High Performance Monte-Carlo Based Option Pricing on FPGAs

Xiang Tian, Khaled Benkrid, and Xiaochen Gu
The University of Edinburgh, School of Electronics and Engineering,
Mayfield Road, Edinburgh EH9 3JL, Scotland, UK
(x.tian,k.benkrid,x.gu)@ed.ac.uk

***Abstract*—High performance computing is becoming increasingly important in the field of financial computing, as the complexity of financial models continues to increase. Many of these financial models do not have a practical close form solution in which case numerical methods are the only alternative. Monte-Carlo simulation is one of most commonly used numerical methods, in scientific computing in general, with huge computation benefits in solving problems where close form solutions are impossible to derive. As the Monte-Carlo method relies on the average result of thousands of independent stochastic paths, massive parallelism can be adopted to accelerate the computation. Computer clusters with off-the-shelf accelerator hardware are increasingly being proposed as an economic high performance implementation platform for many scientific computing applications. This paper is part of this trend as it presents an implementation of a Monte-Carlo simulation engine for option pricing on an FPGA-based supercomputer, called Maxwell, developed at the University of Edinburgh. The latter consists of a 32 CPU cluster augmented with 64 Virtex-4 Xilinx FPGAs connected in a 2D torus. Our engine can implement various Monte-Carlo simulations on the Maxwell machine with speed-ups in excess of 100x compared to equivalent software implementations. This is illustrated in this paper in the context of an implementation of the GARCH option pricing model. Real hardware implementation shows that our FPGA-based implementation of the GARCH model outperforms an equivalent software implementation running on a workstation cluster with the same number of computing nodes (CPU/FPGA) by a factor of 340.**

***Index Terms*—Financial computing, High performance computing, Monte-Carlo simulation, FPGA**

I. Introduction

High performance computing is of great importance in the field of finance when it comes to solving problems which are defined on models of financial variables. Many of these problems, however, cannot be solved with an analytical solution because of the large number of coupled degrees of freedom in these problems. In these instances, a numerical computational technique called the Monte-Carlo method is often used. The latter relies on repeated random sampling of model equations in order to compute their solutions. It is a technique that is widely used in physical chemistry, computational physics, and related applied fields. Monte-Carlo simulations are also used to forecast a wide range of events and scenarios, such as the weather, product sales and consumer demand. For instance, Figure 1 depicts forecasts for electricity demand in a geographical area based on historical data, using Monte-Carlo simulation.

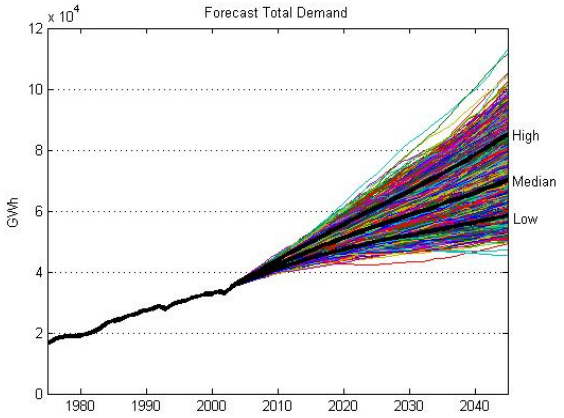

Figure 1. Future demand for electricity in a city (http://www.electricitycommission.govt.nz/opdev/modelling/demand/natforecast)

In financial computing, the Monte-Carlo technique is used to simulate the various sources of uncertainty that affect the value of the instrument, portfolio or investment in question. Many financial computing applications have no close form solutions, as they depend on three or more stochastic variables. Here, Monte-Carlo simulation tends to be numerically more efficient than other procedures [1] . This is because the computational time of Monte-Carlo simulations increases approximately linearly with the number of variables, whereas in most other methods, computational time increases exponentially with the number of variables. One of the important characteristics of Monte-Carlo simulation is parallelism as multiple independent paths are computed. Hence, massive parallelism can be adopted to accelerate the simulation. This paper presents an implementation of Generalized Autoregressive Conditional Heteroskedastic (GARCH) option pricing model, using Monte-Carlo simulation, on a FPGA supercomputer called Maxwell. Our FPGA-based Monte-Carlo simulation engine benefits from the maximum possible parallelism with the added advantages of reprogrammability and relative low power.

Compared to previous work, this paper presents the fastest FPGA implementation of a Monte-Carlo based simulation of option pricing, ever reported in the literature. It also presents a complete design and

implementation of a generic Monte-Carlo based simulation engine on an FPGA supercomputer.

The remainder of this paper is organized as follows. First, section 2 presents background information on options, the evolution of stock prices, the GARCH model, and Monte-Carlo based simulation of option pricing. Section 3 then reports previous hardware implementations of Monte-Carlo option pricing simulation. After that, our own hardware implementation is presented in detail, in section 4, including the implementation details of a random number generator using the Box-Muller method, a stochastic volatility computing module, as well the system design details of a complete Monte-Carlo based engine for option pricing. The implementation results on the Maxwell FPGA supercomputer are then presented in section 5. The latter will include information about the Maxwell machine architecture, message passing interface, and front-end software design. A comparison of our hardware implementation results with our own equivalent software implementations as well as other implementations reported in the literature is then presented in section 6. Finally, conclusions and plans for future work are drawn.

II. Background

In this section, we will present basic background material related of relevance to the rest of this paper, including concept of options, the behaviour of stock prices, the GARCH model of stock evolution, and the mathematical underpinnings of Monte-Carlo based simulation of option prices.

A. Concept of Options

Options are traded both on stock exchanges and on over-the-counter markets. There are two basic types of options [1] : a *call* option which gives the holder the right to buy the underlying asset by a certain date for a certain price, and a *put* option which gives the holder the right to sell the underlying asset by a certain date for a certain price. The price in the contract is known as the *exercise* price or *strike* price, and the date in the contract is known as the *expiration* date or *maturity*. Within these types, we also distinguish between two main types of options: *American* options which are call or put options that can be exercised at any time up to the expiration date, and *European* options which can be exercised only on the expiration date itself. There are also *Asian* options on the market which differ from European and American options in that their strike price is the average price of the asset over a period of time, computed by collecting the daily closing price over the life of the option. In this paper, we focus mainly on European options.

B. Evolution of Stock Prices

Since options are traded on the stock market, their value/price changes according to offer and demand. If S is the stock price at time t, the expected drift rate in S should be assumed to be μS for a constant parameter of μ. This means that in a short interval of time, given as δt, the expected increase in S is $\mu S \delta t$. The parameter μ is the expected rate of return on the stock, expressed in decimal form. If the volatility of the stock price is always zero, this model implies that:

Eq 1. $$\delta S = \mu S \delta t$$

In the limit as $\delta t \rightarrow 0$,

Eq 2. $$dS = \mu S dt$$

Or

Eq 3. $$\frac{dS}{S} = \mu dt$$

In practice, a stock price does exhibit volatility however. A reasonable assumption is that the variability of the percentage return in a short period of time (δt) is the same regardless of the stock price. This suggests that the standard deviation of the change, in interval δt, should be proportional to the stock price, which leads to the following model:

Eq 4. $$dS = \mu S dt + \sigma S dz$$

Or

Eq 5. $$\frac{dS}{S} = \mu dt + \sigma dz$$

where variable σ is the volatility of the stock price, and variable μ is its expected rate of return. The discrete-time version of the model is:

Eq 6. $$\delta S = \mu \delta t + \sigma \delta z$$

The variable δS is the change in the stock price S in a small time interval δt, and δz is a random number drawn from a standardized normal (Gaussian) distribution. The parameter μ is the expected rate of return per unit of time from the stock, and the parameter σ is the volatility of the stock price per unit of time. By importing Ito's lemma [1] to derive the process followed by G, defined as:

Eq 7. $$G = \ln S$$

we obtain:

Eq 8. $$dG = (\mu - \frac{\sigma^2}{2})dt + \sigma dz$$

Here, dz is a Wiener process which is related to dt by the following equation:

Eq 9. $$dz = \varepsilon \sqrt{dt}$$

ε is a random variable with a normal (Gaussian) distribution with a mean of zero and a standard deviation of 1.0. Rewriting Eq 8 with the relationship in Eq 9, we get:

Eq 10. $$dG = (\mu - \frac{\sigma^2}{2})dt + \sigma \varepsilon \sqrt{dt}$$

This model is also called the Black-Scholes option pricing model [2] .

C.Stochastic Volatility

One assumption in the Black-Scholes model that is not always true in practice is the assumption that volatility is constant. Indeed, practitioners often find it necessary to change the volatility parameter when using the Black-Scholes model to value options. In the case where the stock price and volatility are correlated, there is no simple solution to the model equations and Monte-Carlo based simulations often become necessary.

One technique for modeling volatility that has become popular is GARCH model [3] . The most commonly used GARCH model is GARCH (1, 1) where the volatility is given by the following equation:

Eq 11. $$\sigma_i^2 = \sigma_0 + \alpha\sigma_{i-1}^2 + \beta\sigma_{i-1}^2\lambda^2$$

Here α, and β are constants which can be estimated from historical data using maximum likelihood methods. σ_0 is the volatility of the stock price at time 0, σ_i and σ_{i-1} are the volatilities at time $i\Delta t$ and $(i-1)\Delta t$. λ is a random variable with a normal (Gaussian) distribution with a mean of zero and a standard deviation of 1.0. Notice that the random variable in the GARCH model is different from the one used in the describing the evolution of stock prices. The two random variables represent two independent stochastic processes.

For options that last less than a year, the pricing impact of a stochastic volatility is fairly small in absolute terms. It becomes progressively larger as the life of the option increases.

D.Monte-Carlo Simulation

Monte-Carlo simulation of a stochastic process like the one depicted in Eq 8 is a procedure for sampling random outcomes of that process, associating an event (e.g. a particular variability of the stock) with a set of outcomes (e.g. a particular strike price at time t) and defining the probability of the event to be its volume or measure relative to that of a universe of possible outcomes. Monte-Carlo simulation tends to be numerically more efficient than other procedures when there are many stochastic variables. Figure 2 gives the evolution of a stock's price in time, for 50 different random paths.

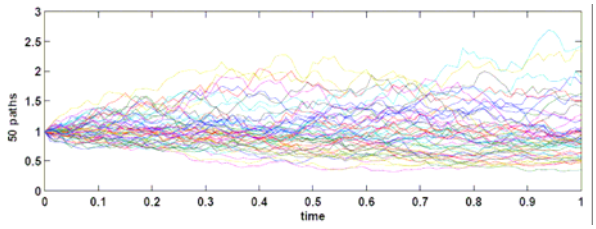

Figure 2. 50 paths of stock price evolution

III. High Performance Monte-Carlo based Financial Computing

Initially, financial computing did not benefit greatly from developments in high performance computing, as the latter aimed mainly at engineering and weapon design applications. Besides, financial experts were initially focusing on developing mathematical models and computer simulations in order to comprehend the behaviour of financial markets and develop risk-management tools. As this effort progressed, the complexity of financial computing applications grew up rapidly. Indeed, as the number of stochastic parameters involved in financial models increased, close form solutions became impractical to derive, and hence Monte-Carlo based sampling methods became an attractive alternative. As a result, parallel computing systems offered a distinctive advantage and were subsequently introduced into the realm of financial computing.

In [4] , a survey of high performance computing systems and computer-aided design tools for financial computing was presented. Figure 3 illustrates the processing time needed to derive the state dependent pricing of a portfolio of mortgage backed securities on a number of computing platforms. This shows that simulations that would take several hours on workstations could be completed within few minutes on a CRAY X-MP supercomputer and within less than a minute on a massively parallel system.

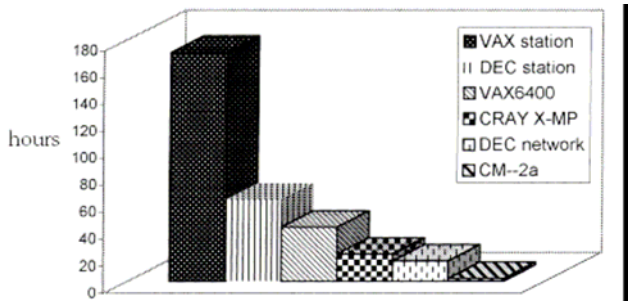

Figure 3. State dependent pricing of a portfolio of mortgage backed securities on a variety of workstations, supercomputer CRAY X-MP, a workstations cluster, and a massively parallel Connection Machine CM-2a (Source: [4])

The cost of cluster computers and supercomputers can, however, be prohibitive. Area and power consumption can also be a major disadvantage of these computing platforms. For these reasons, alternative platforms are being considered. Field Programmable Gate Arrays (FPGAs), for instance, offer the high performance of a dedicated hardware solution of a particular algorithm, with a fraction of the area and power consumption of equivalent microprocessor-based solutions. Moreover, the continuous developments in transistor integration levels mean that it is now possible to implement a considerable number of floating-point arithmetic units on modern FPGAs. If this trend is to continue, FPGA use is set to conquer new application domains, including financial computing. The following

presents a number of recent FPGA-based financial computing application implementations.

In [5] , an FPGA-based Monte-Carlo simulation core used for computing the BGM (Brace, Gatarek and Musiela) interest rate model for pricing derivatives was presented. The BGM interest rate model is commonly used to simulate the fluctuation of interest rates over time, something which has an influence on nearly all economic activity. Results show that around 25 times speed-up can be obtained by using an FPGA, compared to an equivalent Pentium IV 1.5GHz based software implementation. Other hardware architectures for Monte-Carlo based financial simulations were published in [6] . In this paper, five different Monte-Carlo option pricing simulation algorithms were explored, including log-normal price movements, correlated asset Value-at-Risk calculation, and price movements under the GARCH model. Using a Xilinx Virtex-4 XC4VSX55 device, implementation results show that FPGA implementations run on-average 80 times faster than equivalent software ones (running on a 2.66GHz PC).

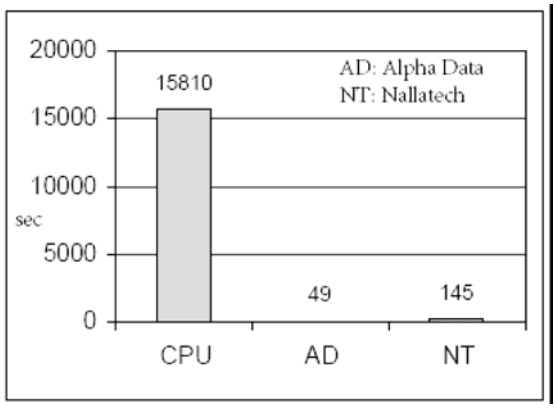

Figure 4. Single node performance of Asian option pricing simulation on the Maxwell machine [7]

The combination of cluster technology and reconfigurable hardware acceleration is a relatively new development in high performance computing, which promises to combine the relatively high performance and low power consumption of reconfigurable hardware with established design flows and consequent knowledge base in traditional microprocessor based high performance computing. Maxwell, a supercomputer with 64 FPGA nodes, is a relatively recent development in this direction [7] . A simple Asian option pricing core was designed as a demonstration application on Maxwell. As mentioned above, Asian options are a special type of options where the strike price is the average price of the asset over a period of time, computed by collecting the daily closing price over the life of the option. The implementation results of this demonstrator application are shown is Figure 4 (AlphaData and Nallatech are the two FPGA companies that donated the FPGA accelerator nodes on the Maxwell machine, 32 each). The results show that the AlphaData nodes lead to ~320-time speed-up compared to an equivalent software implementation, whereas the Nallatech nodes lead to a 109-time speed-up. The discrepancy is due to the design language/flow used for each node type: VHDL for AlphaData and a proprietary C-based hardware language, called DIME-C, for Nallatech.

IV. Our Hardware Architecture of a Monte-Carlo Simulation Engine

Monte-Carlo simulation relies on stochastic sampling, and as such random number generation is a key part of it. Software implementations of random number generators are relatively slow, that is why a hardware implementation is required. However, while software implementations are abundant, hardware ones are relatively few. In the option pricing equations presented in section II above, a Gaussian random number generator is required. In this paper, we use the Box-Muller method for the hardware generation of Gaussian random numbers [8] .

The hardware architecture of our Monte-Carlo simulation engine is shown in Figure 5. In it, N Monte-Carlo computing cores run in parallel to generate N different paths at the same time. Therefore, the total number of paths that each core has to compute is equal to the total number of paths required divided by N. In the end, results from each core e.g. average option price over N paths, are collected by the host (in software). The latter will perform high level operations on these results e.g. averaging the intermediate results to calculate the most probable strike price.

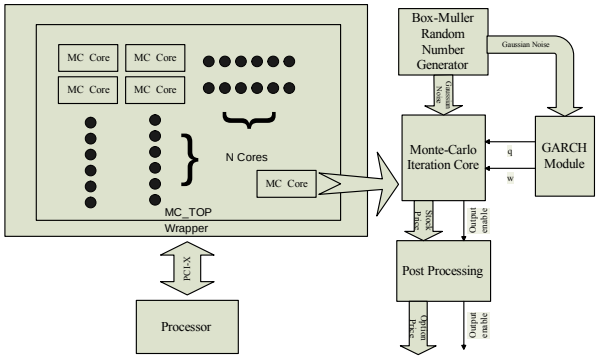

Figure 5. Generic architecture of a Monte-Carlo simulation engine

Each computing core comprises the following components: (a) one Box-Muller random number generator, (b) a simulation core that provides computational resources for iteration, (c) a stochastic volatility computing module based on the GARCH model, and (d) a post processing module e.g. for averaging intermediate option prices.

The following sub-sections describe the detailed design of each module presented in Figure 5. Before that, however, we note that we have decided to use fixed-point arithmetic to implement all of these units after performing a range analysis in MALAB to decide on the minimum required wordlength(s) to satisfy a

particular precision level (0.01% in our case). 26-bit precision was the maximum wordlength used to satisfy a desired precision level, except for the final result accumulator in each MC core (see section D below) which was 48-bit wide.

A. Box-Muller Random Number Generator

The Gaussian number generator is a critical component of the Monte-Carlo simulation engine. In order to speed up the simulation, we chose to build a hardware random number generator for each Monte-Carlo simulation core. Our random number generator is based on the Box-Muller method [9] , which is illustrated in Figure 6.

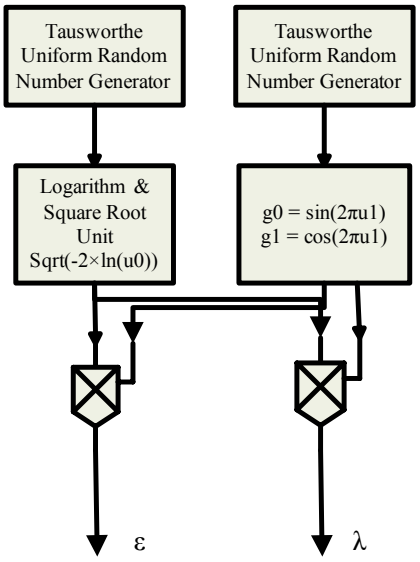

Figure 6. Box-Muller Gaussian noise generator architecture (Two independent samples)

The Box-Muller method is conceptually straight-forward. Given two independent realizations (u_1 and u_2) of a uniform random variable over the interval [0, 1), and a set of intermediate functions f, g_1 and g_2 so that:

Eq 12. $f(u_1) = \sqrt{-2 \times \ln(u_1)}$

Eq 13. $g_1(u_2) = \sin(2\pi u_2)$

Eq 14. $g_2(u_2) = \cos(2\pi u_2)$

Then x_1, x_2 below provide two samples of a Gaussian distribution N(0, 1):

Eq 15. $x_1 = f(u_1)g_1(u_2)$

Eq 16. $x_2 = f(u_1)g_2(u_2)$

As the two sets of random number are statistic independent, one is used for the evolution of stock price and the other is used for GARCH model.

The uniform random number generator used in this design is called Tausworthe URNG [10] , which is described by the pseudo-code shown in Figure 7.

```
unsigned int s0, s1, s2, b;

unsigned int taus()
{
   b = (((s0 << 13) ^ s0) >> 19);
   s0 = (((s0 & 0xFFFFFFFE) << 12) ^ b);
   b = (((s1 << 2) ^ s1) >> 25);
   s1 = (((s1 & 0xFFFFFFF8) << 4) ^ b);
   b = (((s2 << 3) ^ s2) >> 11);
   s2 = (((s2 & 0xFFFFFFF0) << 17) ^ b);
   return s0 ^ s1 ^ s2
}
```

Figure 7. Tausworthe URNG Algorithm

The logarithmic and trigonometric functions are computed using the piecewise linear approximate method presented in [11] . The logarithm errors of both functions are shown in Figure 8 and Figure 9.

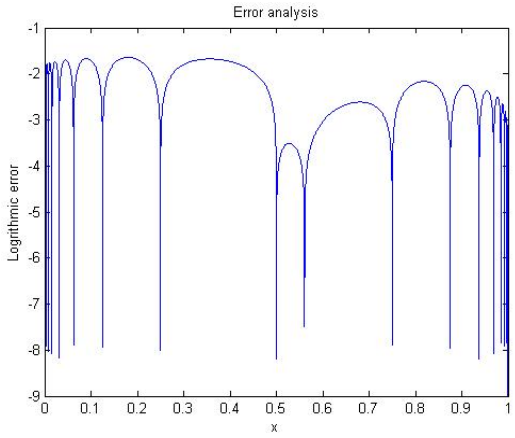

Figure 8. Logarithm error of $f(x) = \sqrt{-2 \times \ln(x)}$

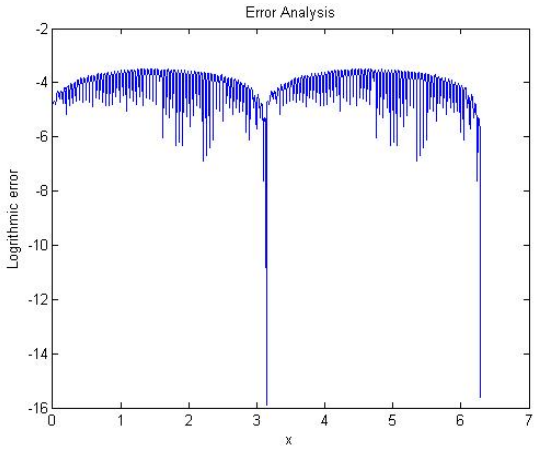

Figure 9. Logarithm error of $g_1(x) = \sin(2\pi x)$

We generated 100,000 samples, which gave the probability distribution function (PDF) shown in Figure 10.

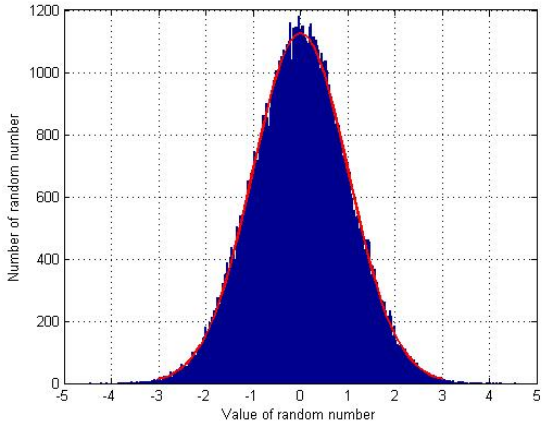

Figure 10. PDF of the generated noise

We used two goodness-of-fit tests to check the normality of the Gaussian noise: the chi-square (χ^2) test and the Kolmogorov-Smirnov (K-S) test [12] . These are used to compute p-values for the outputs. The general convention is to reject the null hypothesis – that the samples are normally distributed if the p-value is less than 0.05. The test results of our design samples generated p-values which were greater than 0.05, confirming the statistical normality of our samples.

B. Monte-Carlo Iterator

In this section, we present the Monte-Carlo Iterator that corresponds to the option pricing model outlined in section II above:

Eq 17. $$S_{\Delta t} = S_0(1 + ((\mu - \frac{\sigma^2}{2})\delta t + \sigma\varepsilon\sqrt{\delta t}))$$

The parameters S_0, μ, and σ are inputs of the core, and ε is a random variable drawn from a standardized normal distribution, which is generated by our Box-Muller random number core.

Assuming volatility is an input to the iterator, and hence is a given, the only variable parameter in the above equation is the random number, we can calculate the other constant expressions before inputting them to the computing core. Rewriting Eq 17 with this in mind gives:

Eq 18. $$S_{\Delta t} = S_0((1 + (\mu - \frac{\sigma^2}{2})\delta t) + \varepsilon\sigma\sqrt{\delta t})$$

Therefore, we can calculate the following coefficients in advance:

Eq 19. $$q = 1 + (\mu - \frac{\sigma^2}{2})\delta t$$

Eq 20. $$w = \sigma\sqrt{\delta t}$$

The architecture of the corresponding Monte-Carlo iteration core is presented Figure 11 (the black block represents a delay).

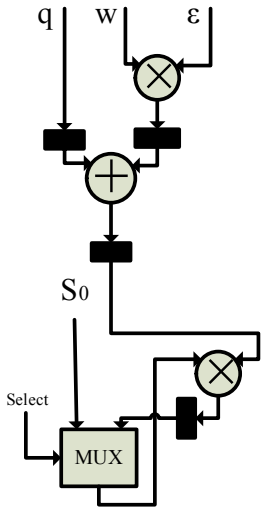

Figure 11. Architecture of one Monte-Carlo Iteration Core

C. Volatility Calculating Module

As volatility is a stochastic process in the GARCH option pricing model, the inputs (q, w) for Monte-Carlo iterator mentioned above are not constants. We use Eq 11 to get the everyday volatility before using Eq 17 to get the evolution of the stock price. Although there is a feedback loop in the equation used to get the volatility, we still want to finish one time of iteration in one cycle. Therefore, only one stage of pipeline can be used in the feedback cycle. In order to minimize the combinatorial logic in feedback cycle, however, we design the GARCH module architecture as shown in Figure 12.

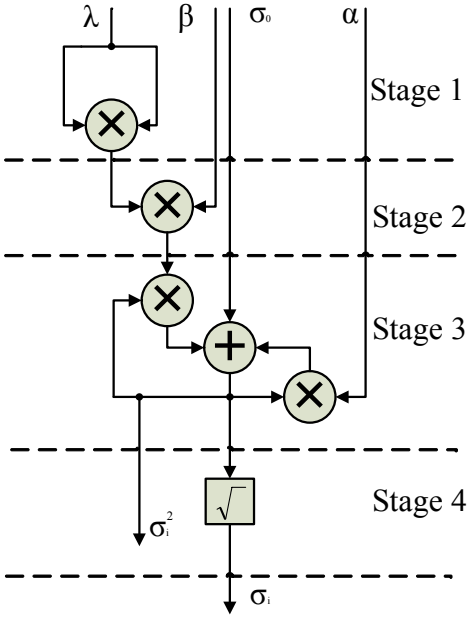

Figure 12. Architecture of the GARCH module

In Figure 12, the dashed line means one stage of pipelining. λ is a Gaussian random number provided by the Box-Muller random number generator. α, β, and σ_0 are inputs from the top module.

As the output of the GARCH model is the volatility of the stock, and the input for Monte-Carlo iterator is q and w, we calculate these two variables in the GARCH module. After rewriting Eq 19:

Eq 21. $$q = (1 + \mu\delta t) - \sigma^2 \frac{\delta t}{2}$$

Then, a pair of q and w can be obtained in each clock cycle using Eq 20 and Eq 21.

D.Post Processing

This stage is designed to gather the results generated by each path and calculate the required output. In the case study outlined in section II, this consists in accumulating the option prices generated from each path, and computing the average. Considering that the number of paths is very large, this would result in a large accumulator and a large divider to get the average result. However, if we ensure that the number of paths is a power of 2, we only need a shift register to get the average result. The corresponding architecture of the post-processing unit is shown in Figure 13.

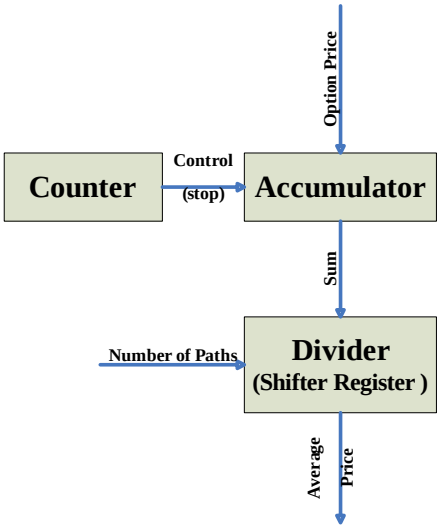

Figure 13. Architecture of the post processing unit

Obviously, using a shifter register instead of a divider saves considerable resources, which can then be used to generate more computing cores on a single FPGA, hence increasing the overall performance. Finally, note that although the number of simulation paths can be tuned more finely using a divider than it is using a shift register the final results are very similar when the number of simulation paths is of 6-orders of magnitude (within four decimals).

V. Real Hardware Implementation and Results

We implemented our Monte-Carlo simulation engine on an FPGA supercomputer, namely the Maxwell machine. As stated above, Maxwell is a high-performance computer developed by FPGA High Performance Computing Alliance (FHPCA) to demonstrate the feasibility of running computationally demanding applications on an array of FPGAs [7] [13] . It comprises 32 blades housed in an IBM Blade Center. Each blade comprises one 2.8 GHz Xeon with 1 Gbyte memory and an FPGA PCI-X card, with two FPGAs each. Half of Maxwell's accelerator cards are Nallatech's off-the-shelf H101-PCIXM cards with Xilinx V4100LX Virtex-4 devices [14] . The other half are AlphaData ADM-XRC-4FX cards which contain Xilinx XV4FX100 Virtex-4 devices. Each FPGA has either 512 Mbytes or 1 Gbyte of private memory.

Whilst the Xeon and FPGAs on a particular blade can communicate with each other over the PCI bus (typical transfer bandwidth in excess of 600 Mbytes/s), the principal communication infrastructure comprises a fast Ethernet network with a high-performance switch linking the Xeons together and RocketIO linking the FPGAs. Each FPGA has 4 RocketIO links enabling the 64 FPGAs to be connected together in an 8 × 8 toroidal mesh (see Figure 14). The RocketIO has a bandwidth in excess of 2.5 Gbits/s per link.

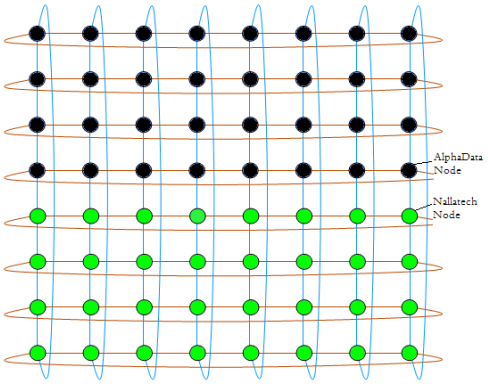

Figure 14. Communication Networks on Maxwell [13]

The following presents the implementation results for a GARCH option pricing model configuration of our Monte-Carlo simulation engine. We designed our hardware components using Verilog-HDL and synthesized them using Xilinx ISE 9.2i. We could fit 11 Monte-Carlo cores (see Figure 5) on one single FPGA chip. These occupied 39,466 slices on an XV4FX100-ff1517 FPGA [14] , which has 42,176 slices overall. Besides, all 160 DSP48s units were utilized. The peak clock frequency of the core is 53MHz. We set the clock frequency on the Maxwell's FPGA nodes to 50MHz.

In our particular implementation instance, we used the AlphaData nodes on the Maxwell machine. We used the ADM-XRC-4FX Co-Processor Development Kit (CPDK) provided by AlphaData to interface our user application core with I/O communication hardware on the FPGA, and generate configuration bitstream. The structure of the CPDK is illustrated in Figure 15. In it, a design is divided into hardware and software parts. A C++ program is used to configure the FPGA, initialize design parameters, and communicate with the user application hardware. Furthermore, as we have targeted distributed blades on the Maxwell machine, we needed to import a communication tool to connect the nodes. We used the Message Passing Interface (MPI) [15] for

this purpose. The Sun Grid Engine job scheduler (SGE) was used to submit jobs to the Maxwell machine front-end.

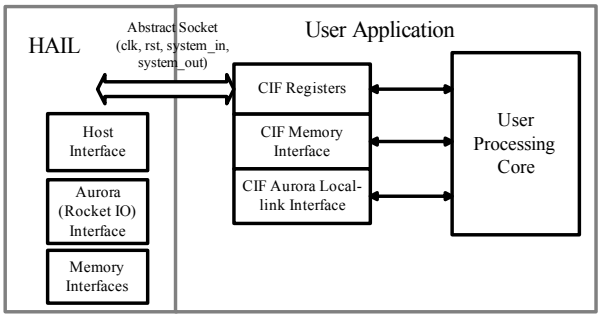

Figure 15. Structure of CPDK API [13]

Figure 16 gives the execution time of the GARCH option pricing model on the Maxwell machine using an increasing number of nodes. This is shown for our FPGA implementation as well as for an equivalent software implementation running on the 2.8 GHz Xeon processors (FPGA vs. Software). In both cases, the execution time reduces linearly as the number of nodes increases. This is because inter-communication time is negligible compared to computing time. Indeed, the only instances where communication between the host software and the Monte-Carlo cores (running on FPGA or on the Xeon processors) is needed is when parameters are broadcasted to the cores at the beginning of the execution, and when results are gathered from the cores at the end of the simulation. Compared to software, our FPGA implementation results in a 340 times speed–up. It is worth mentioning that this speed-up figure is independent of the number of nodes (FPGA/CPU) used.

The FPGA implementation was clocked at 50MHz only, compared to the Xeon's 2.8GHz clock frequency. The reason behind the high speed-up figure of the FPGA implementation, despite the huge difference in clock frequency, is due to the high level of process parallelism (11 cores running in parallel on each FPGA device) as well as the high degree of pipelining used within each core.

In the case of the FPGA implementation, our Monte-Carlo computing core finished one time-iteration per 20 ns. As the inter-communication time is negligible, we approximate the total computing time by the following formula:

$$\text{Eq 22.} \quad ComputingTime = ClockPeriod \times (NumOfPaths \times NumOfDays) \div (NumOfCores \times NumOfNodes)$$

For instance, in the case of the 32 nodes experiment, the clock period is 20 ns, the number of paths is $2^{16}\times 11 = 7.21\times 10^5$, the number of days is 100 (simulation time), the number of cores per FPGA is 11 and the number of nodes is 32. This gives us:

$$\text{Eq 23.} \quad ComputingTime = 20 \times (7.21\times 10^5 \times 100) \div (11 \times 32) \approx 4.097 \times 10^6\ ns = 4.1\ ms$$

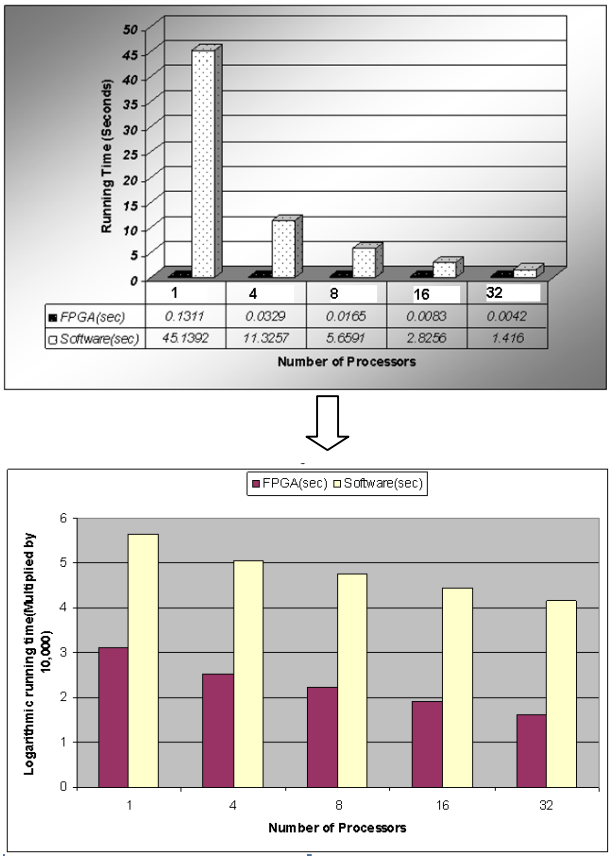

Figure 16. Execution time of the GARCH option pricing model (FPGAs vs. Xeon Processors)

Performing meaningful comparisons with previous work is very difficult because of differences in algorithms used, design parameters, platform used, and experimental set-up. Nonetheless, the following will attempt to make a meaningful comparison with closely related work. The work presented in [6] showed implementation results of the GARCH model. The results show that a Virtex-4 XC4VSX55 based implementation outperforms an equivalent 2.66GHz Pentium IV software implementation by a factor of 49, which is well below our speed-up figure. This paper also gives a result for log-normal walk model, which is equivalent to Black-Scholes option pricing model. A factor of 85 times speed-up can be obtained. To compare with this, we also implemented our design with only the Black-Scholes model i.e. with no volatility (GARCH) module. We generated 20 Black-Scholes iteration cores on the same FPGA, and set the clock frequency to 75MHz. This resulted in a 750x speedup compared to our software implementation, which is far superior to the 85x speed-up figure reported in [6] However, the latter paper does not report the number of Monte-Carlo cores implemented on the device, nor does it report the area consumed by each Monte-Carlo core. The maximum clock frequency is also not reported. It is hence not possible to extrapolate these results to our case study accurately. Nonetheless, it is safe to say that this implementation is slower than ours, given the relative sizes of the FPGA devices used, and the speed-up figures reported. Our implementation

was also scaled to a network of FPGA devices, unlike the single-device implementation reported in [6] .

The closest comparator to our work is the demonstration application on the Maxwell machine [7] , which resulted in the FPGA implementation outperforming its CPU counterpart by a factor of 323. However, the implementation reported in [7] was for Asian options and was without the stochastic volatility module, i.e. the GARCH module in Figure 5, which is in the critical path in our design and occupies about 30% of the overall resources used. To make a more sensible comparison, we removed our GARCH module and added a module used for calculating an averaging of intermediate option prices in time, which is needed for Asian options pricing. With this option added to our core, the speed-up figure we realised was 600x, which is still substantially faster than the 323x speed-up reported in [7] . Careful pipelining and block design and mapping are behind this substantial speed-up.

VI. Conclusion

This paper presented a hardware accelerated implementation of a Monte-Carlo simulation engine for option pricing with stochastic volatility, on a 64-FPGA supercomputer. The fully parallelized and pipelined hardware core results in considerable speed-up compared to an equivalent software implementation (340x). Moreover, the whole design, implementation and testing was achieved in 5 months by two PhD students on their first year of study. This shows that reconfigurable technology can be an efficacious and efficient platform for supercomputing in general. A major enabler for this was the availability of a powerful board/system support package, as well as extensive use of IP cores. This has to be replicated in the future if we are to reach the same conclusion. However, it is worth mentioning that the application considered in this paper was massively parallel with little inter-communication between processes. Future applications are unlikely to follow the same pattern, and as a result the use of inter-FPGA RocketIO links will be needed. Future research agenda includes the extension of this work to a larger set of option pricing models, and financial applications in general. We also intend to experiment with floating-point, fixed-point and other arithmetic types, looking for trade-offs between precision, speed and resources consumption.

References

[1] Hull, J.C. Option, futures, and other derivatives. Prentice Hall, Upper Saddle River, 2000.

[2] Black, F. and Scholes, M. The Pricing of Options and Corporate Liabilities. The Journal of Political Economy. 637-654.

[3] Bollerslev, T. Generalized autoregressive conditional heteroskedasticity. Journal of econometrics, 31. 307-327.

[4] Zenios, S.A. High-performance computing in finance: The last 10 years and the next. Parallel Computing, 25 (13-14). 2149-2175.

[5] Zhang, G.L., Leong, P.H.W., Ho, C.H., Tsoi, K.H., Cheung, C.C.C., Lee, D.-U., Cheung, R.C.C. and Luk, W., Reconfigurable Acceleration for Monte-Carlo based Financial Simulation. In Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, (2005), 215 - 222.

[6] Thomas, D.B., Bower, J.A. and Luk, W., Hardware architectures for Monte-Carlo based financial simulations. In Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on, (2006), 377 - 380.

[7] Baxter, R., Booth, S., Bull, M., Cawood, G., Perry, J., Parsons, M., Simpson, A., Trew, A., McCormick, A., Smart, G., Smart, R., Cantle, A., Chamberlain, R. and Genest, G., Maxwell - a 64 FPGA Supercomputer. in Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on, (2007), 287-294.

[8] Boutillon, E., Danger, J.-L., and Ghazel, A. Design of high speed AWGN communication channel emulator. Analog Integrated Circuits and Signal Processing. 133-142.

[9] G.E.P. Box et al., A note on the generation of random normal deviates, Ann. Math. Statist., Vol. 29, (1958), 610-611.

[10] R.C. Tausworthe, Random Numbers Generated by linear Recurrence Modulo Two, Math. And Computation, vol. 19, (1965), 201-209.

[11] Mencer, O., Boullis, N., Luk, W. and Styles, H., Parameterized Function Evaluation for FPGAs. in Proceedings of the 11th International Conference on Field-Programmable Logic and Applications, (2001), 544-554.

[12] Knuth, D.E. The Art of Computer Programming, Seminumerical algorithms. Addison-Wesley, 1997.

[13] The FPGA High Performance Computing Alliance, http://www.fhpca.org

[14] Xilinx, Virtex-4 Family Overview, Product Specification, http://www.xilinx.com, DS112 (v3.0), September 28, 2007.

[15] Snir, M. and Otto, S. MPI-The Complete Reference. MIT Press, Cambridge, MA, USA, 1998.