A Reconfigurable Simulation Framework for Financial Computation

Jacob A. Bower, David B. Thomas, Wayne Luk and Oskar Mencer
Department of Computing, Imperial College, 180 Queen's Gate
London SW7 2AZ, UK
{jab00,dt10,wl,oskar}@doc.ic.ac.uk

Abstract

This paper presents a framework for the acceleration of Monte-Carlo simulations using reconfigurable hardware. Discrete-time random walk simulations are widely used in the financial computation to calculate derivative prices and evaluate portfolio risk, but increases in model complexity and tighter time constraints now require large computer farms to meet operational demands. We present a model for accelerating such tasks with reconfigurable hardware, using an architecture that exploits parallelism at multiple levels, combining fine-grained pipelining, intra-device multi-threading, and inter-device distributed processing. The architecture adopts a modular design approach, allowing components to be re-used across different applications, while also allowing automatic design space exploration to maximise performance within different devices. Using our framework, we implement two different discrete-time random walks representative of financial simulations, and these show 71 times and 8 times speedup respectively when compared to a C++ software and SSE vectorised implementations.

1 Introduction

As transistor sizes continue to shrink, Field Programmable Gate Arrays (FPGAs) benefit from increased logic densities and increased performance. As a a result, FPGAs now have the ability to implement application-optimised floating-point operations, allowing them to outperform contemporary high-performance microprocessors [1], and so are becoming a viable platform for the acceleration of complex numerical applications. However, while FPGAs exhibit potential for accelerating a wide range of applications, their practical use in industry for accelerating complex arithmetic is currently limited. One reason for this is that any eventual speed-up is often outweighed by the large up-front design effort required when moving algorithms into hardware, particularly in applications where the algorithms are continuously evolving. For example, in the banking sector new financial models and risk-evaluation strategies are constantly developed, and only bring competitive advantage if they can be used in hardware immediately.

In this paper, we present a general purpose framework for accelerating Monte Carlo simulations of discrete-time random walks. Our framework reduces the design effort for simulations in this class, enabling rapid implementation of a single simulation kernel with a well defined interface. From these kernels we are able to extract parallelism at a number of different levels including: running multiple simulations in one kernel pipeline, multiple pipelines in one FPGA and running multiple FPGAs in parallel, distributed over a network.

We focus on Monte Carlo simulations as these are examples of "embarrassingly parallel" applications [2], making them well suited to hardware implementation. FPGAs are at an advantage for implementing Monte Carlo simulations in hardware, as the diversity of simulation models preclude a sufficiently general ASIC implementation. We specifically target discrete-time random walks, as these are of significance in financial computation. Financial computation is an attractive application domain for hardware acceleration, as simulation/analysis needs to be performed in a very short space of time during intense trading activities. As such, rapidly increasing the rate of processing of these simulations can be seen as an enabling technology as users are able to react quicker to dynamic scenarios.

In this paper we present the following contributions:

- A framework for developing discrete-time, random walk Monte Carlo simulations, which both encourages component re-use, and simplifies the implementation and optimisation of fine-grained and coarse-grained parallelism.
- Application of the proposed framework to two simulation applications representative of current financial computing models.
- Performance results from an FPGA implementation, showing speed ups of more than 8 times from a single

FPGA, compared to highly optimised software implementations.

For the remainder of this paper, our discussion will be divided up as follows. First we discuss background in Section 2. We present our Monte Carlo framework in Section 3, discussing how parallelism is extracted both in hardware and by distribution across networked FPGA host nodes. In Section 4 we describe our implementations of Ho-Lee and Vasicek models within our framework, performance results from which are presented in Section 5. Finally in Section 6 we conclude our work and suggest areas for further research.

2 Background

Many simulations of real-world systems make use of the Monte Carlo method. In a Monte Carlo simulation, a model of a real-world system is repeatedly run with inputs derived from appropriately distributed random numbers. Results from a large number of these runs are combined, for example by averaging, to infer probabilistic results of a system. This method of simulation is advantageous as it allows analysis of systems which are either intractable to model fully due to their size, or which are based on the occurrence of probabilistic events.

In this paper we focus on random walk Monte Carlo simulations, where a mathematical model is effectively "walked" through a space, with the direction and distance of each step derived from random inputs. Random walk Monte Carlo simulations are commonly used in finance for the pricing of derivatives [3], where random walks model the movement of asset prices as time progresses.

The quality of results from a Monte Carlo system depends primarily on the number of independent simulation runs. As increasing numbers of independent trials are combined, the calculated answer will approach the true answer. Confidence intervals can be calculated from the standard deviation of the answer, so when a desired precision is required, more simulation runs can be aggregated until the standard deviation falls below the target precision. However, the error in results from Monte Carlo simulations decreases as $1/\sqrt{N}$ [4], so doubling the precision of the answer will require four times as many simulation runs, making the simulation speed critical. Fortunately, each simulation run is completely independent, allowing them to be executed in parallel.

FPGAs, with their increasing capacity and potential for massively parallel computations, are obvious candidates for accelerating simulations. Examples of such simulations accelerated by FPGAs include modelling of radar [5], communication [6], biological [7], physical [8] and financial systems [9]. Also related to the study of Monte Carlo simulations is the subject of generating random numbers with appropriate distributions for use in such simulations. Work on the implementation of random number generators in FPGAs with varying degrees of speed, size, quality of randomness, and varying distributions include [10, 11, 12, 13].

In [9] and [8], the authors explore implementing Monte Carlo simulations in FPGAs. However, both of these works focus on optimisation of their specific target applications, rather than on developing frameworks that can be re-used in other applications. The authors also do not consider the use of multiple FPGAs, which is a key requirement when a single FPGA cannot provide sufficient performance to meet application constraints.

In this paper we present a general Monte Carlo framework, focusing on extracting parallelism arising from the independent nature of Monte Carlo simulation runs, and the pipelineability of the simulation steps.

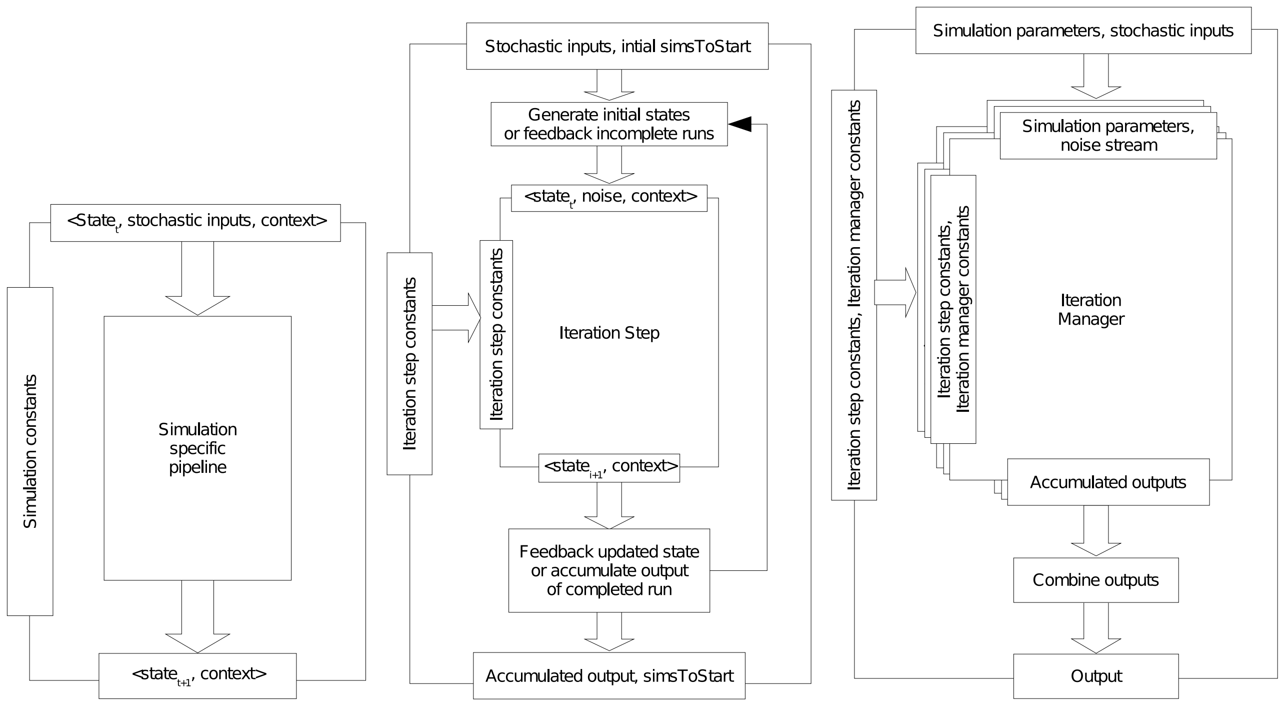
3 Monte Carlo Framework

3.1 Overview

In this section we describe our framework for implementing discrete-time random walk simulations in reconfigurable hardware such as FPGAs. This framework allows designers to focus on just the simulation specific components, with components common to all simulations provided by the framework. In particular the framework encourages and simplifies the use of three different levels of parallelism:

- Filling pipelined algorithm kernels with the data from multiple independent simulations, allowing a form of C-Slow [14] optimisation.
- Managing the execution of multiple pipelined simulation cores in one device.
- Running multiple devices at once, distributed over a network cluster.

At the core of any discrete-time Monte Carlo simulation is an algorithmic kernel which models the behaviour of an entity over a simulated period of time, such as the value of an asset or changes in interest rates. A simulation progresses in fixed times steps through the range $0 \leq t \leq T$, where T is the time horizon of the simulation. As a simulation progresses it will iterate on some form of state, starting with a fixed initial state, common to all simulation runs. The nature of this state, and how it is transformed, will be simulation specific, though typically the transformation will be performed by a single deterministic function. This function takes as input the current state, one or more random numbers, and any global parameters, then calculates the next state. We refer to a single evaluation of this function as


Simulation constants
<State$_t$, stochastic inputs, context>
Simulation specific pipeline
<state$_{t+1}$, context>
Iteration step constants
Stochastic inputs, intial simsToStart
Generate initial states or feedback incomplete runs
<state$_t$, noise, context>
Iteration step constants
Iteration Step
<state$_{i+1}$, context>
Feedback updated state or accumulate output of completed run
Accumulated output, simsToStart
Iteration step constants, Iteration manager constants
Simulation parameters, stochastic inputs
Simulation parameters, noise stream
Iteration step constants, Iteration manager constants
Iteration Manager
Accumulated outputs
Combine outputs
Output

(a) Iteration Step (b) Iteration Manager (c) Multi-Iteration Manager

Figure 1: Monte Carlo financial simulation framework.

an 'iteration step' and the evaluation of this state sequence over $0 \leq t \leq T$ as a 'single simulation run'. We define a 'full simulation' as the combined results of a large number of single simulation runs. The number of simulation runs is specified by the user of the system, or is automatically increased until the accuracy converges to some user-defined accuracy.

Our framework abstracts this process into four components:

Iteration Step. Algorithm specific hardware kernel responsible for advancing the state of a simulation by a single time-step.

Iteration Manager. Generic hardware component which is responsible for feeding into, and consuming data from, the iteration step component.

Multi-Iteration Manager. Generic hardware component which co-ordinates multiple iteration managers within a single FPGA.

Network Distributor . Generic software allowing the instantiation of the multi-iteration managers using multiple FPGA boards, distributed across multiple networked computers, and combining the results of a full simulation.

For the remainder of this section we will describe these components and show how they fit together to make a general purpose discrete-time random walk Monte Carlo framework based on FPGAs.

3.2 Iteration Step

In our framework, the 'iteration step' component is the container for implementing a simulation specific kernel. Figure 1a shows a graphical outline of this component. The iteration step component is the part of our framework which must be tailored for each specific simulation. In Section 4 we show specific examples of what an iteration step may involve, and how they can be implemented and optimised.

When implementing an iteration step kernel, pipelining is required to achieve good performance in hardware. We expect that iteration steps are a pure deterministic function of input state, simulation constants, and random inputs, with no data hazards, allowing the iteration step to be pipelined to and arbitrary degree.

In our framework we take advantage of pipelined step kernels, by filling alternate pipeline stages with independent simulations. For example, a pipeline with a latency of twelve cycles would ideally have twelve calculations from twelve different simulation in progress at once. Hiding the latency of the pipeline by executing multiple calculations is often called a C-Slow pipeline, and is only possible because the calculations are all independent, with no data hazards or ordering constraints. Filling a pipeline with multiple concurrent simulation runs allows us to keep all stages in the pipeline active, and thus extracting maximum parallelism from a hardware kernel.

At the end of a kernel pipeline, we identify to which simulation run each result belongs by flowing through an opaque context identifier. This is combined with other elements to make the per-cycle inputs to our iteration step in the form: $< \text{state}_t, \text{stochastic inputs}, \text{context} >$. Within this tuple, t is the iteration time-step, 'state' is algorithm specific data being iterated on, 'stochastic inputs' are random numbers appropriately distributed for the simulation, and 'context' is an input used by an 'iteration manager' (described below) to distinguish in-flight simulations in the pipeline. The per-cycle output of the iteration step components is of the form $< \text{state}_{t+1}, \text{context} >$.

As well as per-cycle inputs an iteration step may also need, for example, a set of constant inputs which represent interest rates or estimates of price volatility. The number and meaning of these constants are simulation step specific, but only need to be directly understood by the iteration step; all other components can treat them opaquely. These inputs will be fixed across a whole simulation, i.e. many hundreds or thousands of simulation runs, and so do not need to be included in the per-cycle simulation input.

3.3 Iteration Manager

The iteration step component for an algorithm by itself only transforms state_t to state_{t+1} . In order to provide initialisation, feedback and termination of separate simulation runs in an iteration step pipeline, we define a component called the 'Iteration Manager'. This is responsible for generating initial input states for an algorithm specific iteration step, advancing simulations in the pipeline over the range $0 \leq t \leq T$, and finally accumulating outputs from single simulation runs.

In order to allow maximum flexibility in the implementation of each iteration step implementation, we design our iteration manager to be independent of the specific latency of an iteration step. For this reason we require the 'context' field in the inputs and outputs of our iteration step. This technique removes the need to compute and take into account static timing information about our iteration step. Furthermore, it allows for more advanced optimisation techniques such as returning results out-of-order and taking many cycles for "difficult" calculations.

A structural illustration of our iteration manager is shown in Figure 1b. Our iteration manager is primarily composed of a loop, at the top of which is a process for selecting the next per-cycle input to the iteration step pipeline. When the iteration step is ready to accept a new input, the manager selects between continuing an existing simulation or starting a new one. If there is a feedback state: $\text{state}_t, t < T$ waiting, this is selected, otherwise a new initial statei: state_0 is created and a register, $simsToRun$, is decreased. The $simsToRun$ register is initially set to be the total number of simulation runs to perform, and when it eventually reaches zero, no more simulations will be started.

At the bottom of the iteration manager loop, the per-cycle outputs are examined. If the state is state_T, the final state in a simulation, the result will be accumulated as an output and nothing is fed-back allowing a new simulation to be started. If the state is some intermediate state, state_t, then output is not be affected and the state is fed-back into the pipeline.

Inputs to the iteration manager consist of an initial value for $simsToRun$, stochastic inputs, and iteration step specific constants. The only outputs consist of the current value of $simsToRun$, so we can determine when a full simulation is complete, and the accumulated results from all completed simulation runs. For our implementation, accumulation is simply a sum of the final state values for simulation runs. At the end of a full simulation, this sum can be used to generate an average by dividing it by the total number of simulation runs, the value of which is user-specified and hence statically known. The actual division operation is performed in software (described in Section 3.5) to save hardware resources. The register used in accumulation is wide enough to account for the maximum accumulatable value for the planned number of simulation runs.

3.4 Multi-Iteration Manager

A single iteration manager is only capable of advancing at most one simulation per clock cycle. This means that completing $simsToRun$ simulations of T steps will take at least $simsToRun \times T + k$ cycles. In most cases there will be enough area to implement two or more iteration managers in a single FPGA. If there is enough space to accommodate Z iteration managers, then we can reduce the number of cycles taken to:

$$\frac{simsToRun \times T + k}{Z} \tag{1}$$

Our 'multi-iteration manager' component, shown in Figure 1c, splits a request for $simsToRun$ simulations into batches of $simsToRun/Z$ and divides them between iteration managers. As the iteration managers update their output accumulators, these are then accumulated together in the same manner to provide an aggregate result. The external interface for the multi-iteration manager is identical to the standard iteration manager, except it needs to receive Z sets of stochastic inputs.

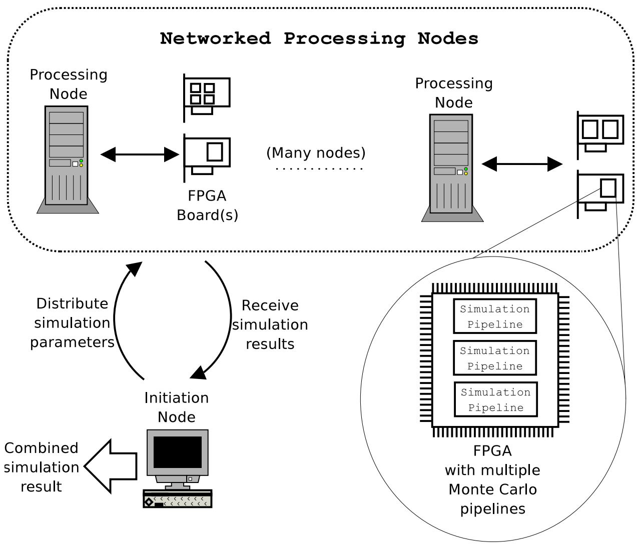

Networked Processing Nodes
Processing Node
FPGA Board(s)
(Many nodes)
Processing Node
Distribute simulation parameters
Receive simulation results
Simulation Pipeline
Simulation Pipeline
Simulation Pipeline
FPGA with multiple Monte Carlo pipelines
Initiation Node
Combined simulation result

Figure 2: Distributing simulations across multiple boards and computers.

3.5 Multi-FPGA Distribution

As well as extracting multiple levels of parallelism from our simulations in a single FPGA, we also consider the case where we have multiple FPGAs. Figure 2 shows our model for running simulations across multiple FPGA boards hosted in computer nodes attached to a network.

In our model a simulation is started from a single 'initiating' node. Software on this node activates other configured nodes on the network, causing them to upload FPGA bit-streams and starting simulations running with user specified parameters and run lengths. Nodes hosting the execution boards eventually receive the accumulated sums from all the local full simulations performed in FPGAs. These sums are then passed back to the initiating node which collects the total of the results from all machines. The initiating node then combines these to produce a final answer.

It is not a necessity that the execution environments for our simulation runs be homogeneous. Individual nodes in a network may have multiple FPGA boards with multiple FPGAs with in them. It is also possible to combine the results of software implementations of the same simulation with the results computed in FPGAs using this infrastructure.

The lack of a requirement for homogeneity in our proposed distribution model allows a smooth integration of FPGAs into an environment where simulations are needed. For example, in our prototype distribution system, we instantiate processes on networked hosts using remote execution over SSH. The application we develop to automate the distribution of a full simulation from an initiating node takes a list of hosts suitable for execution, and details of FPGA hardware if present. Hence any machine running an SSH daemon is capable of contributing to a simulation, but those with FPGAs contribute considerably more.

4 Simulation Algorithms

In this section we discuss two algorithms we have implemented to explore the benefits of our proposed financial framework: Vasicek [15] and Ho-Lee [16]. These are two examples of discrete-time random walk simulations, inspired by forms of analysis used in the finance industry. We implement both of these algorithms in our Monte Carlo framework by creating an 'Iteration Step' for each one using Celoxica's Handel-C tool [17]. Handel-C is also used for implementing the higher levels of our hardware framework. We also implement software versions of both algorithms which we hand vectorise with SIMD instructions targeting the Intel Xeon architecture. We compare the performance of software and FPGA implementations in Section 5.

4.1 Ho-Lee

Our first simulation kernel is the Ho-Lee model:

$$x_{t+1} = x_t + c_0 \times x_t + c_1 \times r_t \tag{2}$$

The simple nature of this equation makes it straight forward to implement in hardware requiring only multiplication and division units. For the same reason it is also efficiently performed in SIMD processors. We provide both fixed (signed, 16.16 precision) and floating-point (IEEE single-precision) versions of this simulation as iteration step kernels in our framework. For our fixed and floating point units, we use the implementations provided by Celoxica.

4.2 Vasicek

The second simulation we implement is the Vasicek model:

$$x_{t+1} = x_t + c_0 \times e^{(c_1 \times x_t)} + c_2 \times e^{(c_3 \times r_t)} \tag{3}$$

The Vasicek iteration step presents two main challenges. Firstly, the exponential function has only recently become feasible to evaluate in hardware, so there are no standard implementations or libraries to use. Secondly, the use of exponential presents range and accuracy problems within the evaluation of the formula. This is less of a problem when using a floating-point representations, but in fixed point it becomes difficult to balance the competing concerns of range and accuracy, particularly when the constants used in the Vasicek formula can affect both so much. For this reason we implement only a floating-point version of the Vasicek iteration step.

In order to deal with a lack of pre-existing standard libraries for computing the exponential function, we provide our own implementation. In our implementation we use range reduction by breaking a single exponential into the product of two exponentials which are easier to compute:

$$\begin{aligned} e^x &\Rightarrow e^{\lfloor kx \rfloor/k + (x - \lfloor kx \rfloor/k)} \\ &\Rightarrow \underbrace{e^{\lfloor kx \rfloor/k}}_{\text{Integer exponent}} \times \underbrace{e^{x-\lfloor kx \rfloor/k}}_{\text{Fractional exponent}} \end{aligned} \tag{4}$$

We use two different methods for computing the fractional and integer exponents. For the integer exponent we use a ROM of pre-computed values, and for the fractional part we use a degree 6 polynomial approximation.

For computation of exponent in our software Vasicek simulation, we use optimised math libraries provided by Intel [18].

4.3 Stochastic Input Generation

By the definition of the Monte Carlo method, we require a source of random number inputs for both our hardware and software simulations. For our hardware implementation, we use a random number generator previously developed in our research-lab based on the Box-Müller method [11]. For our software implementation we use an implementation based on the Wallace method [19]. We use the Wallace method in software as this has been shown to achieve high-performance in instruction processors.

5 Results

In evaluating the Vasicek and Ho-Lee simulations created within our framework, we primarily focus on performance gains achieved versus equivalent software simulations. For these comparisons we use a normalised measurement value: ‘steps/second’. We define a ‘step’ as a single iteration of a simulation kernel, for example a single evaluation of (3). So, for our fully pipelined iteration step kernels, this leads to one step being completed every cycle for each pipeline instantiated.

Our performance results for both vectorised and pure-C++ software implementations of the Ho-Lee and Vasicek simulations are shown in Table 1. We obtain our performance results by running the software on a workstation with a 2.66GHz Xeon processor.

Figures 3, 4 and 5 show performance results for our hardware simulations versus software. Figures 6, 7 and 8 show resource usages and clock-rates for our hardware implementations. As our framework allows repetition of simulation pipelines within an FPGA, the size of our target device has a strong influence on maximum speed-up. For this reason we present maximum speed-up results for a number of different FPGA devices with varying logic resource capacities. We consider all devices in the Xilinx Spartan-3 and Virtex-II series. However it should be noted that for this work we only use a maximum of 8 simulation pipelines in any one device due to tool-chain limitations.

Of our performance results, the highest speed-up over vectorised software we achieve is 8.29 times, seen in Figure 3 for 8 instances of a fixed-point Ho-Lee pipeline on a Spartan3. Interestingly, the speed-up for hardware floating-point Ho-Lee is more modest at only 3.6 times. The main reason for the limited speed-up in the floating-point implementation is the high-performance of the vectorised software Ho-Lee implementation. Thus the only way we can achieve a speed-up in hardware is through including more instances as is achievable in the fixed-point version. The floating-point version, however, consumes too many FPGA resources to allow such replication.

For our Vasicek hardware implementation, which is floating-point only, we manage to achieve a maximum speed-up of 7.1 times. While this is not as high as for our fixed-point Ho-Lee simulation, this speed-up is achieved with a smaller number of repetitions and using the same level of numeric precision as the software version. We attribute this to our hardware exponential function performing better than even the Intel optimised software equivalent. Indeed, if we compare our hardware Vasicek to our unoptimised software (using a standard `exp` implementation), we get our highest speed-up of 71.8 times.

Also in our Vasicek simulations, we see an example of a situation where increasing the number of repetitions actually causes a decrease in performance. This can be seen in Figure 5 our 6 Vasicek instances in VirtexII run slower than just using 5. Correspondingly in Figure 8 we see a drop in clock-speed combined with the largest area usage of all the designs. It seems that at this size, FPGA resources are too constrained to keep the clock-rate up.

For our fixed-point implementation we even can obtain a small increase in performance even with a very limited size Spartan3 device (xc3s400).

In our resource graphs we note that the designs use quite a significant number of the LUTs for shift-registers. For example in our Virtex-II, 8 times Ho-Lee instance design, 10.8% of LUTs are configured as shift registers. This use pattern arises due to heavily pipelined nature of this design.

A final point, not apparent in our graphs, is that one of the main areas for resource constraints in our designs is for hardwired block multipliers. In order to allow more designs in each device, we instruct our synthesis tools to only use multipliers in the random number generators.

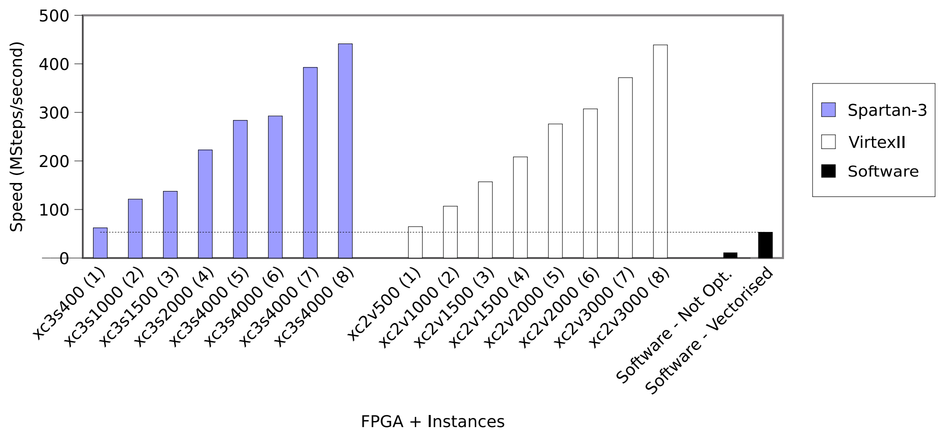


Figure 3: Ho-Lee fixed-point FPGA implementation performance.

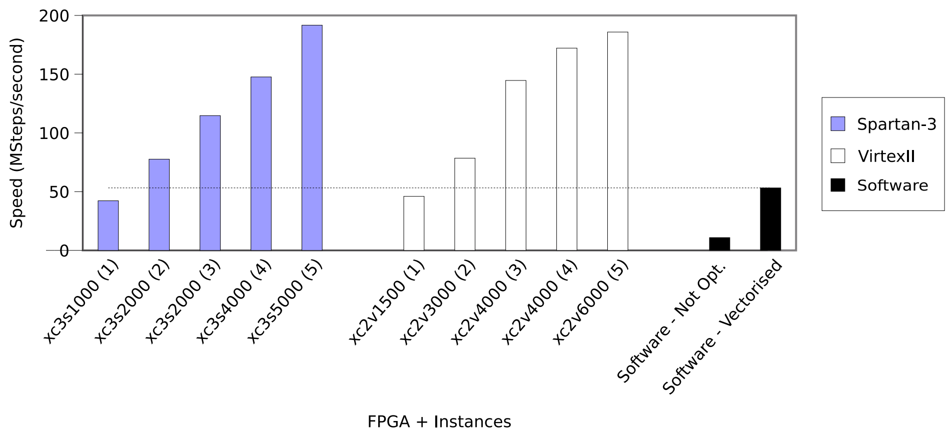


Figure 4: Ho-Lee floating-point FPGA implementation performance.

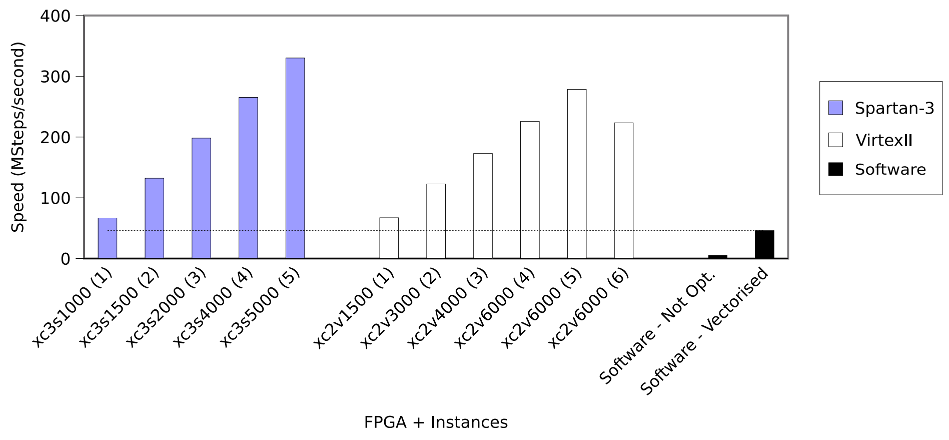


Figure 5: Vasicek floating-point FPGA implementation performance.

Algorithm	Vectorised	MSteps/second
Ho-Lee	No	10.9
Ho-Lee	Yes	53.2
Vasicek	No	4.6
Vasicek	Yes	46.0

Table 1: Performance of software only simulations running on a 2.66GHz Xeon processor.

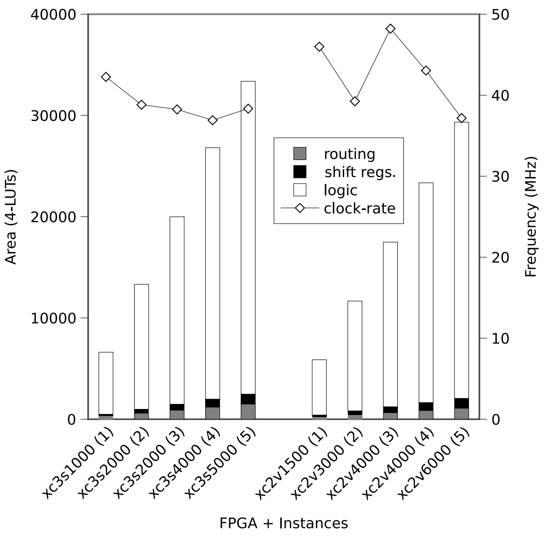


Figure 6: Resource usage and clock-rate of Ho-Lee floating-point hardware.

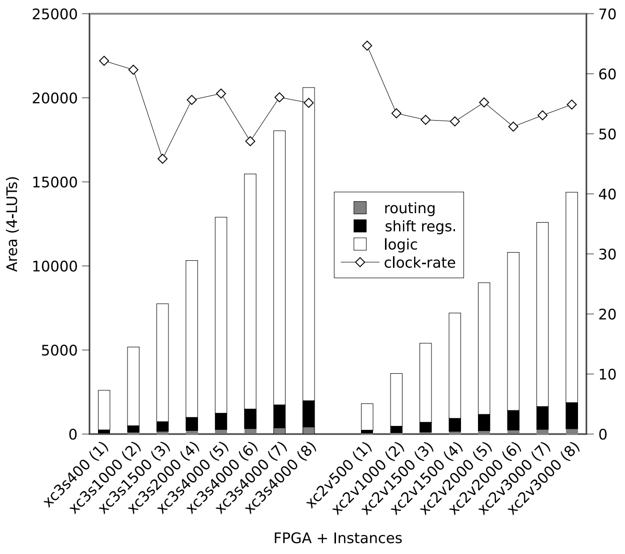


Figure 7: Resource usage and clock-rate of Ho-Lee fixed-point hardware.

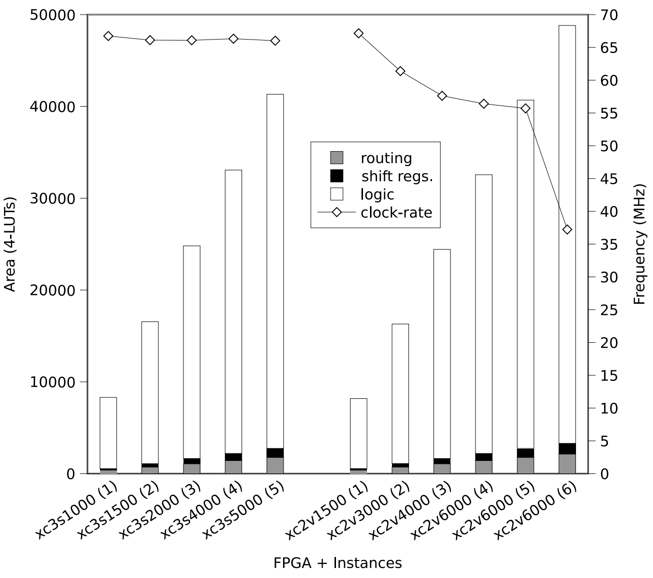


Figure 8: Resource usage and clock-rate of Vasicek floating-point hardware.

6 Conclusion

This paper demonstrates that FPGAs can profitably accelerate financial simulations. A general purpose framework is designed for implementing simulations of discrete-time random walks using the Monte Carlo method. Using this framework we implement hardware kernels for Ho-Lee and Vasicek simulations. We also create optimised software of the same simulations and compare the performance of these with that of our hardware. Through this comparison we find that we are able to accelerate Ho-Lee simulations 8.3 times compared to our optimised software using 8 fixed-point simulation instances in a Spartan3 FPGA. We also achieve a speed-up of 7.2 times for Vasicek simulations using single-precision floating-point in hardware. Furthermore, our hardware Vasicek implementation runs 71.8 times faster than a pure C++ software implementation. This speed-up over unoptimised software is also significant, since even industrial implementations do not always support SSE-based optimisations. For example parallelism in highly data dependent simulation kernels may be limited by the number of general purpose processors available, and may be difficult to accelerate using vector instructions. We show that with a hardware implementation we can run as many simulation instances as FPGA resources allow, and that C-Slow scheduling can be used to further parallelise a data-dependent simulation. We also describe how an FPGA accelerated simulation can be integrated into a heterogeneous distributed execution environment, comprising both FPGAs and software processors using commodity remote execution software.

While our framework helps to reduce the complexity of producing discrete-time random walk Monte Carlo simulations, we still need to find ways of automating the optimisation of algorithmic kernel cores. An example is to automatically generate optimised transcendental functions like e^x, which we have shown are needed in these kinds of algorithms. Our framework itself also has significant potential for generalisation to other classes of Monte Carlo simulations.

Acknowldedgements

The support of UK Engineering and Physical Sciences Research Council, Morgan Stanley, Celoxica and Xilinx is gratefully acknowledged.

References

[1] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *International symposium on Field programmable gate arrays*, pp. 171–180, ACM Press, 2004.

[2] J. Basney, R. Raman, and M. Livny, "High throughput Monte Carlo," in *SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[3] P. Wilmott, *Paul Wilmott Introduces Quantitative Finance*. Wiley, 2001.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992.

[5] R. Andraka and R. Phelps, "An FPGA based processor yields a real time high fidelity radar environment simulator," in *Conference on Military and Aerospace Applications of Programmable Devices and Technologies*, 1998.

[6] J. Chen, J. Moon, and K. Bazargan, "A reconfigurable FPGA-based readback signal generator for hard-drive read channel simulator," in *IEEE Design Automation Conference*, pp. 349–354, 2002.

[7] M. Yoshimi, Y. Osana, T. Fukushima, and H. Amano, "Stochastic simulation for biochemical reactions on FPGA," in *International Conference on Field Programmable Logic and Applications*, pp. 105–114, 2004.

[8] C. Cowen and S. Monaghan, "A reconfigurable Monte-Carlo clustering processor (MCCP)," in *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 59–65, 1994.

[9] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *International Conference on Field-Programmable Technology*, pp. 215–224, 2005.

[10] N. B. Liberati and F. Martini, "A multi-point distributed random variable accelerator for Monte Carlo simulation in finance," in *International Conference on Intelligent Systems Design and Applications*, pp. 532–537, 2005.

[11] D.-U. Lee, W. Luk, J. D. Villasenor, and P. Y. K. Cheung, "A gaussian noise generator for hardware-based simulations," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1523–1534, 2004.

[12] D. B. Thomas and W. Luk, "Efficient hardware generation of random variates with arbitrary distributions," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 2006.

[13] D. B. Thomas and W. Luk, "High quality uniform random number generation through LUT optimised linear recurrences," in *International Conference on Field-Programmable Technology*, IEEE Computer Society, 2005.

[14] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement C-slow retiming for the Xilinx Virtex FPGA," in *International symposium on Field programmable gate arrays*, pp. 185–194, ACM Press, 2003.

[15] O. Vasicek, "An equilibrium characterisation of the term structure," in *Journal of Financial Economics*, vol. 5, pp. 177–188, 1977.

[16] T. Ho and S. Lee, "Term structure movements and pricing interest rate contingent claims," in *Journal of Finance*, vol. 41, pp. 1011–1030, 1986.

[17] Celoxica, "Handel-C," *http://www.celoxica.com/*.

[18] *Intel Math Kernel Library*. Reference Manual.

[19] C. Wallace, "Fast pseudorandom generators for normal and exponential variate," in *ACM Transactions on Mathematical Software*, vol. 22, pp. 119–127, 1996.