Efficient Reconfigurable Design for Pricing Asian Options

Anson H.T. Tse, David B. Thomas, K.H. Tsoi, Wayne Luk
Department of Computing
Imperial College London, UK
{htt08,dt10,khtsoi,wl}@doc.ic.ac.uk

ABSTRACT

Arithmetic Asian options are financial derivatives which have the feature of path-dependency: they depend on the entire price path of the underlying asset, rather than just the instantaneous price. This path-dependency makes them difficult to price, as only computationally intensive Monte-Carlo methods can provide accurate prices. This paper proposes an FPGA-accelerated Asian option pricing solution, using a highly-optimised parallel Monte-Carlo architecture. The proposed pipelined design is described parametrically, facilitating its re-use for different technologies. An implementation of this architecture in a Virtex-5 xc5vlx330t FPGA at 200MHz is 313 times faster than a multi-threaded software implementation running on a Intel Xeon E5420 quad-core CPU at 2.5GHz; it is also 2.2 times faster than the Tesla C1060 GPU at 1.3 GHz.

Keywords

FPGA, GPU, Asian option pricing, acceleration

1. INTRODUCTION

Arithmetic Asian options are examples of *derivatives*: financial instruments whose value is dependent on the price of some underlying asset such as a bond or stock. Unlike simpler options, which provide a payoff depending on the instantaneous price of the underlying, arithmetic Asian options provide a payoff depending on the arithmetic average price of the underlying during the option life-time. This averaging makes arithmetic Asian options cheaper and less sensitive to market manipulation, but also means there is no closed-form solution for the pricing.

Monte-Carlo methods provide a accurate way to price Asian options, but have slow convergence, so a huge number of simulations is needed. FPGAs have previously been explored for financial Monte-Carlo simulations, such as pricing European and American options, as they can exploit the large amounts of parallelism found in simulations. This paper extends the scope of FPGA accelerated simulations to include path-dependent Asian Options.

Our contributions are:

- A highly-optimized pipelined parallel FPGA architecture for implementing Asian option pricing; this design is described parametrically to facilitate its re-use for different technologies.
- A GPU algorithm for implementing Asian option pricing based on CUDA API.
- Evaluation of the FPGA and GPU implementations versus a single-thread implementation on CPU, showing 313 times speedup for the FPGA, and 141 times for the GPU.

This work was presented in part at the first international workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2010), Tsukuba, Ibaraki, Japan, June 1, 2010.

2. ASIAN OPTIONS

An option is a type of financial instrument which provides the owner of the option with the right, but not the obligation, to buy or sell an underlying asset such as a stock or bond at some point in the future. A *call option* allows the option owner to buy the underlying asset for some pre-agreed strike price K, while a *put option* gives them the right to sell at price K. The decision to exercise the option (i.e. buy or sell the asset) is always made by the option owner, and the option issuer has to abide by that decision, so the option owner must pay the issuer to create the option. Hence putting an accurate value on an option is critical for both parties.

For simple European call options, the owner can exercise only at the expiry date. If the underlying asset price S at expiry date is higher than the strike price K, the owner can profit by buying the stock at lower price K from the option issuer and then immediately selling it at the higher price S in the market, providing a gain of $(S-K)$. If the underlying asset price is lower than the strike price, $S < K$, then the gain is zero because the option will not be exercised.

The payoff of European call option on expiry is

$$P_{call} = \max(S - K, 0) \tag{1}$$

and the payoff of European put option at expiry is

$$P_{put} = \max(K - S, 0). \tag{2}$$

For an arithmetic Asian option [4], the payoff is calculated using the arithmetic average of the prices over the life time of the option. One advantage of this option type is that it is more difficult for the option issuer to manipulate market prices to reduce the option payoff, as the payoff depends on the path followed by the asset price, not just the price at expiry.

The payoff of an arithmetic Asian call option is:

$$P_{call} = \max\left(\frac{1}{n+1}\sum_{i=0}^{n} S(t_i) - K, 0\right) \tag{3}$$

Table 1: An example of stock price paths ($S_0 = 1.00, K = 1.03, T = 2, r = 0.1$)

Path:	$t = 0$	$t = 1$	$t = 2$	Avg	Payoff
Path 1	1.00	1.22	1.25	1.16	0.13
Path 2	1.00	1.18	1.41	1.20	0.17
Path 3	1.00	0.92	0.88	0.93	0.00
Path 4	1.00	1.11	1.32	1.14	0.11
Path 5	1.00	0.99	1.09	1.03	0.00
Path 6	1.00	1.16	1.09	1.08	0.05
Path 7	1.00	1.19	1.39	1.19	0.16
Path 8	1.00	0.91	0.86	0.92	0.00
Path 9	1.00	1.22	1.21	1.14	0.11
Path 10	1.00	0.94	0.84	0.93	0.00
Avg Payoff					0.07

where $t_1..t_n$ are the times at which the asset price is observed, and $S(t)$ is the asset price at time t.

A common assumption is that asset prices move according to a log-normal random walk. Under this model, price of an European option at present time can be calculated with a closed-form solution called the Black-Scholes Equation [1]. However, there is no such solution for arithmetic Asian options, due to their highly path-dependent properties. Monte-Carlo methods are commonly used to solve this problem. The idea is to generate a huge number of random paths for each probabilistic variable, then take the average of the results. We illustrate the idea with an pricing example with the following parameters $S_0 = 1.00, K = 1.03, r = 0.1, T = 2$ and $steps = 2$. Table 1 shows an example of stock price paths. Firstly, 10 stock price paths from $t = 0$ to $t = 2$ are simulated. Then the average stock price for each path is calculated as in 'Avg' column. The payoff of each path is then calculated according to Equation 3 as in 'Payoff' column. Finally, the average payoff across all these paths is calculated. The final result is the expected value of the arithmetic Asian call option at $t = 2$. The arithmetic call option value at present time can be obtained by discounting this final answer backward by multiplying e^{-rT}. The option price in the above example is 0.057.

3. RELATED WORK

Financial analysis and pricing applications are often computationally intensive, so there has been much interest in the use of accelerators such as FPGAs and in this domain. Research into FPGA-accelerated pricing can be broadly split into two groups: lattice methods, which work backwards from exercise time to the current price, using a pre-determined lattice of asset prices and times; and Monte-Carlo methods, which work forwards from the current asset price to expiry time using multiple randomly chosen paths.

Lattice methods targeting FPGAs include binomial trees [3], finite-difference methods [2], and Quadrature pricing [11]. Such algorithms are generally more efficient than Monte-Carlo methods, but they cannot easily handle complex features, such as path-dependence in Asian options.

Monte-Carlo methods are particularly suitable for implementation in FPGAs, as they contain abundant parallelism: each asset-price path can be evaluated in parallel with any other path, allowing exploitation of coarse-grained and pipeline parallelism. An FPGA-accelerated Monte-Carlo application covers pricing under the BGM interest rate model [12]. This provides 25 times speedup over software, using an optimised VHDL design and customised data widths. Other designs for European option have also been presented [10].

Recent work has focused on complex types of Monte-Carlo simulation, such as correlated asset prices [7], American exercise features [9], and discrete-event simulations [8].

This paper extends parallel reconfigurable design for Monte-Carlo simulation to include path-dependent arithmetic Asian options, describing how path-dependency can be incorporated without sacrificing performance.

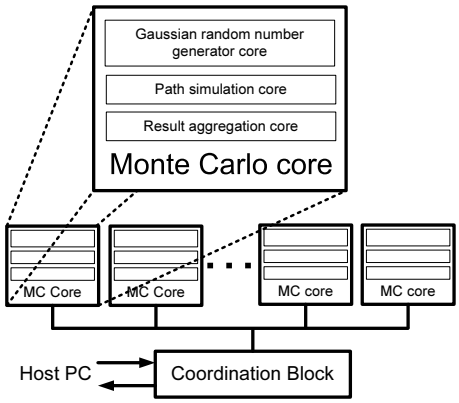

Figure 1: Overall hardware architecture.

4. HARDWARE ARCHITECTURE

In this section, we present our hardware design for Asian option pricing. As shown later, the operator latency is described parametrically, facilitating the design to be reused in different technologies. Figure 1 shows the overall hardware architecture. There are two main types of components in the design: one or more identical Monte-Carlo cores (MC); and a single shared Coordination Block (CB). The Monte-Carlo cores contain a Gaussian random number generator, a path simulation core and a result aggregation core; each MC core is capable of generating random asset price paths, calculating payoffs, and accumulating the average payoff. Multiple identical MC cores are instantiated to make maximum use of the device, so the Coordination Block manages the MC cores, allowing them to work in parallel to price the same option. The CB is also responsible for communicating with the external controller, for example a PC.

4.1 Monte-Carlo core

4.1.1 Gaussian random number generator

The Gaussian random number generator uses the piecewise linear generation method [6], which provides high-quality fixed-point Gaussian samples, while using only a small amount of logic and block-RAMs. To provide a good approximation to the Gaussian distribution, two independent piecewise linear RNGs are used, both of which provide a good approximation to the Gaussian distribution. The outputs of the generators are then added together, providing a better approximation to the Gaussian distribution, due to the Central Limit Theorem.

The resulting Gaussian RNG produces a stream of 24-bit fixed-point random numbers, with a period of 2^{128}. The quality of the stream has been checked with the Chi-squared test for sample sizes up to 2^{32}, and shows no significant deviation from the Gaussian distribution.

Algorithm 1 Monte-Carlo pricing algorithm

```
payoffSum = 0
for i = 1 to NumberOfSimulation do
  SumOfPrice = S_0
  S = S_0
  for i = 1 to Steps do
    W ← NextRandomNumber
    S ← S e^{drift+vsqrdt×W}
    SumOfPrice ← SumOfPrice + S
  end for
  payoff ← SumOfPrice / (Steps+1) - K
  payoff ← max(0,payoff)
  payoffSum ← payoffSum + payoff
end for
Price ← e^{-rT}(payoffSum/NumberOfSimulation)
```

4.1.2 Path simulation

Under Black Schole's model where the stock price movement is governed by a geometric Brownian motion process, the stock price is given by the equation:

$$S_{i+1} = S_i e^{((r-\frac{v^2}{2})\delta t + v\sqrt{\delta t}W)} \quad (4)$$

where r is the interest rate, v is the volatility of the underlying stock price, δt is the time period between two time steps, W is a Gaussian random number $\sim N(0,1)$, S_i is the underlying stock price at step i and S_{i+1} is the underlying stock price at step $i+1$. We could define the following equations:

$$drift = (r - \frac{v^2}{2})\delta t \quad (5)$$

$$vsqrdt = v\sqrt{\delta t} \quad (6)$$

such that

$$S_{i+1} = S_i e^{drift + vsqrdt \times W} \quad (7)$$

The values of $drift$ and $vsqrdt$ can be precomputed in advance and the path simulation core can simulate the stock price movement with these two static values. The payoff of arithmetic Asian options at expiry time is:

$$payoff = \max(0, \frac{1}{n+1}\sum_{i=0}^{n} S_i - K). \quad (8)$$

The option price at present time is defined as the payoff discounted backward to the current time as the following equation:

$$OptionPrice = e^{-rT} payoff \quad (9)$$

where T is the time to expiry. Therefore, we have the Monte-Carlo Asian option pricing algorithm as shown in Algorithm 1.

The hardware architecture of the price movement path simulation based on the Algorithm 1 is shown in Figure 2. The static input parameters include S_0, K, $vsqrdt$, $drift$ and $steps$ (number of simulation steps). The dynamic input parameter is the Gaussian random number W. The underlined parameter near each operator is the number of pipeline stages (latency) of that operator. Therefore, it takes $d_a + d_m + d_e$ clock cycles for W to reach the second multiplication operator.

There are 3 multiplexers namely MUXA, MUXB and MUXC controlling the computation flow. MUXA selects S_0 at the beginning in order to calculate the S_1 price. The

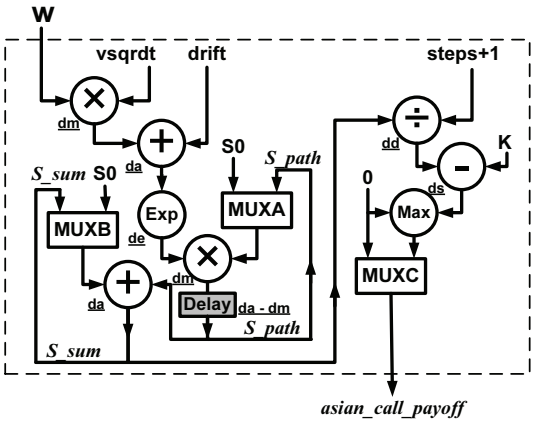

Figure 2: Architecture of the price movement path simulation core. The underlined parameter denotes operator latency.

Table 2: MUXs' behavior in path simulation

MUX	Selecting behavior:
MUXA	Select S_0 for first $d_a + d_m + d_e$ cycles. Repeat: Select S_0 for p cycles and then select S_path for $p \times (steps - 1)$ cycles
MUXB	Select S_0 for first $2d_a + d_m + d_e$ cycles. Repeat: Select S_0 for p cycles and then select S_path for $p \times (steps - 1)$ cycles
MUXC	Select 0 for first $3d_a + d_m + d_e + d_d + d_s$ cycles. Repeat: Select 0 for $p \times (steps - 1)$ cycles and then select max() output for p cycles

signal s_path indicates the updated S in the path and is feed back to MUXA. Therefore, MUXA selects signal s_path afterward to provide a loop for iterating next S_i. The loop containing MUXA and the second multiplication operator is the "stock price updating loop".

MUXB selects S_0 at the beginning in order to compute the sum of price S_0 and S_1 to the result signal S_sum. S_sum is then feed back to MUXB to form the "sum of price updating loop". MUXB selects S_sum afterward after the $S_0 + S_1$ computation.

The number of the pipelined stages must be identical for all the pipelined loops in order to guarantee a consistent computation schedule. Let p be the maximum number of pipeline stages for these pipelined loops. Pipelined registers are added to ensure the number of pipelined stages of all loops equal to p. In this architecture, $p = d_a$. Therefore, a $d_a - d_m$ cycles pipelined delay register is inserted after the second multiplication operator for balancing.

As the computation is pipelined, the feedback result will reappear at the MUXA and MUXB after p cycles. Therefore, we simulate p paths at the same time in this pipelined fashion. The computed 1 step of S for the first simulation will arrive MUXA and MUXB just after the other $p-1$ computations of the other simulations.

MUXC selects the output of max operator only when that p simulations reached the end of the path (i.e. S_path reached S_n). Therefore, MUXC only selects the max operator output for p cycles as the $asian\_call\_payoff$ at that moment, and selects value 0 for the rest of time. In conclusion, p path simulations are completed after $p \times steps + 3d_a + d_m + d_e + d_d + d_s$ cycles for the pipeline stages. The whole process repeats and we could expect another p completed path sim-

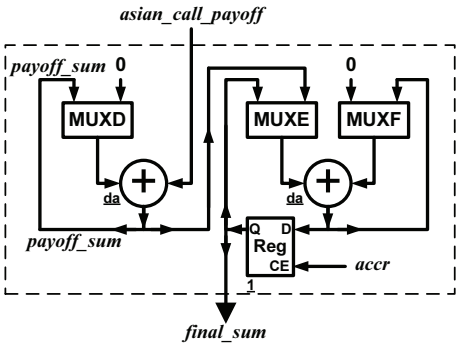

Figure 3: Architecture of the result aggregation core.

Table 3: MUXs' behavior in result aggregation

MUX	Selecting behavior:
MUXD	Select 0 for $p \times steps+$ $4d_a + d_m + d_e + d_d + d_s$ cycles and then select *payoff_sum* afterward.
MUXE	Don't care for $p \times steps \times N_{batch}+$ $4d_a + d_m + d_e + d_d + d_s$ cycles and then select *payoff_sum* for p cycles and then select RegOut afterward.
MUXF	Select 0 for $p \times steps \times N_{batch}+$ $5d_a + d_m + d_e + d_d + d_s$ cycles and then select AdderOut afterward.

ulations after another $p \times steps$ cycles. Table 2 summarize the behavior of the MUXs in the path simulation core.

4.1.3 Result aggregation

The architecture of the result aggregation core is shown in Figure 3. As discussed in Section 4.1.2, a batch of p payoff results are generated for every $p \times steps$ cycles and passed to the result aggregation core. These payoff results are accumulated until the number of batches reached the required number of batches. Let N_{batch} be the required number of batches, N_{mc} be the required number of Monte-Carlo simulations and C be the number of MC cores in the hardware. N_{batch} is defined as:

$$N_{batch} = \lceil \frac{N_{mc}}{p \cdot C} \rceil \tag{10}$$

MUXD selects 0 for the first $p \times steps + 4d_a + d_m + d_e + d_d + d_s$ cycles for initialization, then it selects the accumulated payoff (signal *payoff_sum*) afterward to form a "sum of payoff loop". When the number of batches reached N_{batch}, we have to aggregate the final p consecutive *payoff_sum* values together.

To aggregate these p consecutive values using a *p-stage* pipelined adder, we make use of one direct feedback loop and one feedback loop involving a register with special clock-enable timing. MUXF selects 0 for initialization and selects the output of the last adder afterward to form a direct feedback loop. MUXE only selects the *payoff_sum* for p cycles when the final p consecutive values are ready and then selects the output of the register. The input of the register is connected to the output of the last adder. In other words, the register and MUXE form another feedback loop with a register in the middle. Table 3 shows the behavior of MUX in the result aggregation core.

The clock-enable of the D-type register is controlled by a special signal sequence "accr". The "accr" is set to 1 for a dedicated timing so as to buffer the desired intermediate output of the adder. The desired intermediate result stays at the output of the register and the output of the MUXE.

If x is the index of clock cycle and p is the number of pipeline stages, the sequence of signal "accr" is given by the following equation:

$$accr(x) = \begin{cases} 1 & \text{if } x \bmod 2^{k+1} = 2^k - 1, \\ & pk \le x < p(k+1), \\ & \forall k \in N, 0 \le k \le \log_2(p) \\ 1 & \text{if } x \bmod 2^k = 2^{k-1} - 1, \\ & pk \le x < p(k+1), \\ & \forall k \in N, \log_2(p) < k \le \log_2(p) + 1 \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

Let Y_i be the values to be aggregated. From Equation 11, $Y_1 + Y_2, Y_3 + Y_4, ...$ are computed in the first p cycles and $Y_1 + Y_2 + Y_3 + Y_4, Y_5 + Y_6 + Y_7 + Y_8, ...$ are computed at the second p cycles. Similarly, $\sum_{i=1}^{p} Y_i$ will be computed and appear at register out. The number of cycles required to obtain the final sum is $\lceil p(\log_2(p) + 1) \rceil$.

This register, with a special clock-enable signal sequence, is of general use for a design requiring a "reduce" function in a "map-reduce" computation with commutative operator with any number of pipeline stages. If a multiplier is used as the commutative operator, the result of $\prod_{i=1}^{p} Y_i$ will be computed at the register output instead of $\sum_{i=1}^{p} Y_i$.

4.2 Coordination Block

4.2.1 Functional description

The Coordination Block is the main control unit of the overall hardware architecture. It provides the communication with the host PC. The option parameters are first sent from host PC to the Coordination Block. The Coordination Block then distributes the parameters to all MC cores. The communication time between the FPGA and PC is negligible as there are only a few bytes of input parameters and results transferred between them.

The Gaussian random number generators in MC cores are also initialized by the Coordination Block. Different sequences of bits are connected to different Gaussian random number generators as the random seeds. The Coordination Block also controls the overall timing of the computation. It generates MUXs selection signals and 1 "accr" signal sequence to all MC cores as discussed in previous subsection. The timing of generating these signals is followed strictly by the requirement as in Table 2 and Table 3.

4.2.2 Path delay optimization

The counters, condition checking and controlling logic are implemented in the CB only instead of in the MC cores. In this way, the logic redundancy is significantly reduced. However, the use of global controlling signals may suffer from a decrease of clock rate due to a long critical path delay.

Path delay consists of 2 parts: logic delay and routing delay. When there are many computational cores, the routing delay of the controlling signal to the farthest computational core will be significant. If the logic delay of producing that controlling signal is long as well, the performance of a parallel architecture will be drastically reduced.

For example, if we set a global controlling signal m to 1 after a complex "condition A" checking, the example hardware code is as follow:

if Condition A **then**
  $m <= 1$
end if

Assume that the logic delay for checking complex condition A is 4ns. If there is only 1 MC core, the routing delay for signal m from the logic result to the MC core is very short (<1ns). Therefore, the total path delay is less than 5ns and the hardware could be running at 200MHz. However, if there are many MC cores, the routing delay to the farthest MC core will be very long (up to 4ns in our experiment). If the routing delay is 4ns, the total path delay becomes 8ns and it could only be running at 125MHz, which is a drastic performance reduction.

Therefore, the hardware design of CB is optimized carefully with path delay partitioning. For the above hardware code, we divide it into two parts:

if 1 cycle before Condition A **then**
  $state <= SETM$
end if

if $state = SETM$ **then**
  $m <= 1$
end if

The routing delay of setting an internal state to $SETM$ is very short (<1ns). Therefore, the path delay of the first part is less than 5ns even with the complex condition A checking.

The logic delay of checking "state = $SETM$" is also very short (<1ns). Therefore, the path delay of the second part including the long routing delay is still less than 5ns. As a result, the overall hardware could still be running at 200MHz.

This path delay partitioning optimization technique can be applied to any hardware design involving one control module sending controlling signals to multiple computational cores. It is an essential technique to maintain a high clock rate while maximizing the degree of parallelism.

5. GPU IMPLEMENTATION

Graphics Processing Units (GPUs) have been used for acceleration in various application [5]. They are Single Instruction Multiple Data (SIMD) computing devices. Parallelizable tasks are executed on the GPU as "kernel" by a computation grid. Each computation grid consists of a grid of thread blocks. Each block and thread has a unique block ID and thread ID. The "kernel" is executed by all threads in parallel with the same code, but on different sets of data. Intra-block communication between threads can be through the shared memory. The inter-block communication can be through the global memory. The speed of accessing thread's local register is the fastest. The speed of accessing shared memory is faster than accessing global memory. Our implementation on GPU is based on Compute Unified Device Architecture (CUDA) API provided for nVidia GPUs. A typical CUDA co-processing flow involve 4 steps:

- Copy processing data to GPU memory from main memory of the host.
- Instruct GPU to start processing.
- Wait till the threads inside GPU finished executing the kernel in parallel.
- Copy the result back to main memory.

Table 4: FPGA resource consumption

16 MC Cores		
Resource	**Used**	**Utilization**
Slices	41,968	80%
FFs	110,789	53%
LUTs	95,749	46%
RAM	16	4%
DSP48Es	192	100%

We design our CUDA implementation of Asian option pricing by 2 procedures, namely Gaussian random number generator procedure and path simulation procedure.

5.1 Gaussian random number generator procedure

In this procedure, we first allocate the GPU's global memory space for the total amount of random numbers that we needed for the simulations. If the number of simulations is N, the number of steps is M and single precision is used, we allocate 4NM bytes of global memory in GPU. Then we execute the Mersenne Twister random number generator kernel using all the threads to generate random numbers at the memory space. A Box-Muller transformation kernel is then executed on that memory space to form Gaussian random numbers.

5.2 Path simulation procedure

In the path simulation procedure, each thread simulates the price movement path as Algorithm 1 and sum up the payoff in the shared memory. Therefore, these payoff sums can be accessed by other threads in the same block. The first thread in each block then sum up all the payoff sums within the same block and stored it in the global memory location. Finally, a final aggregation kernel is executed by 1 thread only. This thread sums up the results by all blocks from the global memory location and returns the total payoff sum to the main program. The main program then compute the option price from the returned result.

6. IMPLEMENTATION RESULT

Our FPGA implementation targeted a Xilinx xc5vlx330t FPGA chip on an Alpha Data ADM-XRC-5T2 card, which contains 207,360 slices, 192 DSP48E, 324 BlockRAM units. We design our hardware architecture manually in VHDL to maximize performance. The design is synthesized, mapped, placed and routed using Xilinx ISE 10.1.03. The floating operators used are from Xilinx Floating-Point Operator 4.0. Single precision floating point arithmetic is used so that the parameters in Figure 2 and Figure 3 are given by: $d_a = 12, d_m = 8, d_e = 14, d_d = 27, d_s = 12$ and $p = 12$. There are 16 MC cores in the design. The summary of resource consumption is shown in Table 4. Our result shows that no more MC cores can be fitted in the hardware, as all the DSP48Es are used up for 16 MC cores.

The performance of different implementations of arithmetic Asian call option pricing is evaluated. We choose an arithmetic Asian call option with parameters $S_0 = 100, K = 105, v = 0.15, r = 0.1, T = 10$ and $steps = 3650$. The number of Monte-Carlo simulations is 10,000,000. The acceleration of FPGA implementations and GPU implementations is compared to a reference software implementation. The ref-

Table 5: Performance comparison of pricing an arithmetic Asian call option

	FPGA	GPU	CPU
Type	xc5vlx330t	Tesla C1060	Xeon E5420
Frequency	200MHz	1.3GHz	2.5GHz
Time(s)	11.4s	25.31s	3580.97s
Speedup	313x	141x	1x
Max Power	34W	200W	80W

erence PC contains an Intel Xeon E5420 2.5GHz processor with 16GB RAM. The software implementation is written in the C language with the *gsl* library and compiled using *gcc* with speed optimization options `-O3` using OpenMP multi-threaded API. The number of threads used in software implementation is 4 in order to fully utilize the 4 cores in Xeon E5420. The targeted GPU is nVidia Tesla C1060 with 4GB of on board RAM. The summary of the performance comparison is shown in Table 5.

From the results, it can be seen that a speedup of 313 times is achieved by the xc5vlx330t FPGA. For the GPU, a speedup of 141 times is achieved by Tesla C1060. The xc5vlx330t FPGA is 2.2 times faster than the Tesla C1060 GPU. The time for pricing an arithmetic Asian option is reduced from 1 hour to 11.4 seconds.

The maximum power usage of different devices is also estimated in Table 5. The power usage of xc5vlx330t is estimated by Xilinx XPower Estimator 11.4.1 with toggle rate 100% and clock rate 200MHz. Since in our design all the flip-flops do not actually toggle all the time, setting the toggle rate to 100% is just to estimate the upper bound of power usage. It is interesting to note that the xc5vlx330t FPGA demonstrates higher performance in speed than the GPU and CPU, and consumes less power. If we have a cluster of PCs with Intel Xeon 2.5GHz without communication overhead between the nodes, then a cluster with 313 PC nodes would achieve the same performance as an xc5vlx330t FPGA. However, the energy consumption of that cluster would be 736 times more than an xc5vlx330t FPGA.

7. CONCLUSION

This paper presents a high performance hardware architecture for Asian option pricing. A CUDA based GPU implementation is also presented for performance comparison. To our knowledge, this is the first reported hardware implementation that accelerates the arithmetic Asian option pricing algorithm. By exploiting efficient Gaussian random number generators, massive parallelism and highly pipelined datapath, our FPGA implementation is faster than a comparable multi-threaded software implementation by 313 times, and it is faster than a CUDA based GPU implementation by 2.2 times. The maximum power consumption of the FPGA is also estimated to be much lower than those for the CPU and the GPU. This performance improvement in speed and in power consumption offers a practical solution to financial institutions to reduce option pricing time and costs, with considerable energy savings.

The presented hardware architecture demonstrates a practical reference design for high performance path-dependent financial simulation. With a slight modification of the path simulation core, the hardware architecture can be used for pricing any exotic and path-dependent financial options including lookback options and barrier options.

Future work includes the design of a distributed financial computation framework in a heterogeneous cluster with FPGAs, GPUs and CPUs. The automated generation of the proposed efficient architectures will also be investigated.

Acknowledgments. The support of Imperial College London Research Excellence Award, UK Engineering and Physical Sciences Research Council, Alpha Data, nVidia, and Xilinx is gratefully acknowledged.

8. REFERENCES

[1] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[2] Q. Jin, D. Thomas, and W. Luk. Exploring reconfigurable architectures for explicit finite difference option pricing models. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 73 –78, 2009.

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

[4] A. G. Z. Kemna and A. C. F. Vorst. A pricing method for options based on average asset values. *Journal of Banking & Finance*, 14(1):113–129, March 1990.

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

[6] D. B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 1–6, 2006.

[7] D. B. Thomas and W. Luk. Sampling from the multivariate Gaussian distribution using reconfigurable hardware. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–12, 2007.

[8] D. B. Thomas and W. Luk. Credit risk modelling using hardware accelerated Monte-Carlo simulation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[9] X. Tian and K. Benkrid. American option pricing on reconfigurable hardware using least-squares Monte Carlo method. In *Proc. Int. Conf. on Field-Programmable Technology*, pages 263 –270, 2009.

[10] X. Tian, K. Benkrid, and X.Gu. High performance Monte-Carlo based option pricing on FPGAs. *Engineering Letters*, 16(3):434–442, 2008.

[11] A. H. T. Tse, D. B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

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