

Financial Monte Carlo Simulation on Architecturally Diverse Systems

**Naveen Singla
Michael Hall
Berkley Shands
Roger D. Chamberlain**

Naveen Singla, Michael Hall, Berkley Shands, and Roger D. Chamberlain, "Financial Monte Carlo Simulation on Architecturally Diverse Systems," in *Proc. of Workshop on High Performance Computational Finance*, November 2008 (associated with SuperComputing'08).

Exegy, Inc.

and

Dept. of Computer Science and Engineering
Washington University in St. Louis

Financial Monte Carlo Simulation on Architecturally Diverse Systems

Naveen Singla*, Michael Hall[†], Berkley Shands[†], and Roger D. Chamberlain*[†]

*Exegy, Inc., St. Louis, Missouri

[†]Dept. of Computer Science and Engineering, Washington University in St. Louis
nsingla@exegy.com, {mhall24,berkley,roger}@wustl.edu

Abstract

Computational finance relies heavily on the use of Monte Carlo simulation techniques. However, Monte Carlo simulation is computationally very demanding. We demonstrate the use of architecturally diverse systems to accelerate the performance of these simulations, exploiting both graphics processing units and field-programmable gate arrays. Performance results include a speedup of $74\times$ relative to an 8 core multiprocessor system ($180\times$ relative to a single processor core).

1. Introduction

Monte Carlo simulation finds frequent application in computational finance, most notably for options pricing and risk assessment [1]. Generally, Monte Carlo simulation is used to solve problems that are not well suited for direct solution, e.g., integration over very high-dimensional spaces. While the Monte Carlo simulation approach is widely applicable to many problems, it has the drawback that it is computationally very demanding, with a relatively slow convergence rate. As a result, acceleration of Monte Carlo simulation has received a significant amount of attention.

Since individual trials of a Monte Carlo simulation are independent of one another, one approach to acceleration is via parallel execution of the simulation model on multiple processors. We use a parallel implementation executing on 8 processor cores (four dual-core AMD Opterons) as our baseline system for performance comparison purposes.

A complementary approach for accelerating the Monte Carlo simulation is the use of architecturally diverse computers. Architecturally diverse computers integrate more than one type of computing resource into the system, such as field-programmable gate arrays (FPGAs) and/or graphics processing units (GPUs). The non-traditional computing resources are often referred to as co-processors, since they

frequently cooperate with the traditional processor cores (CPUs) in the solution of the complete problem.

In this paper, we will explore the use of both GPUs and FPGAs as co-processors in an architecturally diverse system [2]. The problem addressed is the computation of value at risk for a portfolio of financial instruments. To our knowledge, this is the first reporting of the use of both GPUs and FPGAs in the acceleration of an individual application.

2. Financial Monte Carlo Simulation

The application that we focus on is the calculation of the value at risk (VAR). The VAR is an indicator of the risk associated with a portfolio of financial instruments. It is defined as the maximum loss that is not exceeded with a given probability over a specified period of time. The probability is specified as a confidence level. The two confidence levels frequently used in practice are 95% and 99%. For example, a VAR of \$10,000 at 95% confidence level indicates that the probability that the losses will exceed \$10,000 is less than 0.05.

The VAR is calculated by estimating the value of the portfolio at the end of the specified time period. Since the underlying models for pricing financial instruments are driven by stochastic processes, at the end of the time period we obtain a distribution for the value of the portfolio. Given the desired confidence level the VAR can be calculated by inverting the cumulative distribution function of this distribution. We use the standard Black-Scholes model for the dynamics of the price of the financial instruments, namely stocks [1]. The Black-Scholes model assumes that the price of a stock is driven by a Brownian motion. Under this assumption the distribution of the stock price at a particular time in the future can be specified by knowledge of the current stock price, the volatility of the stock, any dividends on the stock, and a drift rate. It is well-known that under the Black-Scholes model the stock price follows a log-normal distribution therefore the VAR for a portfolio consisting of a single stock can

be computed in closed form (utilizing the Gaussian cumulative distribution function). However, when the portfolio consists of multiple stocks that are correlated the closed form expression is very difficult to obtain. This problem is exacerbated by considering instruments for which the payoff is a nonlinear function of the price (e.g. options and futures). For a portfolio consisting of K securities, a direct computation of the VAR would involve calculating a K -dimensional integral. Using conventional numerical methods the convergence rate in the error of the computation is $O(n^{-2/K})$, where n is the number of samples, which can be prohibitively slow when K is large. For Monte Carlo methods the error converges as $O(n^{-1/2})$ regardless of the dimensionality of the problem making it an attractive approach for the computation of the VAR.

The Monte Carlo approach to VAR calculation involves simulation of the value of the portfolio at the end of the time period. The differences between the value of the current portfolio and the simulated future portfolios provide estimates of the profit and loss (P&L) over the time period. The VAR then is simply the appropriate value of the sorted P&L estimates. For example, assuming that the portfolio is simulated n times, the VAR for a 95% confidence level is the value in the sorted P&L estimates array indexed by the greatest integer smaller than or equal to $0.05n$. To simulate the values of the components of a portfolio under the Black-Scholes model we need to generate correlated Gaussian random numbers and propagate them forward under the model. The VAR can then be calculated as described earlier. Figure 1 shows the functional pipeline for this simulation.

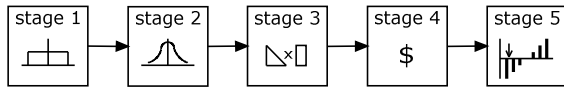


Figure 1. Computation pipeline for financial Monte Carlo simulation.

The pipeline stages are as follows:

- Stage 1: Uniform pseudo-random number generation – the Mersenne Twister [3] is used to generate random numbers that are uniformly distributed between 0 and $\text{MAXINT} (2^{32} - 1)$.
- Stage 2: The uniformly distributed random numbers are transformed into a Gaussian (normal) distribution with $\mu = 0$ and $\sigma^2 = 1$.
- Stage 3: The vector of independent normally distributed random numbers is transformed into a vector of correlated random numbers. This is accomplished by multiplying the vector by a lower triangular matrix. This lower triangular

matrix is obtained by the Cholesky factorization of the specified correlation matrix.

- Stage 4: The correlated Gaussian random numbers are used to generate random walks according to the Black-Scholes model. The values of the portfolio and the P&L values are also calculated in this stage.
- Stage 5: The P&L values are aggregated and sorted to obtain the VAR.

3. Accelerating the Simulation

In the previous section, the financial Monte Carlo simulation application is described as a functional pipeline. Here, we describe a collection of approaches to accelerate the performance of the pipeline relative to a serial implementation on a single processor. First, individual pipeline stages can be executed concurrently on different computational resources. For example, if stage 1 is executed on an FPGA, stages 2 to 4 on a GPU, and stage 5 on a CPU, the throughput can be greater than if all 5 stages are executed on the CPU.

With the exception of stage 5, each of the pipeline stages can be executed in parallel by straightforward replication. As an example, consider Figure 2, which illustrates parallel copies of stages 2 through 4. This can be implemented on the GPU through its standard threading mechanisms. Alternatively, multiple copies of stages 1 to 4 can be executed both on the GPU and the CPU.

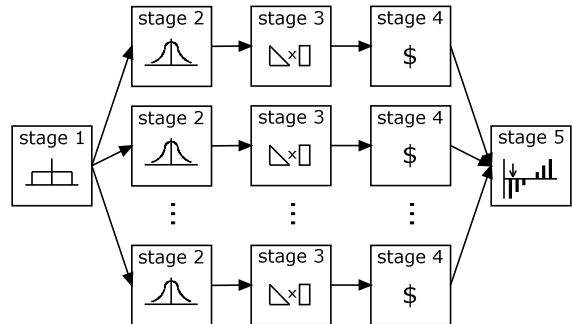


Figure 2. Monte Carlo simulation with parallel instances of stages 2 through 4.

Finally, parallelism can be exploited within the internals of a pipeline stage. Section 4.2 below describes FPGA implementations of stages 1 and 2 that exploit this type of parallelism. In general, the form of parallelization that is appropriate depends upon the computational platform that will be executing the stage. In what follows, we will describe and measure the performance implications of several such decisions.

4. Design Description

For the performance results presented below, we deployed each of the stages on a number of different computing platforms. We will describe the resulting set of systems in a bottom up fashion, first describing the individual stage designs followed by the integration into a complete system.

4.1. Stages 1 to 4 Deployed on a GPU

NVIDIA’s compute unified device architecture (CUDA) abstracts out the minute details of the GPU’s architecture and allows users to define kernels that execute on parallel threads [4]. The computation is divided into a grid of thread blocks where the grid and blocks can have up to three dimensions. The number of threads that can be launched simultaneously is limited by the architecture of the GPU. All the threads have access to a global memory on the GPU. In addition, the threads within a block are assigned a shared memory, and each thread is also assigned a local memory. The stages of the Monte Carlo simulation are implemented in this framework and, wherever necessary, are molded to fully exploit the available resources.

Stage 1, the generation of uniformly distributed random numbers, is implemented using the Mersenne Twister (MT). The iterative nature of the MT algorithm limits its parallelizability. The simplest alternative is to parallelize by replication: implement multiple instantiations of the same MT executing in parallel. In this case care must be taken to seed the MTs appropriately to prevent generating correlated random sequences. In contrast, Matsumoto and Nishimura proposed a scheme [5] that creates independent MTs for each parallel instantiation according to user-specified parameters. Primary among these parameters is an identifier that is unique to each parallel process (such as an identifier of the thread on the GPU) that is encoded in the characteristic polynomial of the MT. This ensures that the pseudorandom sequences generated by the MTs will be (fairly) uncorrelated.

NVIDIA’s CUDA package provides an implementation of parallel MTs based on Matsumoto and Nishimura’s dynamic creation algorithm. We used this implementation for generating 32-bit uniformly distributed random numbers. The state of a MT is described by a length N vector of words. The k -th word in the state vector is updated by using the k , $k+1$, and $k+M$ elements of the state vector and a twist matrix. The twist matrix is what allows the MT to achieve its equidistribution property in high dimensions. The uniform random numbers are generated

by a series of bit-shift and bit-masking operations on the state vector. It turns out that under these conditions the period of the random number generator is a Mersenne prime specified by the parameters of the MT. The most commonly employed example is MT19937 which uses a length 624 vector of 32-bit words to generate uniform random numbers between 0 and MAXINT with a period of $2^{19937} - 1$. Further details of the MT can be found in [3].

For the GPU implementation we choose MTs with smaller state vectors ($N = 19$) to reduce the computational load per thread. For a portfolio of K instruments we use K parallel MTs, as long as K is not too large. All the parameters for the K MTs are identical except the twist matrix and the bitmasks. The MTs still generate 32-bit words. The period of the MTs so obtained is $2^{607} - 1$. The uniform random numbers generated by the parallel MTs are transformed to the unit interval and stored in the global memory. Also, each MT generates blocks of B random numbers where B is chosen to maintain a balance between computation time of the downstream stages and the memory required to store the random numbers.

The transformation of uniform random numbers to numbers with the standard Gaussian distribution is implemented using the inverse cumulative distribution function (CDF) method: given a uniform random number u in the unit interval and a CDF $F(x)$, the numbers $F^{-1}(u)$ are distributed according to $F(x)$. Since the inverse CDF for the Gaussian distribution has no known analytical form, approximations to the inverse CDF are used in practice. We used Acklam’s approximation to the inverse CDF of the standard Gaussian [6]. This approximation uses a ratio of polynomials to approximate the lower and upper tails of the Gaussian CDF and another ratio of polynomials to approximate the region in between. Generally, inverse CDF methods are not as fast as acceptance/rejection methods: however, they allow us to balance the computational pipeline. We launch the maximum allowable threads on the GPU with each thread transforming one uniform random number to a Gaussian random number. Recall that stage 1 outputs $K \cdot B$ uniformly distributed random numbers to stage 2.

As mentioned earlier, transforming a vector (or in this case B vectors) of uncorrelated Gaussian random numbers into correlated ones involves multiplication by a lower triangular matrix. Given that we know the correlation matrix, this lower triangular matrix can be computed offline. In the implementation of this multiplication we did not exploit the lower triangular structure of the matrix. We used the BLAS (Basic Linear-Algebra Subrou-

times) implementation in CUDA to implement this matrix-matrix multiplication. For our implementation the correlation matrix is a $K \times K$ matrix and the Gaussian random numbers are arranged in a matrix with K rows and B columns.

Stage 4 involves generation of the random walks using the Black-Scholes model. This computation is as follows:

$$S(T) = S(0) \exp\left((r - \sigma^2 / 2)T + \sigma\sqrt{T}z\right); \quad (1)$$

where $S(t)$ is the price of the stock at time t ; T is the time period over which we are calculating the VAR; r is term incorporating the risk-free interest rate and any dividends on the stock; σ is the volatility of the stock; and z is a correlated Gaussian random number generated by stages 1 through 3. The risk-free interest rate is the rate of return on a riskless asset (such as a government issued bond). This computation is implemented on the maximum possible independent threads on the GPU each thread using the result of one of the threads from stage 3. This computation gives $K \cdot B$ prices that are then summed up to obtain B portfolio values followed by the P&L values.

Stage 4 outputs the B P&L values to stage 5 and this process is repeated until the desired number n of simulations are completed.

4.2. Stages 1 and 2 Deployed on an FPGA

When deploying a computational task onto an FPGA, the application developer's job is not simply to code an algorithm, but he/she is required to design hardware. Here, we describe the hardware design for the Mersenne Twister pseudo-random number generator used to implement stage 1 of the Monte Carlo simulation application. It is based in significant part on the hardware framework presented by Dalal and Stefan [7]. This is followed by the description of the hardware design for the zigurat

algorithm [8] used to implement stage 2, based on the framework presented by Zhang et al. [9].

As described above, there are several variants of the Mersenne Twister (MT). The variant we implemented on the FPGA was the MT19937, which requires 624 words of internal state and has a period of $2^{19937} - 1$. In our hardware design, the MT is parallelized by grouping the 624 words into N -word vectors which can be accessed in parallel across multiple memory banks. The design was simplified by restricting the value of N to a factor of 624 so that there are an integer number of N -word vectors. The N -word vectors are shown in Figure 3.

The FPGA has dual-ported block RAMs that can do simultaneous read and writes. As mentioned in the previous section, the algorithm for the Mersenne Twister requires the k , $k+1$, and $k+M$ elements of the state vector where k is the variable used to iterate over the state vector and M is a constant offset. These elements are used to compute the next state. For the near recurrence elements k and $k+1$, buffering can be used so that only one read is required for both elements. Another read will be necessary to get the $k+M$ element. Once the next state has been computed, one write will be required to update the state vector. These three operations will therefore take two clock cycles.

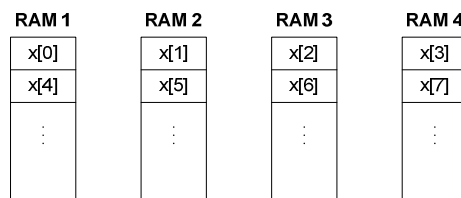


Figure 3. N -word vectors across multiple memory banks.

Figure 4 shows the architecture of the Mersenne Twister. In this design, reading and writing to block RAM is timed using a D flip-flop and the addresses supplied are given by the two counters. The block

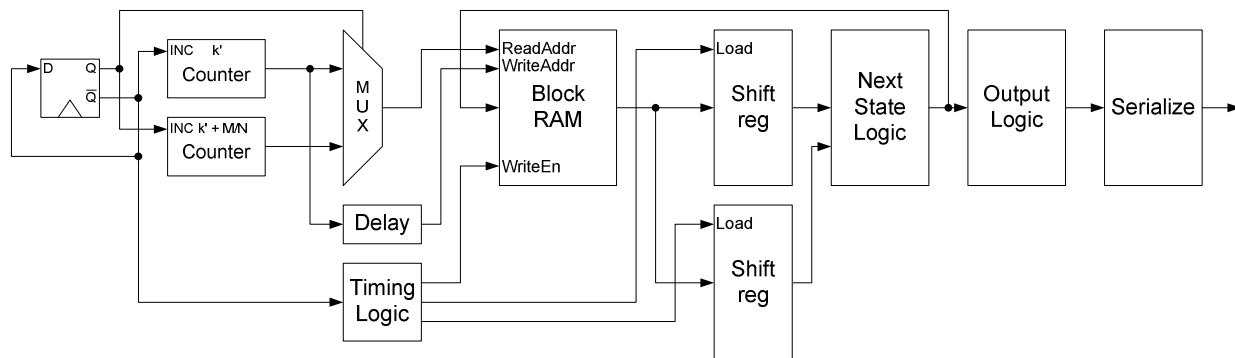


Figure 4. Architecture of Mersenne Twister uniform pseudo-random number generator.

RAM consists of N -word vectors where N is the amount of parallelism in the MT. The read address alternates between the current N -word vector, k' , and an offset of the vector, $k'+M/N$. The write address is simply a delayed value of the read address. The data from the block RAM is then loaded into the shift registers one at a time. Once both shift registers have been loaded, the next state is computed and written back into block RAM. The resulting next state is also used to compute the output set of generated random numbers which are serialized in the last step before leaving the MT. The serialization step loads the output random numbers into a shift register every two clock cycles and writes out half of the random numbers at every clock cycle.

In the design described above, parallelism is present in a number of places. First, the execution of the two counters, the memory accesses, the Next State Logic block, the Output Logic block, and the Serialize block are all concurrent with one another. Second, the functions performed by the various logic blocks are significantly more complex than that supported by individual instructions on a traditional processor. As a result, an individual execution of a logic block is often comparable to a sequence of processor instructions.

Stage 2, the transformation of the above generated, uniformly distributed random numbers into Gaussian (normal) distributed random numbers, is accomplished in the FPGA through the use of the ziggurat algorithm.

The ziggurat algorithm works by partitioning the positive half of the normal PDF into horizontal rectangular regions of equal area as illustrated in Figure 5. Most of the area of an individual rectangle exists entirely under the normal curve. The part which does not is called the wedge region. In the special case of the bottommost rectangle, it is called the tail region since it contains the tail of the normal distribution and is not finitely bounded. To generate a random number output, one of the rectangular regions is selected uniformly at random, the input uniform random number is then scaled by the dimensions of the rectangle and tested to determine whether or not it falls within the wedge region. If it falls outside the wedge region, it can be immediately accepted. If it is in the wedge region, accept or reject decisions are made based on the area of the curve defining the boundary of the wedge. If it is in the tail region, then an alternative (curve fitting) calculation is performed to generate a random number in the tail of the distribution.

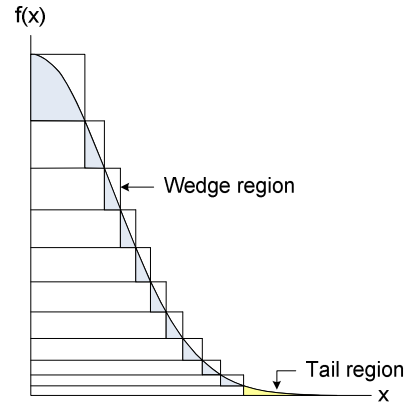


Figure 5. Ziggurat partitioning of the Gaussian (normal) PDF.

The architecture of the Gaussian transformation is illustrated in Figure 6. In the first step, a uniform random number is consumed at the input and a subset of the number is used as the index to a lookup table stored in read-only memory (ROM). The read-only memory returns two values: a scaling factor, x_i and a constant, k_i . The scaling factor will be used to compute the next Gaussian random number x by multiplying the uniform random number (from 0 to 1) by the scaling factor, x_i such that $0 \leq x < x_i$. The constant will be used to test whether or not the uniform random number falls within a wedge or tail region. If it is in one of these regions, then the index, i , and value x are written to FIFO2 for further processing. Otherwise, x is written to FIFO1 and is accepted immediately as a Gaussian random number.

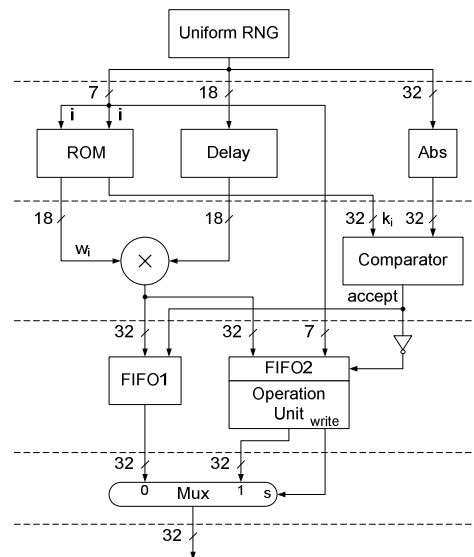


Figure 6. Architecture of the Gaussian transformation.

The operation unit consists of basic hardware components which are controlled by a finite state machine. The bulk of these computations are Taylor series expansions of the necessary functions. Since events that occur within the wedge and tail regions are rare, these calculations can take place over many clock cycles using simple hardware. The use of a FIFO before the operation unit allows several requests to queue up while the operation unit is active. This allows a nearly continuous stream of Gaussian random numbers to be concurrently produced by the main pipeline in Figure 6.

4.3. Integration into Complete Application

In this paper, we explore the performance achieved with several distinct mappings. The first is a baseline mapping, in which all 5 stages are executed on a single CPU. The uniformly distributed random numbers are generated using the MT19937 algorithm. Acklam’s approximation to the inverse of the Gaussian CDF is used to convert these uniformly distributed random numbers into Gaussian distributed random numbers. The BLAS library is used to implement the matrix-vector multiplication. This is followed by the random walks and the aggregation and sorting steps.

We also implemented the pipeline on multi-core processors, with stages 1 to 4 being replicated 8 times (one copy per processor core). This mapping is typical of multiprocessor implementations. Standard shared-memory threading (via `pthread`s) is used to coordinate between the replicated copies of stages 1 to 4 and the single copy of stage 5. In this multiprocessor implementation the matrix-vector multiplication is not implemented in BLAS but using a hand-coded loop (which outperformed the BLAS routines on 8 processors).

The third mapping is a typical use of the GPU as an acceleration engine. Here, stages 1 to 4 are mapped to the GPU and stage 5 is mapped to one of the processor cores (as described in Section 4.1). NVIDIA’s CUDA library is used to manage the data flow from the individual copies of stages 1 to 4 into the processor’s memory for stage 5. As mentioned in Section 4.1, the GPU outputs the P&L values to the processor. The processor aggregates the values into a single array and sorts it to obtain the VAR.

The fourth mapping attempts to exploit both the GPU and the FPGA as accelerators. Here, stage 1 is mapped to the FPGA, stages 2 to 4 are mapped to the GPU, and stage 5 is mapped to a processor core. The FPGA implementation of stage 1 was built within the Auto-Pipe infrastructure [10], which utilizes Exegy’s DMA engine [11]. Large buffers (~8 MB) in the

processors’ main memory subsystem are used to receive the data stream from the FPGA. These buffers are passed to the CUDA library for delivery to the GPU. In effect, data moving from FPGA to GPU is buffered temporarily in the main system memory.

The fifth mapping exploits the fact that stages 1 through 4 can all be executed in parallel, executing stages 1 and 2 on both the GPU and the FPGA and executing stages 3 and 4 on the CPUs and the GPU.

The 5 mappings are summarized in Table 1, which indicates the computational resource utilized in each stage for each mapping.

Table 1. Mapping summary. C is the CPU, nC is n CPUs, F is the FPGA, and G is the GPU.

Map	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	C	C	C	C	C
2	8C	8C	8C	8C	C
3	G	G	G	G	C
4	F	F	G	G	C
5	F,G	F,G	7C,G	7C,G	C

5. Performance Results

We consider a portfolio of 1024 stocks and nominal values for the parameters in the Black-Scholes model. These parameters are uniquely available from the data stream provided by an Exegy XTP ticker plant. 1024 random walks, one for each stock, are needed to obtain a single value of the portfolio at the end of the time period. This then constitutes a “trial” for the Monte Carlo simulation. We simulate the portfolio 2^{20} times resulting in 2^{30} random walks.

We used NVIDIA’s GeForce GTX 260 as the GPU, attached via a 16-lane PCIe interconnect. This GPU has double-precision computation capability but we used single-precision representation for all our experiments. The GTX 260 has 24 multiprocessors with a theoretical maximum of 1024 threads per multiprocessor. The actual number of threads that can be launched is limited by the number of registers available on the multiprocessors. To keep the GPU busy we simulated 1024 trials simultaneously (B , in section 4.1) resulting in 2^{20} random walks. The FPGA is a Virtex-4 LX80 from Xilinx, connected to the motherboard via PCI-X. The processors are all 2.2 GHz AMD Opterons.

We report performance in random walks per second (walks/s). Recall that the overall experiment contains 2^{30} ($\approx 10^9$) such random walks. Table 2 presents the quantitative performance results. Relative to mapping 1, the overall speedup is $180\times$. Relative to mapping 2, the overall speedup is $74\times$.

Table 2. Performance results for Monte Carlo simulation of VAR.

Mapping	Performance
1	0.45 Mwalks/s
2	1.1 Mwalks/s
3	80 Mwalks/s
4	60 Mwalks/s
5	81 Mwalks/s

Clearly, the greatest performance improvement comes in the mappings where the GPU is utilized. Since the majority of the time in the sequential algorithm is spent in stage 3 (a matrix-vector multiply), and the GPU excels at that class of computation, this result is reasonable to expect.

The performance of mapping 4 relative to mapping 3 is somewhat of a surprise. When migrating stages 1 and 2 off of the GPU and onto the FPGA, we would expect the performance to improve. Instead, we observe a performance degradation. A detailed investigation concluded that the performance decrease is due to the fact that in our implementation the GPU is not performing I/O and computation concurrently. Instead, it is waiting for the random numbers to be input from the FPGA prior to initiating the stage 3 and 4 computations. (The bandwidth achieved on this link is 1.2 GB/s.) Since the GPU can create the random number stream (which it can do at 2.8 GB/s) faster than it can ingest the random number stream, mapping 4's performance suffers as a result.

Overall, the best performance is observed with mapping 5. Here, the FPGA is feeding its random number stream into the CPUs and the GPU is internally creating its own random number stream. Since stage 3 is such a computational bottleneck, further gains in performance could be achieved by moving stage 3 onto the FPGA, for example using the approach described by Thomas and Luk [12].

6. Conclusions

This paper has described the acceleration of an important problem in computational finance, the risk assessment of a portfolio via Monte Carlo simulation techniques. Two classes of co-processor accelerator are employed, a GPU and an FPGA.

The performance results show a speedup of $74\times$ relative to an 8 processor implementation, with the bulk of the performance gain due to the GPU. The overall performance can be improved further by enabling concurrent computation and I/O on the GPU and by exploiting the FPGA to do a larger fraction of the overall load.

Acknowledgements

This work was supported in part by NSF grants CCF-0427794, CNS-0720667, and CNS-0751212. R.D. Chamberlain is a principal in Exegy, Inc. The authors would also like to thank NVIDIA for their support.

References

- [1] Paul Glasserman, *Monte Carlo Methods in Financial Engineering*, Springer, 2004.
- [2] Roger D. Chamberlain, Joseph M. Lancaster, and Ron K. Cytron, "Visions for Application Development on Hybrid Computing Systems," *Parallel Computing*, **34**(4-5):201-216, May 2008.
- [3] Makoto Matsumoto and Takuji Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. on Modeling and Computer Simulation*, **8**(1):3-30, January 1998.
- [4] *NVIDIA CUDA Compute Unified Device Architecture. Programming Guide*. Available online at <http://www.nvidia.com/cuda>.
- [5] Makoto Matsumoto and Takuji Nishimura, "Dynamic creation of pseudorandom number generators," *Monte Carlo and Quasi-Monte Carlo Methods*, pp. 56-69, Springer, 2000.
- [6] Peter Acklam, "An algorithm for computing the inverse normal cumulative distribution function." Available online at <http://home.online.no/~pjacklam/notes/invnorm>.
- [7] Ishaan L. Dalal and Deian Stefan, "A Hardware Framework for the Fast Generation of Multiple Long-period Random Number Streams," in *Proc. of 16th Int'l ACM/SIGDA Symp. on Field Programmable Gate Arrays*, pp. 245-254, February 2008.
- [8] George Marsaglia and Wai Wan Tsang, "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, **5**(8), Oct. 2000.
- [9] Guanglie Zhang, Philip H.W. Leong, Dong-U Lee, John D. Villasenor, Ray C.C. Cheung, and Wayne Luk, "Ziggurat-based hardware Gaussian random number generator," *Proc. of Int'l Conference on Field Programmable Logic and Applications*, pp. 275-280, 2005.
- [10] Mark A. Franklin, Eric J. Tyson, James Buckley, Patrick Crowley, and John Maschmeyer, "Auto-Pipe and the X Language: A Pipeline Design Tool and Description Language," in *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.
- [11] Roger D. Chamberlain and Berkley Shands, "Streaming Data from Disk Store to Application," in *Proc. of 3rd Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pp. 17-23, September 2005.
- [12] David B. Thomas and Wayne Luk, "Sampling from the Multivariate Gaussian Distribution using Reconfigurable Hardware," in *Proc. of Int'l Symp. on Field-Programmable Custom Computing Machines*, pp. 3-12, April 2007.