FPGA ACCELERATION OF QUASI-MONTE CARLO IN FINANCE

Nathan A. Woods

XtremeData, Inc.
Schaumburg, IL, USA
email: nathan@xtremedatainc.com

Tom VanCourt

Altera Corporation
Santa Cruz, CA, USA
email: tvancour@altera.com

ABSTRACT

Today, quasi-Monte Carlo (QMC) methods are widely used in finance to price derivative securities. The QMC approach is popular because for many types of derivatives it yields an estimate of the price, to a given accuracy, faster than other competitive approaches, like Monte Carlo (MC) methods. The calculation of the large number of underlying asset pathways consumes a significant portion of the overall run-time and energy of modern QMC derivative pricing simulations. Therefore, we present an FPGA-based accelerator for the calculation of asset pathways suitable for use in the QMC pricing of several types of derivative securities. Although this implementation uses constructs (recursive algorithms and double-precision floating point) not normally associated with successful FPGA computing, we demonstrate performance in excess of 50× that of a 3 GHz multi-core processor.

1. INTRODUCTION

The price of many financial derivative securities can be expressed as intractable integrals of very high dimensionality [1]. For example, it is not unusual today for the risk-neutral price of an exotic derivative security to be a function of 20 or more underlying assets valued at 100 or more points in time, yielding an integral over 2,000+ dimensions. Often, millions of simulations are required to achieve an estimate of the price to the desired accuracy.

Generally for integration problems of such high dimensionality, Monte Carlo (MC) and the related Quasi-Monte Carlo (QMC) methods are the only practical solution. QMC methods have in recent years become the method of choice for the pricing of a large class of derivatives because of faster convergence compared to standard MC [2]. Notoriously computationally intensive, repetitive, and embarrassingly parallel, such simulations are an interesting candidate for hardware acceleration.

A large body of work has been compiled over the years demonstrating that field-programmable gate arrays (FPGAs) are fast and efficient devices for generating high-quality pseudo-random numbers (see for example [3] and [4]). Some authors have also investigated the hardware acceleration of applications that utilize pseudo-random numbers, such as MC simulation [6]. However, we know of no work that investigates the efficacy of FPGAs for accelerating the generation of quasi-random sequences, nor do we know of any studies that investigate FPGA acceleration of QMC simulations.

In this paper, we present an FPGA accelerator suitable for the pricing of a class of financial instruments by QMC simulation. Specifically, we focus on the calculation of quasi-random Brownian motion, which according to our measurements can consume as much as 80% of the run-time of a simple QMC options pricing simulation.

This paper is organized as follows. In section 2, we briefly review the QMC method. In section 3, we present a mathematical description of Brownian motion and present a simple algorithm for its calculation. This is followed in section 4 by a description of the hardware accelerator. We present hardware resource utilization and performance results in section 5. In section 6, we conclude with a discussion of potential improvements in the design and future work.

2. QUASI-MONTE CARLO SIMULATION

We begin with a short review of QMC methods. The following summary closely follows the introduction in [7].

Consider the MC estimation of an integral over the *s*-dimensional hypercube (with no loss of generality) of the form

$$\mu = \int_{[0,1)^s} f(\mathbf{u})\, d\mathbf{u}, \tag{1}$$

where $\mathbf{u} = [u_0, u_1, \ldots u_{s-1}]^T$ is a vector of uniform random numbers and f represents the transformation from that vector to the simulation output $f(\mathbf{u})$, assumed an unbiased estimator of μ. The form of the estimator of μ considered here is given by

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=0}^{n-1} f(\mathbf{u}_i), \tag{2}$$

where $P_n = [\mathbf{u}_0, \ldots \mathbf{u}_{n-1}] \subset [0,1)^S$ is the point set over which the average is taken, and the number of points n corresponds to the number of simulation runs. In MC, the $\mathbf{u}_i$ are independent and uniformly distributed over $[0,1)^S$

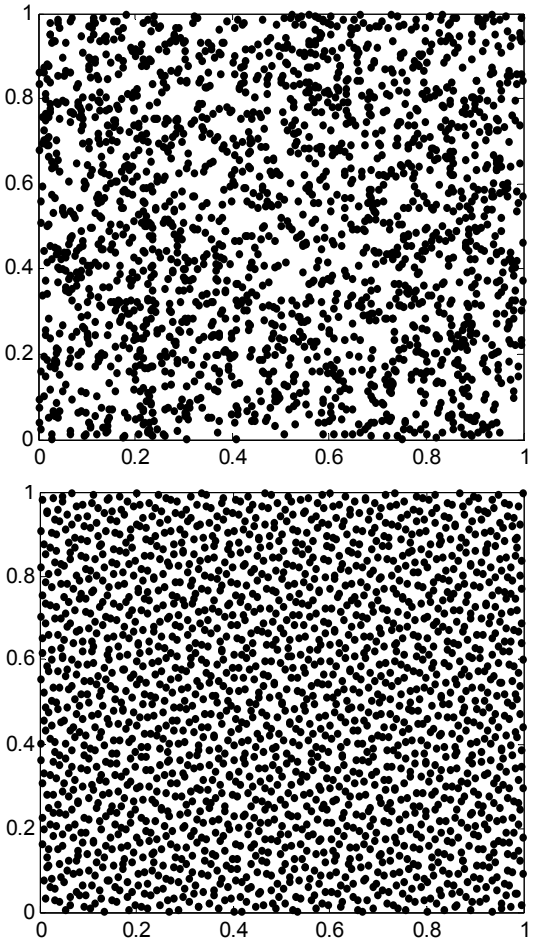

Fig. 1. First 2,048 points from MT19937 (top) and Sobol (bottom).

and produced in practice by pseudo-random number generators. Then, $\hat{\mu}_n$ is unbiased and has variance σ^2/n. If $\sigma^2 < \infty$, we have from the central-limit theorem that the size of a confidence interval for μ converges at a rate $O(\sigma n^{-1/2})$ independent of the number of dimensions, s.

Quasi-Monte Carlo (QMC) attempts to take the points $\mathbf{u}_i$ more uniformly distributed over $[0,1)^S$ than typical, independent, uniform random points. Loosely speaking, such generators are designed to minimize local deviation of point density from the global point density for all n, or *discrepancy* [2]. To illustrate this, Fig. 1 shows a 2-D projection of s-D points produced by a uniform pseudo-random number generator (MT19937) next to projections of the first 2 dimensions of a quasi-random number generator (Sobol). Clearly the quasi-random points occupy the space more uniformly than the pseudo-random points, which tend to produce clusters and gaps.

As a result of this improved uniformity, the asymptotic convergence of QMC is $O(n^{-1} \ln^S n)$, as determined by the Koksma-Hlawka inequality [1]. This is significantly better than the convergence rate of MC. However, this fact alone does not explain the superiority of QMC over MC in financial analytics for typical values of n and s [7]. We will return to this discussion in Section 4 when we introduce the Brownian Bridge.

3. BROWNIAN MOTION

The value of financial derivatives, such as options, depends on the future value of one of more underlying assets: the price of several stocks, for example. The evolution of the each asset's value is typically modeled as a separate stochastic process driven by Brownian motion [1]. A standard Brownian motion, $W(t)$, is a continuous function of t on the interval $[0,T]$ specified by:

$$W(0) = 0 \tag{3}$$

$$P\big(W(t), W(t+\Delta)\big) = P\big(W(t)\big)P\big(W(t+\Delta)\big), \Delta > 0 \tag{4}$$

$$W(t) - W(s) \sim N(0, t-s),\ 0 \le s < t \le T \tag{5}$$

For example, a standard Brownian motion can be calculated using the recurrence

$$W(t_{i+1}) = W(t_i) + \sqrt{t_{i+1} - t_i}\, z_{i+1}, \tag{6}$$

for i=0,1,…n-1, where t_i=iT/n, and $z \sim N(0,1)$. In this paper, we consider the hardware acceleration of the quasi-random generation of the Brownian motion. To do this, we use the following procedure for some number of assets n_a and time steps n_t, such that $s = n_a \times n_t$:

1. Generate a $s \times 1$ vector of uniform, quasi-random samples, **u**.
2. Transform each element u_i of **u** to a normal random variate, z_i, and form a vector **z**.
3. Generate a standard Brownian motion from **z**

Fig. 2 depicts these steps in a block diagram. The implementation of each block in Fig. 2 is described in the next section.

4. HARDWARE ACCELERATOR

Many quasi-random sequences have been devised, including the Halton, Sobol, Faure and Neideretter sequences to name just a few. We chose to implement the Sobol quasi-random sequence [8] because of its simplicity and demonstrated effectiveness in financial pricing simulations [1], [2], [9], [10].

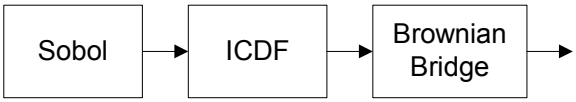

Fig. 2. Block diagram of quasi-random Brownian motion hardware accelerator.

To generate Gaussian variates from the Sobol points, we chose the inverse transform method. In this method, a uniform variate is transformed to a Gaussian variate by applying the inverse cumulative density function (or ICDF, defined in Section 4.2.) As far as we know, we are the first to consider this method in hardware. As [1] and [2] note, this method is the most desirable method for QMC simulation, for the following reasons:

- The inverse method, unlike all rejection methods, does not destroy the carefully crafted uniformity of the quasi-random sequence.
- The inverse method requires just one uniform input per normal output, and so does not increase the original dimensionality of the problem.
- The inverse transform is continuous and monotone, enhancing the effectiveness of variance reduction techniques.

Finally, we chose to implement the Brownian bridge algorithm to generate the Brownian path. The combination of quasi-random Sobol points and the Brownian bridge efficiently and effectively exploits the low *effective* dimensionality (see [7] for a definition) that many integrands found in financial analytics exhibit, and is the key to fast convergence of QMC in many cases [7], [10].

The remainder of this section describes the three blocks shown in Fig 2. Note that the pipeline structure allows parallel execution of all three stages of the computation.

4.1. Sobol Quasi-Random Number Generation

Let us define a vector **x** as an s-dimensional vector of binary words of w bits each. Then, the n^{th} element of the j^{th} dimension of the Sobol sequence, $x_n^{(j)}$, as specified by the Antonov-Saleev variation of the Sobol sequence [11] is specified by the recurrence

$$x_n^{(j)} = x_{n-1}^{(j)} \otimes v_k^{(j)}, \tag{7}$$

where $\otimes$ denotes bit-wise XOR, k is the index of the single bit that differs in the Gray code of n and n-1, and $v_k^{(j)}$ is the k^{th} direction vector in the j^{th} dimension. The direction vectors are generated by a linear recurrence over finite field $F_2 = \{0,1\}$

$$v_i^{(j)} = a_1 v_{i-1}^{(j)} \otimes a_2 v_{i-2}^{(j)} \otimes \mathrm{L}\ a_q v_{i-q}^{(j)} \otimes \left(v_{i-q}^{(j)} / 2^q \right). \tag{8}$$

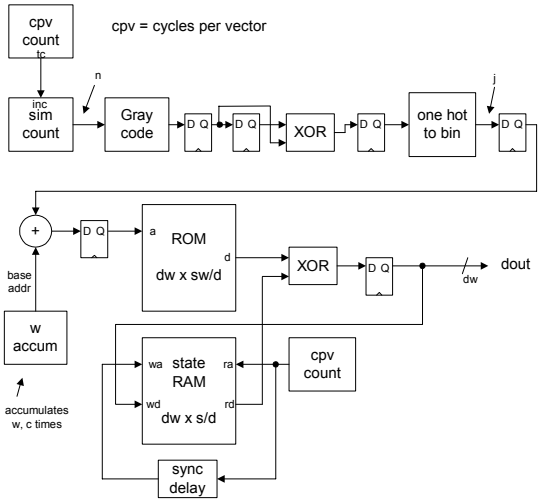

Fig. 3. Block diagram of Sobol sequence generator circuit**.**

The coefficients a_i, i=1,…q, are the coefficients of a degree q primitive polynomial over F_2, with $i > q$. Different primitive polynomials are used for each dimension. Space does not permit us to discuss the initialization of (8), an important consideration. In our implementation, the initialization is performed off-line, so any initialization method can be supported.

A block diagram of our circuit that generates the Sobol sequence is shown in Fig 3. The pipelined design permits the generation of a Sobol vector **u** with an arbitrary number of dimensions, s, set at compile-time and limited only by available on-chip memory resources. The bit-width of each vector component is also a compile-time parameter, w, but fixed to 32 in this accelerator. To control hardware resource consumption, the number of dimensions computed per cycle, d, is a compile-time parameter, permitting the generation of an s-dimensional vector over $c=s/d$ cycles. The circuit shown in Fig. 3 assumes that c is greater than the sum of the read and write latency of the state RAM. This is usually the case since s is typically large. If this is not the case, then the design implements the state RAM using flip-flops. Direction vectors are computed off-line and loaded into the ROM (loading mechanism omitted) before sequence generation commences.

4.2. Inverse Cumulative Normal Density Function

Transformation of uniform quasi-random numbers u_i to Gaussian random numbers z_i is accomplished directly using $z_i = \Phi^{-1}(u_i)$, where $\Phi^{-1}(x)$ is the inverse normal cumulative density function (ICDF). The goal is to evaluate the function $x(u) = \Phi^{-1}(u)$ defined by

$$\int_{-\infty}^{x(u)} \phi(t)\, dt = u, \tag{9}$$

where $\phi(t)=(2\pi)^{-1/2}\exp(-t^2/2)$. As no closed form solution to the integral in (9) is known, one must resort to approximations.

Two popular approximations for $\Phi^{-1}(x)$ used today in finance are the Acklam approximation [12] and the Moro approximation [13]. Both approximations employ rational polynomials, with Acklam's approximation the more accurate with a relative error $< 1.15\times10^{-9}$. Because our goal was to develop a single, general purpose accelerator, we opted for the more stringent error criterion, and set as a goal an approximation at least as accurate as Acklam's.

Our implementation exploits the knowledge that the function is fed by a 32-bit uniform quasi-random integer and is therefore valid for inputs in the interval $[2^{-32}, 1)$. Since the normal ICDF is anti-symmetric about 0.5, it suffices to consider the interval $[2^{-32}, 0.5]$. We split this interval into two disjoint regions, $R_c \subseteq [2^{-m}, 0.5)$, called the central region, and $R_t \subseteq [2^{-32}, 2^{-m})$, the tail region, and handle the input 0.5 as a special case.

We subdivide the central region into m octaves on power of two boundaries. Finally, we subdivide each octave into 2^r segments of equal length, so that the j^{th} segment of the i^{th} octave selects the region

$$R_c^{(i,j)} \subseteq [2^{-(i+2)} + j\Delta,\ 2^{-(i+2)} + (j+1)\Delta) \qquad (10)$$

where $\Delta=2^{-(i+2+r)}$, $i=0,\ldots,m-1$, and $j=0,\ldots,2^r-1$. For each segment, we fit a minimax polynomial of order k_c. For the tail region, we first transform the input x using

$$y = \ln\left(-\ln\left(x\right)\right) \qquad (11)$$

and then fit one minimax polynomial in y of order k_t. A block diagram of the circuit that computes the ICDF is shown in Fig. 4. This circuit approximates the ICDF with a relative error $< 10^{-10}$.

The compile-time parameters of the circuit that achieve this accuracy are $m = 11$, $r = 6$, $k_c = 3$, and $k_t = 7$. Except for range reduction, all computations are carried out internally using double-precision floating-point arithmetic.

The range reduction, central calculation, and result merge units are fully pipelined. The result merge unit guarantees that the ICDF outputs its results in the same order in which the inputs arrived. If this is not possible (e.g. because a tail calculation result is required but not available), the unit stalls. The central calculation unit can compute one k_c-order polynomial per cycle. To reduce hardware resources consumed, we exploit the fact that the tail calculation is required, on average, only once for every 2,048 cycles, and implement this calculation in a multi-cycle manner using one floating-point logarithm [14], adder, and multiplier. Furthermore, the ICDF design is multidimensional and can convert d separate, uniform inputs to d Gaussians in parallel, with d a compile-time parameter. We require only $\lceil d/2{,}048 \rceil$ tail computation units

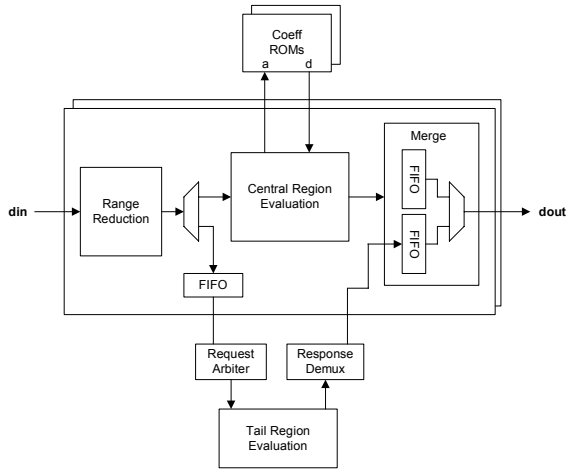

Fig. 4. Block diagram of ICDF circuit.

and $\lceil d/4 \rceil$ coefficient ROM units. In the former case, this is accomplished by arbitrating for the tail calculation unit, and in the latter case, this is accomplished by double-clocking a dual-port ROM for each of the k_c+1 coefficients. Double-clocking is optional at compile-time, and if not selected, $\lceil d/2 \rceil$ coefficient ROM units are required.

4.3. Brownian Bridge

The recurrence in (6) generates a Brownian path causally. However, we may generate the points along a Brownian path in any order by simply drawing the points from a distribution conditioned on the points already generated. Since the statistics of the path are Gaussian, the conditional mean and variance can be easily computed in closed form.

The Brownian Bridge (BB) is a divide-and-conquer algorithm that constructs a Brownian path by successively computing a point in the path between two previously computed endpoints [1]. The algorithm computes the end-point of the path first, then the mid-point of the path, followed by the mid-points of the two sub-paths to the left and to the right, and so on, generating all required intermediate points in an ever-refining manner. Somewhat surprisingly, this recursive decomposition is more amenable to FPGA acceleration than the incremental algorithm of (6), since it exposes parallelism that the FPGA can exploit.

Let us define a known point on the left, $W(t_i)$, and a known point on the right, $W(t_{i+1})$. Then, a point $W(t)$ with $t_i < t < t_{i+1}$ is given by

$$W(t) = aW\left(t_i\right) + \left(1-a\right)W\left(t_{i+1}\right) + bZ \qquad (12)$$

where $a = \dfrac{t_{i+1}-t}{t_{i+1}-t_i}$, $b = \sqrt{\dfrac{\left(t_{i+1}-t\right)\left(t-t_i\right)}{t_{i+1}-t_i}}$, and Z is a draw from a normal distribution with zero mean and unit variance. Each bridge calculation requires a new draw Z.

The coefficients a and b do not depend on Z or $W(t_i)$ and so can be stored in a table of at most n_t entries, where n_t is the number of equally spaced discrete time steps determined at compile time.

Pseudo-code describing the operation of the hardware BB is shown in Fig 5. The pseudo-code assumes a pipelined floating point unit (FP) computing midpoint values using (12) and a queue that holds intervals that have not yet been scheduled for processing in the FP unit. Although not shown in Fig. 5, this implementation does not enqueue or evaluate intervals a single discrete step in length. Fig. 6 illustrates the major hardware elements that implement the algorithm in Fig 5.

The FP pipeline is built using experimental tools [15] that generate RTL code for entire expressions at a time, rather than stringing together function blocks representing primitive operations. By fusing multiple operations, these tools exploit FP-specific optimizations that generate FP blocks with the following properties:

- Logic utilization and latency as low as 50% of block-based design, and
- One result per cycle throughput.

Reduced logic per pipeline frees logic resources that can be used to build more parallel pipeline units.

Initially, the FP pipeline is empty. The initial interval is sent to the FP pipeline. The BB circuit then idles until the midpoint value of the initial interval emerges from the FP unit. Then, two sub-intervals are available for computation: the left-mid and mid-right subintervals. These are sent to the FP unit on successive cycles, and the BB circuit idles until the FP latency elapses. Then, on two successive cycles, the left and right sub-interval midpoints emerge making four new sub-intervals available for computation, and so on. The FP pipeline fills completely in $\lceil \log_2 L \rceil$ generations of interval subdivision, where L represents the FP pipeline latency in cycles. Once filled, the controller generates one n_a-dimensional point per cycle until all points have been computed. The FP pipeline has fixed latency, so one controller can manage multiple pipelines. As a result, steady state operation generates n_a floating point values per cycle, limited only by the capacity of the FPGA fabric.

Until the pipeline fills, the BB controller evaluates interval midpoints strictly in decreasing order by interval size, consuming a new vector Z of n_a Gaussian variates for each midpoint. As (12) shows, longer intervals have larger b coefficients, so the earlier draws from the Gaussian variate generator have the larger effect on the overall shape of the path. To understand the significance of this, one must know two facts. First, the lower dimensions of the Sobol sequence, used to produce the earlier Gaussian draws, are more uniform (i.e. have lower discrepancy) than higher dimensions for finite n [2]. Second, the values of many derivative securities are determined more by the overall shape of the path than by its fine structure [1]. Thus, the combination of the Sobol sequence and the BB implements a strategy whereby the most uniform quasi-random points are applied to those dimensions which contribute the most to the variance of the estimate of the price [7]. (More complex queuing logic could force *all* intervals to be evaluated in decreasing order by length, but the additional complexity and hardware cost have not been justified.) This is another reason for superior convergence of QMC compared to MC in this implementation.

```
send initial interval to FP pipeline;
work_in_progress = TRUE;
while (work_in_progress)
   if (result available at FP pipeline) {
      report result's position and value;
      use result as new midpoint;
      send new left subinterval to FP pipeline;
      enqueue new right subinterval;
   } else if (queue is not empty) {
      remove one interval from queue;
      send that interval to FP pipeline;
   } else if (FP pipeline is empty)
      work_in_progress = FALSE;
```

Fig. 5. Pseudo-code for iterative construction of BB

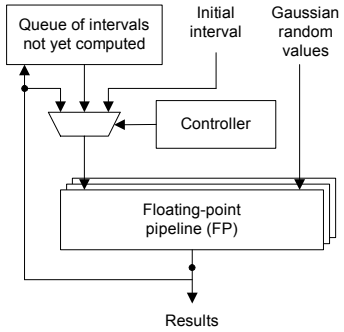

Fig. 6. Diagram of BB circuit with n_a FP pipelines

The BB generates points in non-sequential order, so a pathway should not be sent to a client application until it is completely computed. In practice, results would probably be double-buffered, so one pathway could be accessed by the application while another is being computed.

5. RESULTS

Our accelerator targeted the Altera Stratix III EP3SE260-3 FPGA. The accelerator was described using the VHDL hardware description language and synthesized, placed, and routed with Altera's Quartus II v7.2 software.

The various hardware resources consumed by the accelerator are shown in Table 1, for a system that handles n_t=512 time steps and n_a=8 assets in IEEE 754 double-precision floating point. The design consumes roughly 86% of the logic resources, 78% of the DSP blocks, 33% of the M9K memories, and 60% of the M144K memories in the FPGA. The maximum clock frequency achieved for the entire design was 110 MHz, but we anticipate a 2x

improvement in F_{max} with further optimizations. We compare the performance of our hardware accelerator to the performance of a C++ implementation provided by a major investment bank, running on a modern x86 processor, an Intel dual-core Xeon Woodcrest running at 3 GHz. We measured the execution time to produce n_a=8 Brownian pathways of n_t=512 time steps each. The theoretical execution time of the hardware accelerator was determined from cycle-accurate simulation combined with post-place and route timing results. The results are shown in Table 2. The performance ratio is defined as the execution time of software divided by the execution time of the hardware accelerator for the same task. We achieve a performance improvement over 50×, and we anticipate higher performance on longer bridges (projected to 70× for n_t=1024), because of the lower relative contribution of startup time to total time for computing the bridge. Since all Brownian pathways are independent, it is possible to increase FPGA performance linearly by simply increasing n_a to exploit the resources on larger FPGAs.

Table 1. Hardware Accelerator Resource Usage.

Unit	ALM	FF	M9K	M144K	DSP blk
Sobol	315	464	15	29	0
ICDF	51,584	72,270	211	0	47
BB	36,066	55,896	58	0	28
Total	87,965	128,630	284	29	75

Table 2. Performance Comparison Results.

Time steps	CPU (μs)	FPGA (μs)	Perf. Ratio
512	383	6.7	57×

6. CONCLUSIONS

We present a hardware accelerator that generates multiple, quasi-random, standard Brownian motions suitable for the acceleration of quasi-Monte Carlo simulation of financial derivatives. We are not aware of previous FPGA implementations of the Sobol quasi-random generator, the Gaussian ICDF, or the Brownian Bridge algorithm, so we believe these to be novel uses of FPGA computation.

The accelerator achieves a speedup of over 50× compared to a single thread running on a modern multi-core x86 processor for this task. We know of additional optimizations that could increase this speedup significantly, by improving the clock rate of the FP pipeline and by reducing its latency (therefore reducing idle cycles during startup). Furthermore, we expect the performance of the accelerator to increase linearly as FPGA devices increase in density, due to increases in the number of asset values n_a computed in each cycle, whereas our preliminary measurements suggest that one can expect sub-linear increases in performance as the number of cores increases on an x86 processor. Therefore, it is reasonable to expect the performance advantage of FPGAs over multi-core CPUs to increase in the future for this task.

REFERENCES

[1] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer, 2004.

[2] P. Jackel, *Monte Carlo Methods in Finance*, Wiley, 2002.

[3] I. Dalal and D. Stefan, "A hardware framework for the fast generation of multiple long-period random number streams," *Proc. ACM/SIGDA 18th Int. Symp. Field-Programmable Gate Arrays*, pp. 245-254, Feb. 2008.

[4] D. Thomas and W. Luk, "FPGA-optimised high-quality random number generators," *Proc. ACM/SIGDA 18th Int. Symp. Field-Programmable Gate Arrays*, pp. 235-244, Feb. 2008.

[5] D. Lee et al., "A Gaussian noise generator for hardware-based simulations," *IEEE Trans. on Comp.*, vol. 53, no. 12, Dec. 2004.

[6] M. Gokhale et al., "Monte Carlo radiative heat transfer simulation," in *Proc. of Field Programmable Logic and Applications*, Springer, pp. 95-104, 2004.

[7] P. L'Ecuyer, "Quasi-Monte Carlo Methods in Finance," *Proc. of Winter Simulation Conf.*, pp. 1645-1655, 2004.

[8] I. M. Sobol', "Uniformly distributed sequences with an additional uniform propoerty," *USSR Journal of Computational Mathematics and Mathematical Physics*, vol 16, pp. 1332-1337, 1976.

[9] P. Ackworth et. al, "A comparison of some Monte Carlo and quasi Monte Carlo methods for options pricing," *Monte Carlo and Quasi-Monte Carlo Methods*, P. Hellekalek, G. Larcher, H. Niederreiter, P. Zinterhof, eds., Springer-Verlag, Berlin, pp. 1-18, 1996.

[10] R. Calfish et al, "Valuation of mortgage-backed securities using Brownian bridges to reduce effective dimenstion," *Journal of Computational Finance*, 1997, 1:27-46.

[11] I. A. Antonov and V. M. Saleev, "An econominal method of computing LP tau-sqeuances," *USSR Journal of Computational Mathematics and Mathematical Physics*, vol. 19, no. 1, pp. 252-256, 1980.

[12] P. J. Acklam, "An algorithm for computing the inverse normal cumulative distribution function," University of Oslo, Statistics Division, June 2000.

[13] B. Moro, "The full monte," *Risk* 8(Feb), pp. 57-58, 1995.

[14] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for FPGAs," *Conf. Rec. of the 39th Asilomar Conf. on Signals, Syst., and Comp.*, Oct. 2005, pp.1186-1190.

[15] M. Langhammer, "Floating-point Datapath Synthesis for FPGAs," *Proc. Of 2008 Int. Conf. on Field Programmable Logic and Applications*, submitted.