear that the option will expire worthless if the option holder would lose money by trading in the underlying asset, i.e. buying the asset at the strike price and selling it in the market, in the call option case, and vice versa in the put option case.

To obtain the present value of the payoff, the final payoff value is discounted at a continuously compounded rate by multiplying by the discount factor e^{-rT}. The price $S(t)$ is a random variable

Manuscript received October 05, 2015; revised June 06, 2016; accepted July 11, 2016. Date of publication July 18, 2016; date of current version August 12, 2016. This work was supported by the Capital Markets Collaborative Network and Citi Belfast. The guest editor coordinating the review of this manuscript and approving it for publication was Dr. Emmanuelle Jay.

D. Crookes is with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT7 1NN, U.K. (e-mail: d.crookes@qub.ac.uk).

S. Trainor is with the Institute of Electronics, Communications and Information Technology, Queen's University Belfast, Belfast BT7 1NN, U.K. (e-mail: strainor05@qub.ac.uk).

R. Jiang is with the Department of Computer Science and Digital Technologies, Northumbria University, Newcastle upon Tyne NE1 8ST, U.K. (e-mail: richard.jiang@northumbria.ac.uk).

Digital Object Identifier 10.1109/JSTSP.2016.2592619

for all $t \in [0, T]$, and we wish to calculate its expected present value. The expected value $E[X]$ is the integral, over the range of possible values of X, of each value x multiplied by the probability density at x, taken with respect to x. If $f(x)$ is the probability density at x, then $E[X] = \int_{-\infty}^{\infty} x f(x) dx$, where $f(x)$ may be zero on some of this range.

The expected present value of the option can be written as

$$E\left[e^{-rT} \max\left\{S\left(T\right) - K,\ 0\right\}\right] \tag{4}$$

The distribution of the random variable $S(T)$ is given by the solution of the SDE (1), as follows

$$S\left(T\right) = S\left(0\right) \exp\left(\left(r - \frac{1}{2}\sigma^2\right) T + \sigma W\left(T\right)\right) \tag{5}$$

The random variable $W(T)$ is normally distributed, with mean 0 and variance T. Thus the log of the stock price is normally distributed and the stock price is lognormally distributed.

The expected present value is an integral with respect to the lognormal density of $S(T)$, and it can be evaluated using the Black-Scholes formula in terms of the standard cumulative normal distribution function Φ:

$$\begin{aligned} &B\left(S\left(0\right), \sigma, T, r, K\right) = \\ &S\Phi\left(\frac{\log\left(\frac{S(0)}{k}\right) + \left(R + \frac{1}{2}\sigma^2\right) T}{\sigma\sqrt{T}}\right) \\ &\quad -e^{-rT} K\Phi\left(\frac{\log\left(\frac{S(0)}{K}\right) + \left(r - \frac{1}{2}\sigma^2\right) T}{\sigma\sqrt{T}}\right) \end{aligned} \tag{6}$$

B. Asian Options

Asian options are path-dependent options on one underlying asset, and this path-dependence means that Monte Carlo methods are a good way to price them. An arithmetic Asian option's price depends on the average price of the underlying asset at a preselected number of time points during the lifetime of the option. The payoff of an arithmetic Asian option is:

$$P_A = \max\left\{0, \quad \frac{1}{m}\sum_{i=1}^{m} S\left(t_i\right) - K\right\} \tag{7}$$

for fixed times $0 = t_0 < t_1 < \ldots < t_m = T$. The starting time is 0, and T is the expiry time. The expected discounted payoff, i.e. the fair price for the option, is $E[e^{-rT} P_A]$.

The price at each step in the price path can be calculated from the previous one in the same way as the complete single step in the European option:

$$\begin{aligned} S\left(t_{i+1}\right) \ = \ & S\left(t_i\right) \exp\left(\left(r - \frac{1}{2}\sigma^2\right)\left(t_{i+1} - t_i\right)\right. \\ & \left. + \sigma\sqrt{t_{i+1} - t_i}\ Z_{i+1}\right) \end{aligned} \tag{8}$$

where $0 \le t_i < t_{i+1} \le T$, and Z_{i+1} is a sample from a multi-dimensional standard normal distribution.

Asian options have some advantages over European options. For example, the averaging reduces the overall volatility, thus reducing the risk for the option seller. Asian options are therefore cheaper for the option buyer. Asian options also reduce the risk of losses due to market manipulation, close to the exercise time, when compared with those which depend only on the underlying value at the exercise time. Asian option pricing equations can be considered a mathematical generalisation of the standard European call or put options since, if there is only one step, they are equivalent.

C. Basket Options

A Basket option is an option whose payoff depends on the value of multiple assets. The pricing equations for basket options are a generalisation (to multiple dimensions) of those for standard European or Asian options. Here we investigate basket options, on n assets, described by the equation system:

$$\frac{dS_i\left(t\right)}{S_i\left(t\right)} = r_i dt + \sigma_i dW_i\left(t\right) \tag{9}$$

where $i = 0, \ \ldots, \ \ n-1$, and W_i is the ith component of a n-dimensional Brownian motion. We consider the case of constant interest rates r_i and constant volatilities σ_i.

The price path for each asset can be calculated independently, using the same formula above (see Eq. (8)). Now, however, the random deviates $(Z_{i,j+1})$ are taken from a multi-normal distribution; more correctly, the $(\mathrm{j}+1)\mathrm{th}$ time step of the $i\mathrm{th}$ asset. The elements in these vectors are correlated (unless the distribution is standard multi-normal), and in order to perform the necessary correlation quickly, a parallel correlation function must be implemented.

The payoff could depend on a weighted average of the final asset price estimates, or on the maximum or minimum asset price estimate (which is the case for e.g. lookback options). In the case of the weighted average, the weighting used for an asset can depend on, for example, the performance of that asset, or on the quantity of each underlying asset in the basket.

With fixed parameter values and a fixed number of parameters, it is not possible to model many useful market scenarios. For example it has been shown that the volatility for Equity Index Options should be non-constant, and it depends on current price and time to expiry [4]. The payoff of some derivatives is based on interest rates, and many interest rate models have been developed for the pricing of these, with interest rates being modelled with SDEs, which for interest rate *r* take the form:

$$dr = g(r)dt + f(r)dW(t) \tag{10}$$

where g and f are deterministic functions of r. Examples of this are the Vasicek Model [5], or the CIR Model [6]. These models are used for the pricing of derivatives such as Callable Bonds [7] (bonds that can be bought back by the issuer for a predetermined price at predetermined times). In some cases these interest rate models may be used to simulate the interest rate in other derivatives with more complex price path models, and simulations of this type are possible with the generic pricer presented later in this paper.

Derivative specifications presented to (or possibly by) the option buyer will usually only affect the payoff of the

derivative, and not the model used to simulate the market conditions. However, the payoff specification will determine whether information about the values taken at particular points in the price path need to be recorded for later use, or if those values should be used on-the-fly. For example, an option with Asian payoff characteristics (i.e. the payoff depends on the average value of the assets during the derivative's lifetime) will require the specification of time points, and will require on-the-fly summation (for Arithmetic Asian Options) or multiplication (for Geometric Asian Options) of the prices at those points for the purpose of calculating the average value taken on each price path. If Lookback characteristics are specified, the maximum or minimum value taken along the price paths will need to be recorded. The payoff specification and an appropriate model of market conditions will allow the derivative seller to determine an estimate of the correct price of the derivative. How good the price estimate is will depend on the quality of the model used for the underlying asset price dynamics, and not on the payoff specification. Therefore we concentrate on the generalisation of the asset price path model to allow for better fitting to the conditions in the market.

In this paper we present a system in which the user can interactively specify a derivative by entering the price path model (within the constraints of a general model), specifying any required new parameters, and the time-discretised formula for each of the parameters. The generalised system (the Generic Pricing Engine, GPE) can estimate the solution of any SDE whose solution can be simulated using an Euler Scheme. The equations' parameters may also follow a stochastic process, or be time or state dependent, as long as they can be discretised in the time dimension. This allows for the simulation of many of the stochastic models used in financial engineering. It should also enable the system to be used for non-financial applications, such as simulating the movement of small particles.

The paper is structured as follows. Section II presents the basis for our generic model and the schemes for its (approximate) simulation. Section III then presents our GPE and the various inputs (user-defined formulae, etc.) which the user can specify to define a new derivative. Section IV discusses a range of stochastic processes where our GPE can give an exact simulation. Section V begins by reviewing existing work on parallelization of financial simulations. It then presents an outline (sequential) implementation of the simpler, and specific, BOPE, leaving Section VI to present the parallel GPE implementation approach. Section VII gives performance results for a range of problem sizes, architectures and coding languages. The main conclusions are summarized in Section VIII.

II. The Generic Model and Its Simulation

Glasserman [8] investigated a more general derivatives pricing model which encompasses all the previously described models, and more. He states that most models in financial engineering can be described by an SDE of the following form:

$$dX\left(t\right) = a\left(X\left(t\right)\right)dt + b\left(X\left(t\right)\right)dW\left(t\right). \tag{11}$$

Typically, $X(t)$ is the price at time t, $dX(t)$ is the predicted change in price of the subsequent time period dt, so that $X(t+dt) = X(t) + dX(t)$. a is called the *drift* term (in the simple case, the interest rate), and b is the *diffusion* term (in the simple case, the volatility). The function a is typically applied to a set of n asset prices plus the time t, and produces a new set of n values; the function b likewise takes a set of n asset prices plus the time t, and produces a new set of $n \times k$ values. W is a k-dimensional Brownian motion (actually it can be a more general Levy process). Over a time period, *dW*(t) is essentially a set of normally distributed random numbers. We use Eq. (11) as the basis of our GPE.

In the general case, it is not possible to simulate the solution to Eq. (11) (which is a stochastic process) exactly (see Section IV for some exceptions). Hence all these methods usually entail some discretisation error, and there is usually a trade-off between the rate of convergence and the computational complexity: more steps in the price paths will mean more accuracy and a faster error convergence rate.

Here we review two common schemes—the Euler and Milstein Schemes—and explain the choice of Euler.

The solution to Eq. (11) (with certain technical restrictions) can be simulated approximately by using an Euler scheme. An Euler scheme is the simplest method to simulate SDEs and, apart from its simplicity, one of its benefits is almost universal applicability. The idea is to divide the whole time interval of interest into a discrete time grid and then to simulate a discrete process to approximate the original continuous-time SDE on the time grid according to its finite-difference counterparts. The Euler Scheme to solve this equation takes the form:

$$\begin{aligned} X\left(t_{i+1}\right) &= X\left(t_i\right) + a\left(X\left(t_i\right)\right)\left(t_{i+1} - t_i\right) \\ &\quad + b\left(X\left(t_i\right)\right)dW\left(t_i\right) \end{aligned} \tag{12}$$

where $dW(t_i) = sqrt(dt)Z_{i+1} \approx \sqrt{t_{i+1} - t_i}Z_{i+1}$, with Z_i being a sample from a multi-dimensional standard Normal distribution for every i.

The Euler Scheme method is actually derived from the Taylor Expansion of the SDE (see Eq. (11)), by keeping just the first three terms in the Taylor expansion, giving Eq. (12).

The alternative Milstein scheme, which in some cases may give better convergence than the Euler Scheme, is obtained by keeping the first four terms of the Taylor expansion. This scheme is of order one in both the drift and diffusion components. A problem with the Milstein Scheme is that it requires the calculation of the derivative of the function b, which may be computationally expensive.

III. A Generic Pricing Engine

We use Glasserman's equation, Eq. (11), as the basis for our GPE. $X(t)$ is the value, at time t, of something whose price is being modelled (e.g. an underlying asset). Thus $dX(t)$ is the infinitesimal change in the value X over time dt. To model the constrained randomness of X, we assume $X(t)$ depends on a random process $W(t)$, whose 'steps', $W(t_{i+1}) - W(t_i)$, are normally-distributed. Thus $dX(t)$ will depend on $dW(t)$. More generally, Eq. (11) enables us to extend the model to enable the

price paths to depend on some function of X and/or t. Thus in Eq. (11), a and b are functions which model the non-random and random processes respectively. The user can define formulae for these functions, for each of the *d* assets. This generic model can not only describe the stochastic price path of the underlying assets of a derivative, but it can also describe the path of a stochastic variable representing, for example, interest rates or volatilities, which may be required in cases of assets with very complex price path dynamics.

In defining our GPE, we provide essentially the Euler scheme in Section II above, but generalised a little to allow the option of multiplying the factors instead of always simply adding them. Also, we allow the option of exponentiating the final factor. Thus the GPE allows anything that can be written in the form:

$$X\left(t_{i+1}\right) = X\left(t_i\right) \oplus a\left(X\left(t_i\right), t_i\right) \oplus \{e\}\left(b\left(X\left(t_i\right), t_i\right) Z_{i+1}\right) \tag{13}$$

where a is a vector of functions (defining the drift, as described above), and b is a matrix of functions (defining the diffusion). The functions can be any functions of $X(t_i)$ and t_i, provided they can be computed by equivalent functions in the implementation environment. $\oplus$ can be either the multiplication or addition operator, and $\{e\}$ symbolises that this last part of the equation can optionally be chosen to be exponentiated. Each function can also have random factors using normally distributed random values, and can use the size of the time interval $t_{i+1} - t_i$. Z_{i+1} is a vector sample from a multidimensional standard normal distribution.

The main part of the GPE is the path simulation. However, the definition of the derivative price itself is:

$$\text{price} = \text{discount factor} \times \text{payoff} \tag{14}$$

where the payoff depends on the path simulation results. The GPE implements general path simulation capabilities.

A. Using the GPE: The User Interface

Our system has a user interface which enables the user to specify a derivative interactively, by typing all the parameters, asset starting values, options and functions required for Eq. (13). Functions are typed as text, selected from a wide set of predefined functions which can be used. To model a derivative, the user enters the following:

1) The initial values for the state (price) vector X.
2) If using strike values, these must be specified, plus whether the derivative is a 'put' or a 'call'.
3) Whether to use additive or multiplicative steps.
4) A vector of functions (*a*) defining the drift (one per asset)
5) Whether to exponentiate the diffusion component.
6) A $k \times k$ matrix of functions (*b*) defining the diffusion.
7) Whether to use final, average, max or min path values. Examples which use final path values are European options on a stock or basket of stocks. Asian options use average path values, and lookback options use maximum or minimum path values.
8) Whether to use a sum, max or min reduction across paths. The sum would normally be used to calculate the average (expected) payoff of a derivative, whereas the maximum or minimum would be used to check the extreme values—possibly for error checking.
9) Whether to use average, max or min across final asset prices. Some basket options use a weighted average of the values for each underlying asset. A derivative whose payoff depends on the best or worst performing asset would use maximum or minimum here.

For convenience, it is possible to specify that all the drift or diffusion functions are the same for all assets (with the matrix b being diagonal)—in which case only one function needs to be specified for drift or for diffusion.

When entering a more complex function, if at any point a new working variable or function is needed, the user enters the keyword 'NEW', and is then prompted each time to type in the variable or function to be used at this point. Later, these formulae will be integrated and converted to code.

IV. GPE for Methods With Exact Simulation

The Euler Scheme and its refinements and extensions, detailed above, can be used to simulate a wide variety of stochastic processes. However, these methods entail some discretisation error. In certain situations, though, it is possible to simulate exactly, without discretisation error. To demonstrate the power and flexibility of our GPE, we now give three examples of exactly simulable processes which can be specified in our GPE, and show the settings necessary to obtain these.

A. Brownian Motion

A multi-dimensional stochastic process $\{W(t),\ 0 \le t \le T\}$ is called a standard Brownian motion on $\mathbb{R}^n$ if:

1) $W(0) = 0$
2) The mapping $t \mapsto W(t)$ is continuous on $[0,\ T]$
3) The increments $W(t_0),\ W(t_1) - W(t_0), \ldots, W(t_k) - W(t_{k-1})$ are independent for all $0 \le t_0 < t_1 < \cdots < t_k \le T$
4) $W(t) - W(s) \sim N(0, (t-s)I_n)$ for all $0 \le s < t \le T$, and I_n is the $n \times n$ identity matrix.

The paths of standard Brownian motion can be simulated by setting $W(0) = 0$, sampling $Z_1, Z_2, \ldots, Z_m$ independently from $N(0, I_n)$ and using the following algorithm:

$$W\left(t_{i+1}\right) = W\left(t_i\right) + \sqrt{t_{i+1} - t_i} Z_{i+1}. \tag{15}$$

A general Brownian motion $X(t)$, which is different from standard in that its increments follow a general multi-Normal distribution, can be simulated as follows:

Let $\mu \in \mathbb{R}^n$ be the mean of the multi-Normal distribution, and let Σ be its covariance matrix (it must be symmetric and positive semi-definite, and we consider only the positive definite case). Find, by Cholesky factorisation (to obtain a lower-triangular matrix and thereby reduce the number of multiplications and additions needed), a matrix A such that $AA^T = \Sigma$. If A is $n \times k$, let $Z_1, \ldots, Z_n$ be independent standard Normal random vectors in $\mathbb{R}^k$, and use the algorithm:

$$X\left(t_{i+1}\right) = X\left(t_i\right) + \left(t_{i+1} - t_i\right)\mu\ + \sqrt{t_{i+1} - t_i} A Z_{i+1}. \tag{16}$$

For greater generality, the parameters can be time-dependent: $\mu(t)$ and $A(t)$. This can be modelled in the GPE as follows:

1) Set the $X(t_0)$ values
2) Select additive steps
3) Set each 'a' function to be 'constant' $\times D$ (D stands for $\Delta t_i = t_{i+1} - t_i$)
4) Set each 'b' function to be 'constant '$\times\sqrt{D}$.

B. Gaussian Short-Rate Models

In some cases, instead of being constant or deterministically time varying, it is beneficial to model the interest rate with a stochastic process. Some of the most commonly used stochastic processes for this are Gaussian processes.

A general class of Gaussian process models, in $\mathbb{R}^n$, used for short rates is described by

$$dX\left(t\right) = C\left(b - X\left(t\right)\right)dt + DdW\left(t\right) \tag{17}$$

where C and D are $n \times n$ matrices, and b, $W(t)$, $X(t) \in \mathbb{R}^n$, and where the coefficients can also be deterministically time-varying. The short rate $r(t)$ can then be specified by $r(t) = a^T X(t)$, where a is constant or deterministically time-varying.

When C is non-singular and diagonalisable, a multi-dimensional simulation can be reduced to multiple independent scalar simulations, linked only through the correlation matrix from the $dW(t)$ term. It can also be reduced to scalar simulations when C is not diagonalisable, but all coefficients are deterministically time-varying.

This model encompasses other Gaussian short-rate models such as the Vasicek, Ho-Lee and Hull-White models. This can be modelled in the GPE as follows:

1) Set the $X(t_0)$ values
2) Select additive steps
3) Set each 'a' function to be: $(NEW \times X[0] + NEW \times X[1] + \cdots + NEW \times X[n-1] + NEW) \times D$
4) Set the constant (or time dependent) values for the 'NEW' variables when prompted by the UI.
5) Set each 'b' function to be '$NEW \times \sqrt{D}$' (and set the constant or time dependent values (or functions) for the 'NEW' variables when prompted).

C. Square-Root Diffusions

A one dimensional stochastic process $X(t)$ described by:

$$dX\left(t\right) = a\left(b - X\left(t\right)\right)dt + \sigma\sqrt{X\left(t\right)}dW\left(t\right) \tag{18}$$

is known as a square-root diffusion. Models of this kind have been proposed by Heston [9] as a model of the stochastic volatility of an asset, and also by Cox, Ingersoll and Ross [6] as a model of the short rate.

Models of this kind can be simulated exactly by drawing samples from an appropriate non-central chi-squared distribution and a Poisson distribution, as well as the standard normal distribution. This exact simulation procedure, however, is difficult to parallelise, and the resulting code will most likely be slow; it would also require a large amount of memory to store pre-calculated random values. The exact simulation procedure is therefore not implemented in our pricer, and an Euler approximation is used instead:

$$\begin{aligned} X\left(t_{i+1}\right) \;=\; & X\left(t_i\right) + a(b - X\left(t_i\right)\left(t_{i+1} - t_i\right) \\ & + \sigma\sqrt{X(t_i)^+}\sqrt{t_{i+1} - t_i}Z_{i+1}. \end{aligned} \tag{19}$$

The multi-dimensional case, including when the underlying 1D processes are correlated, has been studied in [10]. They study processes of the form:

$$dX_t = \mu\left(X_t\right)dt + \sigma\left(X_t\right)dW_t \tag{20}$$

on a suitable state-space $D \subseteq \mathbb{R}^n$ (for some functions σ, D will be a strict subset), with $\mu : D \to \mathbb{R}^n$ and $\sigma\sigma^T : D \to M_S$ affine (linear in X_t), where M_S is the space of real symmetric $n \times n$ matrices. It is shown that Eq. (24) can be expressed as:

$$\begin{aligned} dX_t = {} & (aX_t + b)\,dt \\ & + \Sigma \begin{pmatrix} \sqrt{v_1\left(X_t\right)} & 0 & \cdots & 0 \\ 0 & \sqrt{v_2\left(X_t\right)} & \cdots & 0 \\ & & \ddots & \\ 0 & \cdots & 0 & \sqrt{v_n\left(X_t\right)} \end{pmatrix} dW_t \end{aligned} \tag{21}$$

where $a \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, $\Sigma \in \mathbb{R}^{n\times n}$, and $v_i(X_t) = \alpha_i + \beta_i \cdot X_t$ such that $\forall i$, $\alpha_i \in \mathbb{R}$ and $\beta_i \in \mathbb{R}^n$. (Any affine map can be represented by a matrix multiplication followed by a vector addition.) This model unifies and strictly extends previous affine models to the maximum possible degree. Subject to some technical regularity conditions, the coefficients in Eq. (21) can also be made to be time dependent.

These models, although very general, and probably suitable for the vast majority of models required in financial engineering, do not cover all possible Ito processes (and, if the matrix a is non-diagonalisable, are not themselves Ito processes). In the general case, Ito integrands, represented by σ in Eq. (20) are not required to be linear in X_t (see [11]). The pricer allows more general formulae than this to be used, closely matching the general specification Eq. (11).

This can be modelled in the GPE as follows:

1) Set the $X(t_0)$ values
2) Select additive steps
3) Set each 'a' function to be: $(NEW \times X[0] + NEW \times X[1] + \cdots + NEW \times X[n-1] + NEW) \times D$
4) and enter the values for the 'NEW' variables
5) Set each 'b' function (the diffusion term) to be either
6) $NEW \times \sqrt{\begin{matrix} NEW \times X[0] + NEW \times X[1] + \cdots \\ +NEW \times X[n-1] + NEW \end{matrix}} \times \sqrt{D}$ or zero (if not on the diagonal). Again, the values or functions for the 'NEW' variables should be entered.

In addition to these three cases above, we have also demonstrated that the GPE can be used to obtain exact simulations of Geometric Brownian Motion, and for simulations using forward price data [27].

D. Including the Discount Factor in Our GPE

To estimate the current price of a derivative, we need to discount the estimated payoff for some future time, using Eq. (14). The discount factor used for this can be calculated from the interest rate process used in the derivative model. If the interest rate process is $r(t)$, then the discount factor is:

$$e^{-\int_0^t r(u)du}.$$

We can estimate the above integral by using an Euler scheme:

$$\Sigma_{i=0}^{n-1} r\left(t_i\right) \Delta t_i$$

where $t_0 = 0, t_n = T$ (the final time), and $\Delta t_i = t_{i+1} - t_i$. The payoff of the derivative is multiplied by the discount factor to obtain the current value of the derivative.

The user will be asked to specify whether they wish to simulate the discount factor as part of the calculation. When specifying the 'a' functions, any 'NEW' variables will be labelled as 'A_i_j', with 'i' being the row in the vector 'a' (of functions), and 'j' being the position of the function in that row ('j' is zero for the first 'NEW' variable, one for the second, and so on). After specifying the formula for each 'a' function, if that formula contains any 'NEW' variables, then each will be displayed by its label, and the user will be asked to enter the value or function (which can depend on time, the current state vector and/or have a random element) for that variable. If the user has said they wish to calculate the discount factor, then once they have specified all the 'NEW' variables in an 'a' function, these 'NEW' variables will be listed by their new label and specified value/function, and the user will then be asked to enter the label of the 'NEW' variable they wish to use as the interest rate.

V. Implementation of a Core Pricing Engine

Before considering the acceleration of pricing simulations (in Section VI), in this section we outline just the method for simulating the simpler Basket Option Pricing Engine (BOPE). This will later be extended to give the parallel GPE.

At the heart of the Monte Carlo method is the generation of a large number of paths, which are then reduced to a single value (the price estimate) by taking the average. For simulating Brownian Motion, this requires a large set of normalised random numbers, and several reduction operations.

Thus the pricing core implements the expression:

$$final\,Price = \quad F3(all\ a,\ F2(all\ p,\ F1\,(all\ s,\ calculate\_price\,[p,\ a,\ s])))$$

where *a* is an asset, *p* is the path index, *s* is a step, and F1, F2 and F3 are reduction operators (such as Sum, Average, Max).

In the pseudocode in Fig. 1, the reduction operation $F1(all\ s,\ price[p,\ a,\ s])$ means 'reduce the set of values $price[p,\ a,\ all\ values\ of\ s]$, for a fixed p and a, to a single value'. Similarly for the reduction operations $F2(all\ p,\ pathResult[p,\ a])$ and $F3(all\ a,\ assetResult[a])$.

For example, to obtain any of the European, Asian or BOPEs, we set the parameters and reduction operators as shown in Table I.

```
Generate set of normalised RandomNumber[p, a, s];
For every path p
    For every asset a
        /* Calculate series of prices for path p, one for each step */
        For every step s
            Calculate price[p, a, s] using RandomNumber[p, a, s];
        pathResult [p, a] = F1 (all s, price[p, a, s]);
For every asset a
    assetResult[a] = F2 (all p, pathResult[p, a]);
finalPrice = F3 (all a, assetResult[a]);
```

Fig. 1. Pseudo-code for the sequential BOPE.

TABLE I
Settings to Obtain a Range of Standard Derivatives

Option	Assets	Steps	F1	F2	F3
Euro	1	1	Final value	Average	Single member
Asian	1	Number of steps	Average	Average	Single member
Basket	Number of assets	Number of steps	Final value or Average	Average	Weighted Average*

(*If the payoff of the basket depends on the best performing asset, then $F3$ is Maximum instead of the weighted average.)

```
For every batch b
  Generate new set of normalised RandomNumber[b, p, a, s];
  For every path p in 1 ... P  /* P = total_paths / total_batches */
    For every asset a
      /* Calculate series of prices, one for each step */
      For every step s
        Calculate price[b, p, a, s] using
                                    RandomNumber[b, p, a, s];
      pathResult[b, p, a] = F1 (all s, price[b, p, a, s]);
  For every asset a
    batchResult[b, a] = F2_part1 (all p, pathResult[b, p, a]);

For every asset a
  assetResult[a] = F2_part2 (all b, batchResult[b, a]);
finalPrice = F3 (all a, assetResult[a]);
```

Fig. 2. Simplified pseudo-code for the BOPE.

Memory constraints mean that it is not possible to store the complete arrays $RandomNumber[p, a, s]$ and $price[p, a, s]$ in Fig. 1. We therefore split the computation into appropriately sized batches and accumulate the results. Fig. 2 gives a simplified description of the batched version, where the 3D arrays labelled by path, asset and step, become, for conceptual purposes, 4D arrays labelled by batch, path, asset, and step. The path dimension is broken into smaller batches. The results of each batch calculation are combined (reduced) on-the-fly, and a final calculation over the asset results is performed to obtain the final price estimate. Performing a reduction on a set of batches relies on the property:

$$F^*(sequence) = F'(all\ subsequences, F(subsequences))$$

where for $F^* = Max$ or $\mathbf{\Sigma}(sum)$, $F' = F = Max$ or Σ; and for $F^* = average$, $F = \Sigma$ and $F' = \Sigma$ followed by a final division.

VI. Parallel Implementation of GPE

In this section we firstly review existing work on acceleration of financial simulations using GPUs, multicores and FPGAs. Then we present our parallel implementation environment, and finally discuss the parallel implementation of the GPE, extending the simple BOPE simulation method above.

A. Previous Work on Acceleration of Pricing Engines

Some work has been done in the acceleration of American options pricing on GPUs using Monte Carlo simulations, with Abbas-Turki and Lapeyre [12] reporting speedups of 4.8 $\times$ to 8.7 $\times$ on one GPU over a sequential CPU implementation for single asset American option, with GPU speedup increasing with the number of steps in the simulation paths. Joshi [13], on the pricing of Asian options, achieves a GPU speedup of 150$\times$ over a CPU using quasi-Monte Carlo simulation (which uses low discrepancy sequences instead of random numbers to increase speedup, but is dimension dependent). Tree methods for pricing options have also been implemented on GPUs; for example Solomon *et al.* [14] implement a trinomial option pricer for single asset American lookback options. They report speedups of up to 100 $\times$ for large numbers of time steps; however their method will not work for more than one underlying asset. Several authors have implemented finite difference methods on GPUs; for example Egloff [15] implements a finite difference solver for options on one underlying asset (using CUDA). This system allows multiple single asset options to be priced, one on each multiprocessor on the GPU, and it is necessary to have many individual options in order to maintain high GPU occupancy. An average speed-up of around 24$\times$ is achieved over a sequential CPU implementation. It is suggested that finite difference methods for more underlying assets would achieve even greater speedups over the equivalent CPU implementation because more independent matrix equations would need to be solved. In a later paper [16] the author goes on to implement a two asset option pricer, which has enough necessary computation to occupy the GPU without needing to price more than one. Speedups of 70$\times$ are achieved over a single core CPU, and 30$\times$ over a multithreaded 4 core CPU. The author concludes that, for single asset pricing problems, 300 or more options are needed to sufficiently occupy the GPU, but for two asset problems, one option is sufficient. This illustrates the extremely rapid growth in size of finite difference computations with increasing numbers of underlying assets. Monte Carlo methods for option pricing have been implemented on GPUs, mostly for European type options; however, some studies in parallel Monte Carlo simulation on GPUs have shown that good results can be achieved even for American options, with NVidia researchers using a least squares method to estimate the optimum exercise time [17].

Option pricing has been accelerated using an FPGA. De Shryver *et al.* [18] found a Xilinx Virtex-5 FPGA accelerated system to be slower, but 2.5 times more energy efficient, than a Tesla C2050 GPU accelerated system for a one dimensional Heston (stochastic volatility) option pricing model. The authors predict that 3 such FPGAs would give similar speeds to the GPU while consuming only 3% of the energy, if the full calculation were performed on the FPGAs. Woods and VanCourt [19] used an FPGA to simulate Brownian Motion using Quasi-Monte Carlo methods, giving a 50 times speedup over a single thread CPU version, even though recursive algorithms and double precision are used (which, as the authors note, are "not normally associated with successful FPGA computing"). FPGAs have also been used for Credit Derivatives pricing [20], which involves simulating many scenarios (Monte Carlo simulation) and calculating the loss in each for a set of assets (debt obligations); then the average of these losses gives the overall expected loss. Other work employing FPGAs for option pricing include Tse *et al.* [21], [22], where the FPGA implementation is compared with a GPU implementation and is found to be faster (more than 2 times) and more energy efficient (more than 10 times). However the difficulty in programming FPGAs, and the skills and time required to do so, means that adoption of this technology by financial companies is difficult. FPGA pricing programs will usually have to be fixed, in order to be easy to use without expert knowledge, limiting their usefulness. Additionally, FPGAs do not use standard floating point, and usually use fixed precision arithmetic, which could make accuracy a concern, as well as giving less impressive speedups than GPUs for floating point intensive programs. Some progress has been made towards simplifying FPGA programming for high productivity [23], and, conveniently, OpenCL can also be used on some FPGAs [24], meaning that, in cases where they are most suitable, it is possible to port existing code to FPGAs.

In a study of applying the newer Intel MIC architecture to the problem of pricing American options, it was found that the MIC chip gives a speedup of 28 times over a single CPU core, while the 32 core server gives a speedup of 21 [25].

B. Parallel Implementation Environment

We selected two parallel acceleration architectures: GPUs and multicores. For portability across these platforms, we coded the GPE in OpenCL. For benchmarking purposes, we also coded the specific BOPE in CUDA (for a GPU) and in C (for multicore). OpenCL was chosen because of the availability of compilers for NVidia GPUs, Intel multicores and, in the longer term, for FPGAs. The Intel OpenCL compiler not only distributes the computation over the cores, but also automatically exploits the vector processing capability of the Intel architecture. One negative aspect of using OpenCL is the lack of optimised libraries for operations such as parallel random number generation (RNG). NVidia's libraries are available only in CUDA and not in OpenCL. This required us to develop our own parallel algorithm for RNG in OpenCL.

A key advantage of using OpenCL is the fact that the OpenCL program is compiled at runtime. The OpenCL program is held as a string (the program source string). Because this string can be generated by the program at runtime before compilation, this facilitates the automatic generation of code to implement the user-supplied formulae in the user's derivative specification. The user can specify any function for which there is an OpenCL function.

C. Parallel Implementation of the GPE

Implementation of the pricing engine has three distinct parts: RNG, correlating the random numbers, and the pricing core. The first two parts are fully parallelisable, as they require no communication between individual threads, and the threads will be fully utilised. In the third part, the reduction operators generally mean that not all threads will be active during the reduction computation.

The first part, RNG, can be parallelised in several ways. Each thread could be given its own copy of the random number generator, and be seeded with a simple random generator. Another option is to give each group of threads, that have access to a shared memory, a copy of the generator, or to divide this memory up and have multiple generators sharing one of these blocks of memory.

Manssen *et al.* [26], using CUDA, survey a number of RNG types, and the authors implement a version of a XorShift generator, and to use the skip-ahead method to avoid sequence overlap. The skip-ahead method in their paper involves multiplying the original state vector by a bit-shifting matrix, with the number of multiplications by this matrix equal to the block number of the thread block generating that part of the sequence. To speed up processing, the matrix is pre-computed. The need for this matrix multiplication is a drawback of using this method of parallelisation. Therefore, in order to make our parallel RNG portable, and to avoid the necessity of costly matrix computations, our RNG is implemented by giving each thread its own copy of a sequential generator, and seeding the threads using a simple random number generator.

The second part of the program, the correlation of the random numbers to produce samples from the required multivariate Normal distribution, is parallelised by having each thread work on its own section of the random number array in a coalesced manner. In order to obtain the correlated samples, batches of random numbers, each batch equal in size to the number of underlying assets, are taken to be the elements of a random vector. Each random vector needs to be multiplied by a correlation matrix. The correlation matrix is the same for all random vectors, and so is pre-computed on the host and transferred to the device. This matrix will be accessed many times, and by all threads, so a copy is stored in the shared memory of each multiprocessor.

The third part of the program, the pricing core, uses the previously generated and correlated random numbers. It carries out the actual price path generation, using a pricing formula and the initial input values for asset prices, interest rates, volatilities, time to expiry, etc. Monte Carlo-based simulation is embarrassingly parallel, so we allocate one thread to each path in the current batch. This corresponds to parallelizing for the loop in Fig. 2:

$$\text{for every path p in} 1 \ldots P \ldots .$$

The paths are independent, so the pricing core is easily parallelized in this way. When all the threads for a batch have produced their result, these results need to be reduced. We use a standard binary tree reduction approach, which ensures some parallelism, and also enhances the accuracy of the floating point reduction.

```
For each path p in parallel    /* One thread per path */
  For every step s   /* Note reordering of step & asset loops */
    For every asset a   /* Extra loop over the assets */
      Update the state vector using the user specified functions;
    /* Calculate prices, one for each asset, possibly using all or part
       of the last state vector for each asset price calculation */
    For every asset a
      Calculate price[p, a, s];
    For every asset a
      pathResult[p, a] = F1 (all s, price[p, a, s]);
    blockResults[p_B, a] =
              F2_part1(all pathsPerBlock, pathResult[p, a]);

For every asset a
  assetResult[a] = F2_part2 (all blocks, blockResult [p_B, a]);
finalPrice = F3 (all a, assetResult[a]);
```

Fig. 3. Simplified pseudocode for the parallel GPE.

Our implementation of the GPE can be run on either a GPU or a multicore processor, with only the number of threads and blocks needing to change to migrate between architectures.

In order to implement the GPE, it was necessary to change the order of the loops encountered by each thread. Previously it was possible to travel along the complete path (iterating through all the steps) for each asset and then move on to the next asset until all assets had been valued. In the generalised version the full state information from the current (vector) point, i.e. the state vector, may be used in the calculation of the next point, depending on the formulae entered by the user for the 'a' and 'b' components. In fact, because each variable may be used in the calculation of the next point, it is necessary to add an extra loop (within the 'steps' loop) over the underlying assets in order to update the state vector.

Instead of having loops in the OpenCL code to iterate over the assets to calculate the values for the 'a' component functions and 'b' component functions, a kind of loop unrolling is used. Fig. 3 gives the pseudocode for the parallel GPE. For clarity, it does not show the outer loop iteration over batches.

Within the hardware constraints and library constraints, etc., any instance of the general, Euler discretised formula, Eq. (12), can be generated with the use of prototype functions for the 'a' and 'b' components. The user specifies one function for the 'a' component for each asset, and one for the 'b' component for each asset. The formulae for the component functions can be time dependent, stochastic, or dependent on the current state vector (in keeping with the general formula). The prototype functions are modified accordingly and inserted into the OpenCL prototype state vector calculation string, which, in turn, will be inserted into the program source string. Function calls for the newly created functions are created from prototype function call strings. These are inserted into the main price path calculation string, which is in turn inserted into the program source string. The completed source string is then passed, as usual, to the OpenCL runtime.

The constraints are as follows:

1) The number of simulations must be a power of two.
2) The allowable mathematical operators within each function are those available in the C/OpenCL libraries.

TABLE II
TIMING RESULTS FOR THE BOPE ON 1-CORE (C), 4-CORE (OPENCL), GPU (CUDA) AND GPU (OPENCL) FOR VARYING NUMBERS OF ASSETS AND TOTAL PATHS

Num. Assets	Total Paths	1-core C(s)	4-core OpenCL (s)	GPU CUDA (s)	GPU OpenCL (s)	Speedup (GPU versus C)
8	2^{20}	5.3	0.5	0.04	0.04	133
8	2^{25}	171	14.6	1.2	1.2	143
8	2^{30}	5,462	470	37	38	144
16	2^{20}	11	1	0.1	0.1	110
16	2^{25}	351	32	3.2	3.2	110
16	2^{30}	11,214	1055	103	102	110
32	2^{20}	23	2.5	0.3	0.3	77
32	2^{25}	730	79	9.3	9.4	78
32	2^{30}	23,359	2519	298	303	77

TABLE III
TIMING RESULTS FOR THE OPENCL GPE AND BOPE ON GPU AND SELECTED 4-CORE SETTINGS

Num. Assets	Total Paths	GPU GPE(s)	GPU BOPE (s)	4-core GPE s)	4-core BOPE (s)
8	2^{20}	0.1	0.04		
8	2^{25}	2.7	1.2		
8	2^{30}	84	38	510	470
16	2^{20}	0.2	0.1		
16	2^{25}	6.4	3.2		
16	2^{30}	204	102	1080	1055
32	2^{20}	0.54	0.3		
32	2^{25}	16.3	9.4		
32	2^{30}	522	303	2610	2519

3) The allowable mathematical reduction operations across steps are: average, maximum, minimum.
4) The allowable binary reductions across simulations (paths) are: sum, maximum, minimum.
5) The allowable mathematical operations across assets are: sum, average, maximum, minimum.
6) The pricer currently only allows Normally distributed random values to be selected.
7) The intermediate values generated along each path are not stored. It may be desirable in some cases to store these values for future use, for example if the engine was used to simulate a complex model of the interest rate, the values of which were to be used for pricing simulations.

VII. TIMINGS AND PERFORMANCE

In the tests below, the GPU was an NVidia GTX670, with 2 GB RAM. Using the maximum allowable number of threads per block is optimal for this type of program. Therefore, the number of threads per block is fixed at 1024. For the multicore version, a 4-core Intel Xeon E31245, 3.3 GHz with 16 GB RAM running 64-bit Windows 7, was used. The number of threads per block is 8, and the number of blocks is 1024.

For comparison purposes, we used the GPE to create the standard BOPE, by supplying the appropriate functions and settings. We also separately coded the BOPE in C, CUDA and OpenCL, to give us benchmark performance for comparison.

Table II shows the execution time for the specific, hand-coded BOPE, for nine problem sizes: three different numbers of assets (8, 16 and 32), and three different numbers of paths for improving the accuracy of the Monte Carlo simulation (2^{20}, 2^{25}, 2^{30}—from one million to one billion). There are four implementations: a 1-core, sequential implementation in C; a 4-core implementation in OpenCL; and two GPU implementations: one in CUDA and one in OpenCL. The final column shows the speedup obtained for BOPE by the GPU relative to the sequential implementation (the two GPU implementations are practically identical in speed).

Table II shows that, for the maximum problem size, the GPU gives a speedup of around 77 (about 5 minutes as opposed to six and a half hours). A single chip with four cores gives a speedup of between 11 and 9. The latter shows the effectiveness of the Intel compiler in exploiting the vectorisation capabilities of the Intel core (which could give a theoretical speedup of $4\times$ per core). A further conclusion we can draw is that, for the hand-coded BOPE, there is little difference between the performance of CUDA and OpenCL.

For our second test, Table III gives results for the GPE-generated version of BOPE, for the same range of problem sizes, and compares with the BOPE timings from Table II.

From Table III one slightly surprising observation is that, while going to the GPE (with user-input functions at runtime) from BOPE on a multicore incurs only a small extra cost, on the GPU the speed is reduced by a factor of up to 1.9. This has to do with the way functions are handled by the respective compilers, and the inability of the NVidia OpenCL compiler to handle functions as first class objects efficiently. This was not an issue with BOPE, where every function was hard coded. Nevertheless, on the largest problem size, the GPE on the GPU still gives a speedup of 45, while a 4-core processor achieves a speedup of approximately 9.

VIII. CONCLUSION

In this paper we present an environment to support the rapid design of, and experimentation with, new financial derivatives. Two complementary approaches to supporting this task are presented. Rather than making available a wide range of standard pricing models, we have developed a much more general pricing engine—a GPE, which accepts mathematical formulae from the user. The paper shows that the GPE is expressively powerful enough to specify a wide range of pricing models, including many of the standard models. Having specified a new financial product, the environment exploits the power of parallel processing to enable the user to experiment with the new product more rapidly by accelerating the simulation. The GPE implementation is coded in OpenCL, and is completely portable across multicore and GPU architectures (apart from the architecture-dependent parameters defining the number of threads and blocks). Experiments on the largest problem size show that the GPE, when used to model a specific BOPE, can be accelerated by a factor of $45\times$ on an NVidia GTX670 GPU compared with a single core, sequential coding in C. On a 4-core processor, exactly

the same code runs about 9× faster than the single core implementation. The GPU implementation of the GPE suffers from a slow-down of about 1.9 because of a limitation of the NVidia OpenCL compiler when handling functions as objects.

REFERENCES

[1] S. M. Bartram, G. W. Brown, and F. R. Fehle, "International evidence on financial derivatives usage," *Financial Manage.*, vol. 38, no. 1, pp. 185–206, 2009.

[2] A. Devenow and I. Welch, "Rational herding in financial economics," *Eur. Econ. Rev.*, vol. 40, pp. 603–615, 1996.

[3] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *J. Political Economy*, vol. 81, pp. 637–654, 1973.

[4] L. H. Ederington and W. Guan, "Why are those options smiling?" *J. Derivatives*, vol. 10, no. 2, pp. 9–34, 2002.

[5] O. A. Vasicek, "An equilibrium characterization of the term structure," *J. Financial Econ.*, vol. 5, pp. 177–188, 1977.

[6] J. C. Cox, J. E. Ingersoll, and S. A. Ross, "A theory of the term structure of interest rates," *Econometrica*, vol. 53, pp. 129–151, 1985.

[7] J. C. Hull, *Options, Futures and Other Derivatives*, 8th ed. Boston, MA, USA: Prentice-Hall, 2012.

[8] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. New York, NY, USA: Springer-Verlag, 2004, p. 548.

[9] S. I. Heston, "A closed-form solution for options with stochastic volatility with applications to bond and currency options," *Rev. Financial Stud.*, vol. 6, pp. 327–343, 1993.

[10] D. Duffie and R. Kan, "A yield-factor model of interest rates," *Math. Finance*, vol. 6, no. 4, pp. 379–406, 1996.

[11] B. Oksendal, "The Ito formula and the martingale representation theorem," in *Stochastic Differential Equations*. New York, NY, USA: Springer-Verlag, 2005, pp. 43–49.

[12] L. A. Abbas-Turki and B. Lapeyre, "American options pricing on multicore graphic cards," in *Proc. Int. Conf. Bus. Intell. Financial Eng.*, Beijing, China, 2009, pp. 307–311.

[13] M. S. Joshi, "Graphical Asian options," *Wilmott J.*, vol. 2, no. 2, pp. 97–107, 2010.

[14] S. Solomon, R. K. Thulasiram, and P. Thulasiraman, "Option pricing on the GPU," in *Proc. 12th IEEE Int. Conf. High Perform. Comput. Commun.*, Melbourne, Vic., Australia, 2010, pp. 289–296.

[15] D. Egloff, "High performance finite difference PDE solvers on GPUs," QuantAlea Gmbh, Tech. Rep., Zurich, Switzerland, (2010, Feb.). [Online]. Available: http://download.quantalea.net/ fdm-gpu.pdf

[16] D. Egloff, "Pricing financial derivatives with high performance finite difference solvers on GPUs," in *GPU Computing Gems Jade Edition*, W. Hwu, Ed. Burlington, MA, USA: Morgan Kaufmann, 2011.

[17] M. Fatica and E. Phillips, "Pricing American options with least squares Monte Carlo on GPUs," *Proc. 6th Workshop High Perform. Comput. Finance*, Denver, 2013.

[18] C. de Schryver *et al.*, "An energy efficient FPGA accelerator for Monte Carlo option pricing with the Heston model," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Cancun, Mexico, 2011, pp. 468–474.

[19] N. A. Woods and T. VanCourt, "FPGA acceleration of quasi-Monte Carlo in finance," in *Proc. Int. Conf. Field Program. Logic Appl.*, Heidelberg, Germany, 2008, pp. 335–340.

[20] A. Kaganov, P. Chow, and A. Lakhany, "FPGA acceleration of Monte-Carlo based credit derivative pricing," in *Proc. Int. Conf. Field Program. Logic Appl.*, Heidelberg, Germany, 2008, pp. 329–334.

[21] A. H. T. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk, "Reconfigurable control variate Monte-Carlo designs for pricing exotic options," in *Proc. Int. Conf. Field Program. Logic Appl.*, Milano, Italy, 2010, pp. 364–367.

[22] A. H. T. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk, "Efficient reconfigurable design for pricing Asian options," *ACM SIGARCH Computer Archit. News*, vol. 38, no. 4, pp. 14–20, 2011.

[23] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Proc. Int. Conf. Field-Program. Technol.*, Beijing, China, 2010, pp. 94–101.

[24] D. Manners, "Altera goes for the data centre," *Electron. Weekly*, Oct. 8, 2014. [Online]. Available: http://www.electronicsweekly.com/news/business/altera-goes-data-centre-2014-10/. [Accessed on: Oct. 14, 2014].

[25] Y. Hu, Q. Li, Z. Cao, and J. Wang, "Parallel simulation of high-dimensional American option pricing based on CPU versus MIC," *Concurrency Comput., Practice Experience*, vol. 27, pp. 1110–1121, 2014.

[26] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on GPU," *Eur. Phys. J. Special Topics*, vol. 210, pp. 53–72, 2012.

[27] S. Trainor, "Towards a portable accelerated asset path simulator for derivatives pricing," Ph.D. dissertation, Queen's University Belfast, Belfast, U.K., 2015.

Danny Crookes became a Professor of computer engineering in 1993 at Queens University Belfast (QUB), Belfast, U.K., and was the Head of Computer Science from 1993 to 2002. He is the Director of Research for Speech, Image and Vision Systems at the Institute for Electronics, Communications and Information Technology, QUB. His current research interests include the use of novel architectures (especially GPUs) for high-performance image and speech processing. He is currently involved in projects in acceleration of financial simulations, speech enhancement, face recognition, and image processing for cancer diagnosis. He has published more than 230 scientific papers in journals and international conferences, and has presented tutorials on parallel image processing at several international conferences.

Sean Trainor received the M.Sc. degree in pure mathematics from Queen's University Belfast (QUB), Belfast, U.K., in 2008, and the Ph.D. degree in financial simulation from the Institute for Electronics, Communications and Information Technology, QUB, in 2015. He has worked in developing financial products in several financial markets companies in the City of London.

Richard Jiang received the Ph.D. degree in computer science from Queen's University Belfast, Belfast, U.K., in 2008.

After the Ph.D. study, he was with Brunel University, Loughborough University, Swansea University, the University of Bath, and the University of Sheffield. He joined Northumbria University, Newcastle upon Tyne, U.K., in May 2013, where he is currently a Lecturer in the Department of Computer Science and Digital Technologies. His research interests mainly reside in the fields of artificial intelligence, man–machine interaction, hardware acceleration, visual forensics, and biomedical image analysis. His research has been funded by EPSRC, BBSRC, TSB, EU FP, and industry, and he has authored some 50 publications.