On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis

David A. Bader[*] Aparna Chandramowlishwaran Virat Agarwal
{bader,aparna,virat}@cc.gatech.edu
College of Computing
Georgia Institute of Technology

Abstract

Numerical simulations in computational physics, biology, and finance, often require the use of high quality and efficient parallel random number generators. We design and optimize several parallel pseudo random number generators on the Cell Broadband Engine, with minimal correlation between the parallel streams: the linear congruential generator (LCG) with 64-bit prime addend and the Mersenne Twister (MT) algorithm. As compared with current Intel and AMD microprocessors, our Cell/B.E. LCG and MT implementations achieve a speedup of 33 and 29, respectively. We also explore two normalization techniques, Gaussian averaging method and Box Mueller Polar/Cartesian, that transform uniform random numbers to a Gaussian distribution. Using these fast generators we develop a parallel implementation of Value at Risk, a commonly used model for risk assessment in financial markets. To our knowledge we have designed and implemented the fastest parallel pseudo random number generators on the Cell/B.E..

1 Introduction

The Cell Broadband Engine (or the Cell/B.E.) [8–10, 21] is a novel high-performance architecture designed by Sony, Toshiba, and IBM, primarily targeting multimedia and gaming applications. The Cell/B.E. consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high speed memory controller, and a high bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. Fig. 1 gives an architectural overview of the Cell/B.E. processor. Please refer to [3, 4, 6, 10, 11, 19] for a detailed review of Cell/B.E.'s architecture.

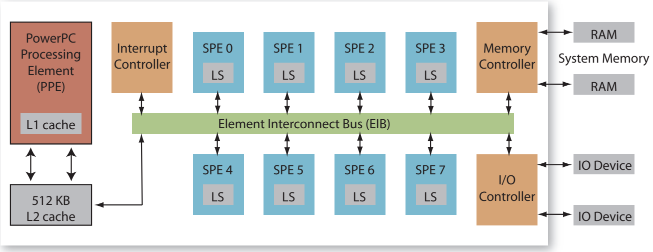

Figure 1: Cell Broadband Engine Architecture.

High quality parallel random number generators have wide applicability in cryptography, computational biology for example protein structure prediction, randomized algorithms for example in network communication, and Monte Carlo simulation. In financial analysis, Monte Carlo is a popular technique used to compute stock/asset prices, commodity prices and risk valuation that require estimating losses based on an underlying stochastic process. Parallel random number generation becomes especially important for the upcoming era of manycore architectures, such as the recently announced Intel TeraFLOPS with 80 cores on a single die [22] and the next generation of IBM Cell/B.E. that may offer 32 SPEs [7] to leverage these highly parallel chips.

In this work we design efficient parallel uniform pseudo-random number generators that have high quality and period, and have the potential to scale well across a large number of cores. We focus our efforts in optimizing the 64-bit Linear Congruential Generator (LCG) [12] and the 32-bit Mersenne Twister [17], on the Cell/B.E.. The LCG is one of the old-

[*] This work was supported in part by an IBM Shared University Research (SUR) award and NSF Grants CNS-0614915 and CAREER CCF-0611589. We acknowledge our Sony-Toshiba-IBM Center of Competence for the use of Cell Broadband Engine resources that have contributed to this research.

est pseudo-random number generators and is known to provide high quality through a simple algorithm based on a linear function and modular reduction. The Mersenne Twister provides a period of $2^{19937} - 1$ and 623-dimensional equidistribution for a certain choice of input parameters, making it well-suited for the purpose of Monte Carlo simulation. We also implement for Cell, Gaussian random number generators, the Gaussian averaging method and Box Mueller transformation, that transform uniformly distributed random numbers to a sequence with Gaussian distribution. These generators are useful in many applications such as in financial analysis, using Monte Carlo method.

We present a detailed performance comparison of our optimized implementations of random number generators over the leading multi-core and single-core processors. Our Cell-optimized 64-bit implementation of Linear Congruential Generator (LCG) with 64-bit prime addend attains a speedup of 33 using the Wallace tree approach, as compared with the performance of leading Intel processors. For our 32-bit implementation of Mersenne Twister, Cell achieves an average speedup of over 14 and 29 using block generation and using sequential approach, respectively, as compared with the performance on current Intel and AMD architectures. In earlier work [1] we implemented the MT generator on Cell/B.E. for use in financial services applications for Option pricing and Collateralized Debt Obligation pricing. The source code for LCG and Gaussian averaging method is freely available from our CellBuzz project in SourceForge (`http://sourceforge.net/projects/cellbuzz/`).

Finally, using these optimized Gaussian generators we develop a parallel 64-bit implementation for Risk Analysis on the Cell/B.E.. Value at Risk (VAR) is a commonly used model for risk assessment in financial markets. Given a portfolio of assets, this model measures the worst expected loss over a given time interval at a given confidence level. For performance analysis, we estimate the risk of one non-linear portfolio consisting of a single stock.

2 Uniform Parallel Random Number Generators

In this section we discuss two pseudo random number generators, Linear Congruential Generator and Mersenne Twister. We briefly describe the algorithm and discuss in detail the various challenges involved and optimization techniques used to achieve high performance on the Cell/B.E.

2.1 The LCG generator

Linear Congruential Generator (LCG) is one of the oldest and most studied pseudo-random number generating algorithm proposed by Lehmer [12]. In this generator each successive element is determined by a simple linear function and a modular reduction.

LCG generates pseudo-random number sequence $\{x_1, x_2, ..., x_n\}$ in the set $[0, 1, ..., m]$ by the recurrence relation of order one given below.

$$x_n = (ax_{n-1} + b) \mod m, \text{ where } n \geq 0$$

where b is the *addend*, a is the *multiplier* and m denotes the *modulus* of the algorithm.

Random numbers in the range $[0, 1]$ are then obtained by normalization ($y_n = \frac{x_n}{m}$). In this work we use a prime *addend*, and a power of two modulus, which gives the algorithm a periodicity of 2^k ($m = 2^k$).

In practice, several spectral methods are used to test the quality (randomness) of the output sequence. Lattice spacing is one such method that helps to get an insight into the granularity of a random number generator, and LCG passes this test. This granularity is computed by applying the Fourier transform on the output of the generator.

LCG's are very sensitive to the change in the value of input parameters. The most important parameter to a LCG is the modulus m, the size of which determines the period of the output sequence. The recurrence relation for a 64-bit LCG ($m = 2^{64}$) is given below.

$$x_n = (ax_{n-1} + b) \mod 2^{64}$$

The number of independent streams available for a LCG generator with these parameters is about 2^{24}, allowing for massive parallelism.

2.1.1 Challenges and Optimization

In this work, we optimize the 64-bit LCG that is available as part of the SPRNG package [15, 16].

The LCG algorithm is data dependent, i.e., to generate a random number it requires the knowledge of the number generated in the previous step. This makes it hard to parallelize the computation of a single stream among the various SPEs. An alternative technique is to instantiate the algorithm based on different parameter values on each SPE. In our design, we select a vector of unique 64-bit addends $[b_{i1}, b_{i2}]$ for each SPE i. This is computed inside the SPE based on its stream identifier. This algorithm ensures that the set of all addends are pairwise relatively prime [14], and ensures that the independent streams on the SPEs have minimum correlation. Alg. 1 gives the pseudo-code of the parallel LCG algorithm on a given SPE i.

Algorithm 1: Parallel LCG for Cell/B.E.. This algorithm gives the pseudo-code on a given SPE i. Each SPE generates two independent streams of 64-bit pseudo-random numbers.
Loop is unrolled for optimizing on the Cell processor Each iteration generates a vector consisting of two random numbers.

Input: Number of Simulations: N, Number of SPEs p, Vector b_i (consisting of 2 unique 64-bit addend parameters), Multiplier a

Output: Random Numbers: N/p

```
$va \leftarrow [a, a]$;
for $j = 1$ to $N/2p$ do
    $vx_j = va * vx_{j-1} + b_i$;
    $vx_j = vx_j * [2^{-64}, 2^{-64}]$;
```

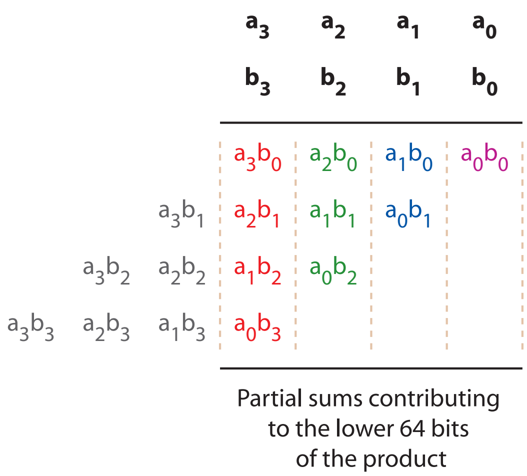

Figure 2: Multiplying two 64-bit numbers using partial products of 16-bit numbers.

The algorithm involves 64-bit multiplication and a 64-bit addition. The SPU instruction set does not have support for multiplication and addition of *unsigned long long* vector datatypes. We implemented 64-bit multiplication with Wallace trees [23], a bit slide adder version of carry-save adders. The 16 partial products of 16-bit subword multiplications (see Fig. 3) are rearranged into a modular arrangement. Each rectangle is a Nonadditive Multiply Module (NMM), and represents a 32-bit sub-product. The output requires only 64-bits, hence the upper 64-bits are discarded. As a result, we have only 10 partial products to add. The summation of sub-products are carried out by 3-input (W_3), 5-input (W_5) and 7-input (W_7) Wallace trees. Note that the 16-bit slice at the right end does not require Wallace trees. W_3 is a 3-to-2 carry-save full adder (CSA) which accepts 3 n-bit operands; generates two n-bit results: n-bit partial sum and n-bit carry. W_5 and W_7 are constructed using three 3-to-2 CSA and five 3-to-2 CSA. The advantage of CSA is that there is no carry propagation between stages.

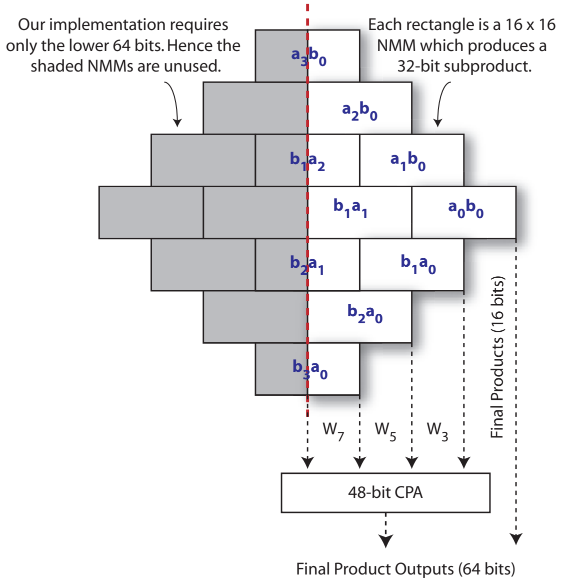
Our implementation requires only the lower 64 bits. Hence the shaded NMMs are unused.
Each rectangle is a 16 x 16 NMM which produces a 32-bit subproduct.
a3b0
a2b0
b1a2
a1b0
b1a1
a0b0
b2a1
b1a0
b2a0
b3a0
Final Products (16 bits)
W7
W5
W3
48-bit CPA
Final Product Outputs (64 bits)

Figure 3: Wallace tree improves the complexity of multiplication by reducing the number of partial products to add.

The last stage of multiplication is a 48-bit Carry Propagate Adder (CPA) which merges the two vectors into a final product. Using Wallace trees we can generate 8 random numbers simultaneously. This algorithm carries out partial product addition faster than normal addition since it adds three partial products at a time, thereby reducing the depth of the adder.

2.2 Mersenne Twister

The Mersenne Twister (MT) is a pseudo random number generator algorithm developed by Makoto and Matsumoto [17]. The algorithm is proven to have a period of $2^{19937} - 1$ and 623-dimensional equidistribution for a certain choice of parameters, thus providing a very high quality for the purpose of Monte Carlo simulation. The sequence of random numbers generated passes well known stringent tests such as Marsaglia [13], and tests on higher dimensional uniformity including the spectral test and k-dimensional test.

Alg. 2 gives the pseudo-code of the Mersenne Twister algorithm. Steps 1 & 2 create masks for upper (u) and lower (ll) bits. In Step 3, an array of size

```
Algorithm 2: Mersenne Twister algorithm
   Input: Integer constants $l$, $s$, $t$, $a$, $r$, Bit masks
          $b$, $c$ each of word size, Algorithm parameters $n$ and $m$
 1 $u \leftarrow 1...10...0$ ($w-r$ ones and $r$ zeroes);
 2 $ll \leftarrow 0...01...1$ ($w-r$ zeroes and $r$ ones);
 3 $x[0], x[1], ..., x[n-1] \leftarrow$ "any non-zero initial values";
 4 $i \leftarrow 0$;
 5 $y \leftarrow (x[i]\&u)|(x[i+1 \mod n]\&ll)$;
 6 if least significant bit of $y$ == 0 then
 7     $x[i] \leftarrow x[(i+m) \mod n] \oplus (y \gg 1)$;
 8 else
 9     $x[i] \leftarrow x[(i+m) \mod n] \oplus (y \gg 1) \oplus a$;
10 $y \leftarrow x[i]$;
11 $y \oplus (y \gg u)$;
12 $y \oplus ((y \ll s)\&b)$;
13 $y \oplus ((y \ll t)\&c)$;
14 $y \oplus (y \gg l)$;
15 output $y$;
16 $i \leftarrow (i+1) \mod n$;
17 repeat from step 5;
```

n is initialized with non-zero initial values. In Step 5, for a given value of the index i, y is computed by concatenating the upper bits of $x[i]$ with the lower bits of $x[i+1]$. Steps 6 to 14 improve the accuracy and equidistribution properties of the output random number. This array is traversed in a round robin manner during the subsequent iterations of the algorithm.

2.2.1 Challenges and Optimization

To obtain a period of $2^{19937}-1$ we choose $n = 624, m = 397$. The values of other parameters can be obtained from Makoto and Matsumoto [17]. As we observe from Alg. 2, the algorithm fits within a working area of 624 32-bit words. This is especially beneficial for the Cell architecture given the limited local store of 256KB available on any given SPE.

Since there is no branch predictor on the Cell SPE, it is important to reduce branches from the code for achieving high performance. We eliminate the branch from the Mersenne Twister algorithm, by replacing the *if-then-else* statement (Steps 5-9) with

$$x[i] \leftarrow x[(i+m) \mod n] \oplus (y \gg 1) \oplus (a \ \& \ wlsb(y))$$

where, $wlsb(y)$ gives the *word* with each bit as the *least significant bit* of y.

We next describe two ways to parallelize this for the Cell. One technique is to optimize the algorithm for a single SPE and use different seeds for various SPEs to generate multiple random streams. Using a dynamic seed for each SPE ensures that the combined stream has high quality of randomness [18]. Another technique is to generate a single stream of random numbers using the various SPEs. It is important to note that in this algorithm the computation from the latter part of the array requires the updated data from the first part which makes the algorithm data dependent. To obtain high performance on Cell, we use the first parallelization technique in our design. However, using different seeds on different SPEs is not enough since the generated random numbers from the various SPEs may be correlated, leading to degraded quality of Monte Carlo simulations. A solution to this is Dynamic Creator [18] that is based on the Mersenne Twister algorithm. This generates different algorithm parameters for the various SPEs which helps in generating multiple independent streams.

To further optimize the algorithm on Cell we use standard optimization techniques such as vectorization, loop unrolling, and data alignment. It is important to note that for the given parameter values of $n = 624$ and $m = 397$, the data access pattern of the algorithm introduces challenges for optimizing this on the SPEs. It is not straightforward to vectorize the code, as the index $i + m$ that is required during the computation in Steps 5-9 may not lie at a 16-byte boundary as required by Cell's vector intrinsics. Thus, we use *spu_shuffle* instructions to create vectors that are quadword aligned. This adds extra instructions to the algorithm and results in a slight degradation of performance.

3 Gaussian Parallel Random Number Generators

The random number generators discussed in the previous section generate uniform random numbers (random numbers uniformly distributed in the interval $[0, 1]$). Many applications, such as the Monte Carlo method, require a random variable with Gaussian (normal) distribution (range $\in [-1, 1]$, mean $= 0$, variance $= 1$). In this section we present three methods that transform a set of uniform random numbers into normalized random numbers and report their performance on the Cell processor.

3.1 Gaussian Averaging Method

Gaussian averaging method [5] transforms a stream of uniform random numbers into a Gaussian (normal) distribution. To generate a Gaussian with mean μ and standard deviation σ, n uniform random numbers are added together into s, and the output is $\mu + \sigma s\sqrt{\frac{3.0}{n}}$. The parameter n determines the accuracy of the transformation. If n is large, then accuracy of the output increases along with an increase in the running time. To optimize this for the Cell we use vectorization and loop unrolling.

3.2 Box Mueller Polar/Cartesian

For every pair of input random numbers, Box Mueller transformation [2] in Cartesian form generates a pair of normalized random numbers. Alg. 3 gives the pseudo-code of this transformation algorithm.

Algorithm 3: Box Mueller transform in Cartesian form.

```
Input: Independent uniform random numbers ($x$,$y$)
Output: Normal random numbers ($\bar{x}$, $\bar{y}$)
$R = \sqrt{-2 * \ln x}$;
$\theta = 2\pi * y$;
$\bar{x} = R * \cos\theta$;
$\bar{y} = R * \sin\theta$;
```

In the Polar form for every pair of input random numbers, a pair of normalized numbers is generated if the input pair lies within a unit disc. Alg. 4 gives the pseudo-code of this transformation algorithm.

Algorithm 4: Box Mueller transform in Polar form.

```
Input: Independent uniform random numbers ($x$,$y$)
Output: Normal random numbers ($\bar{x}$, $\bar{y}$)
$s = x^2 + y^2$;
if $0 < s \leq 1$ then
    $z = \sqrt{\frac{-2*\ln s}{s}}$;
    $\bar{x} = x * z$;
    $\bar{y} = y * z$;
```

The presence of a branch in the Box Mueller transformation algorithm (Polar form) poses several issues during optimization on the Cell. The branch restricts vectorization of the algorithm, and leads to the degradation in performance. This problem is equivalent to extracting elements from a long input array A that satisfy a given condition X, using vector intrinsics. Alg. 5 gives an elegant way to eliminate this branch using a two stage approach.

Algorithm 5: Extracting elements from array A that satisfy a condition X, using a vectorized approach

```
Input: array $A$, length $N$
Output: array $C$ consisting of $j$ elements
for $i \leftarrow 1$ to $N$ do
    $B[i] = X(A[i])$;
$j \leftarrow 0$;
for $i \leftarrow 1$ to $N$ do
    $C[j] = A[i]$;
    $j = j + B[i]$;
```

In comparison to the Box Mueller transform in Cartesian form, this algorithm discards about one in four pairs of input random numbers, but it avoids the use of a trigonometric function (which is comparatively an expensive operation). Thus, Box Mueller in Polar form is a computationally less expensive as compared to the Cartesian form, but harder to optimize on the Cell.

For compute intensive operations such as *log*, *sqrt*, *sin* and *cos* we use the 64-bit vector routines available as part of the SIMD math library for the Cell Broadband Engine.

4 Performance Analysis

We report our performance results from actual runs on a IBM BladeCenter QS20, with two 3.2 GHz Cell/B.E. processors, 512 KB Level 2 cache per processor, and 1 GB memory (512 MB per processor). For performance comparisons we compile our code using the xlc compiler provided with Cell SDK 2.1, with level 3 optimization.

Linear Congruential Generator

Table 1 lists the performance of our 64-bit Linear Congruential generator implementation on Cell in terms of million random numbers generated per second (MRS) and compares with other architectures. For performance results on these architectures we used the SPRNG optimized implementation of LCG and compile it with *icc v9.0* with level 3 optimization for Intel Xeon 5150 (Woodcrest) and *-fast* optimization for Intel Xeon 3 GHz and Intel Pentium 4 processors. Fig. 4 plots the performance and reports the speedup of our implementation across these various architectures. We achieve speedup of 33.2 over a 2.6 GHz Intel Xeon 5150.

Table 1: LCG performance

CPU/Compiler	LCG (Million random numbers/second)
Intel Xeon, 3GHz Intel C/C++ v9.0	8.6
Intel Pentium 4, 3.2 GHz Intel C/C++ v9.0	8.3
Intel Xeon 5150, 2.6 GHz Intel C/C++ v9.0	100.0
IBM Cell/B.E., 3.2 GHz xlc (Cell SDK 2.1)	3323.4

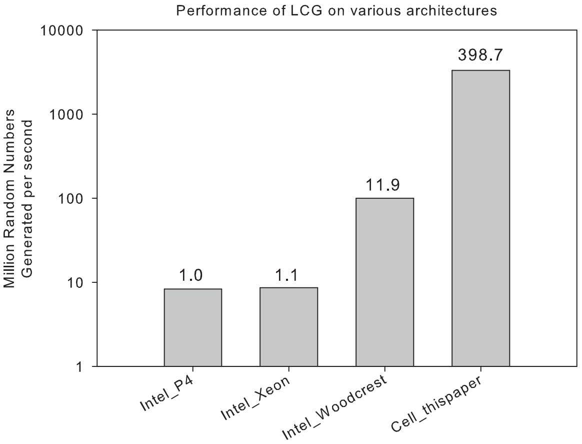

Figure 4: Comparison of performance of 64-bit LCG across various architectures in terms of million random numbers generated per second, as reported in Table 1. The number above each bar represents the speedup of the corresponding architecture as compared with a 3.2 GHz Intel Pentium 4.

Mersenne Twister

Table 2 lists the performance of our 32-bit Mersenne Twister implementation on Cell and compares with other architectures. For performance comparisons with Intel, AMD and IBM PowerPC processors we use results from optimized implementations (using SIMD instructions) of the Mersenne Twister algorithm as reported by Saito and Matsumoto [20].

Fig. 5 reports the performance and plots the speedup of our 32-bit Mersenne Twister implementation on Cell (using one Cell processor) and compares with other architectures. Block approach generates a block of random numbers and Sequential approach generates one random number per iteration. *MT(SIMD)* gives the performance of a vectorized implementation of the Mersenne Twister algorithm. We achieve speedup of 11.5 over Intel Pentium 4, 3.0 GHz in the block random number generation and a speedup of 22.2 using the sequential approach. The Cell optimized implementation generates 3.2 billion psuedo random numbers per second from a single Cell processor.

Table 2: Time in seconds using MT to generate 100 million random 32-bit samples in sequential and block pattern on various architectures. The performance results on the Intel, AMD and IBM PowerPC processors are from Saito and Matsumoto [20].

CPU/Compiler	Output	MT	MT (SIMD)
Intel Pentium-M 1.4 GHz Intel C/C++ v9.0 [20]	block	1.122	0.627
	seq	1.511	1.221
Intel Pentium-4 3.0 GHz Intel C/C++ v9.0 [20]	block	0.633	0.391
	seq	1.014	0.757
AMD Athlon 64 3800+ 2.4 GHz, gcc v4.0.2 [20]	block	0.686	0.376
	seq	0.756	0.607
IBM PowerPC G4 1.33 GHz, gcc v4.0.0 [20]	block	1.089	0.490
	seq	1.794	1.358
IBM Cell/B.E. 3.2 GHz xlc	block	-	0.034
	seq	-	0.036

5 Case Study: Risk Analysis

The Value at Risk (VAR) is a commonly used model for risk assessment in the Financial Services Sector. A VAR statistic has three components: a time period, a confidence level and a loss amount (or loss percentage). This model aims at computing the worst expected loss over a given time interval at a given confidence level. The confidence level is usually either 95% or 99%. We model stock prices using the Geometric Brownian Motion (GBM) model, which is technically a Markov process. This means that the stock price follows a random walk and is consistent with the weak form of the efficient market hypothesis (EMH): past price information is already incorporated and the next price movement is conditionally independent of past price movements.

In this section, we present the estimation of the VAR for a portfolio consisting of a single stock. Monte Carlo (MC) simulation is a popular method for estimating this value when high precision is desired for non-linear portfolios. In MC simulation the number of cycles N in general is very large, and the cycles are independent of one another. Thus, we divide the number of cycles among the various SPEs, with each SPE computing results from N/p cycles, where p is the number of SPEs.

Given the limited local store on an SPE precomputing the normal random numbers and storing them on the PPE should be avoided. Instead, we cal-

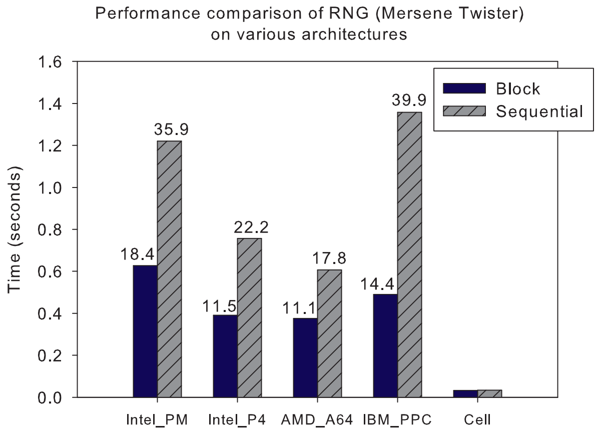

Figure 5: Comparison of running times to generate 100 million 32-bit random samples in sequential and block pattern on various architectures as reported in Table 2. The number above each bar represents the speedup of Cell/B.E. as compared with the corresponding architecture.

culate these numbers during each Monte Carlo cycle. The role of the PPE in the algorithm is to gather input data from the user, partition the work among the various SPEs (divide the total number of cycles), create SPE threads, and gather the computed stock price value from each SPE. The pseudo-code for risk assessment of stock prices using Monte Carlo simulation is given by Alg. 6. In the risk analysis algorithm, Steps 2 & 3 are computationally intensive.

Algorithm 6: Monte Carlo method for Risk Analysis

```
Input: Current Stock Price ($S$), Expected return ($\mu$), Standard Deviation of returns ($\sigma$), Time ($\Delta t$), Number of cycles ($N$)
1 for $j = 1$ to $N$ do
2     Generate uniform random number $r$;
3     Transform $r$ to Gaussian (normal) random number $\epsilon$;
4     Compute $\Delta S = S\ (\ \mu\Delta t + \sigma\epsilon\sqrt{\Delta t})$;
```

Step 4 computes the value of stock price, S. The first term is a *drift* and the second term is a *shock*. For each time period, the GBM model assumes the price will drift up by the expected return. But the drift will be shocked (added or subtracted) by a random shock that is the standard deviation, σ, multiplied by a random number, ϵ.

5.1 Performance Analysis

We use different combinations of parallel uniform and Gaussian random number generators (RNGs) to develop Cell-based 64-bit implementations of the Value at Risk (VAR) model. For uniform RNG we use our optimized implementation of Linear Congruential pseudo random number generator with 64-bit prime addend, and for Gaussian RNG we use the Gaussian averaging method, and the Box Mueller transformation in Polar/Cartesian form described earlier in Section 3. Fig. 6 reports the running time for each of these implementations.

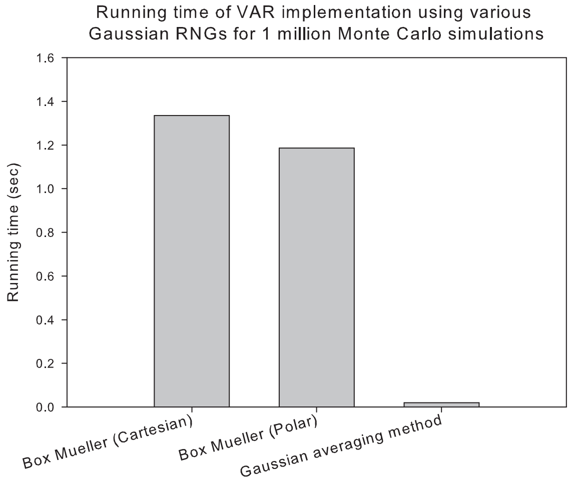

Figure 6: VAR performance

For our 64-bit implementation of the Gaussian averaging method, $n = 8$ random numbers are added together to generate the output. The performance of implementations that use Box Mueller transformantion are significantly lower than the Gaussian averaging method due to inefficient vector routines for mathematical operations such as *sin,cos,sqrt,log.* All of our implementations are optimized for the Cell/B.E. using standard optimization techniques such as vectorization, loop unrolling and data alignment for best performance.

6 Conclusion

We design optimized parallel implementations of two pseudo random number generators, the 64-bit Linear Congruential Generator (LCG) and the 32-bit Mersenne Twister (MT) for the Cell Broadband Engine. To optimize the 64-bit operations in LCG we use Wallace tree method. For the MT implementation, we improve the algorithm to eliminate branches and optimize the code using standard techniques such as loop unrolling and vectorization. As compared with current Intel and AMD microprocessors, our parallel LCG and MT implementations achieve a speedup of 33 and 29, respectively. We also optimize three Gaussian random number generators, Gaussian Averaging method and the Box Mueller transformation in Polar & Cartesian forms and explore their performance and accuracy for the purpose of Monte Carlo simulation. We use these generators for the Value at Risk model, a commonly used model for risk assessment in financial markets. These generators can also be widely used for applications that are run on Monte Carlo simulation. To our knowledge we have designed and implemented the fastest parallel pseudo random number generators on the Cell/B.E..

References

[1] V. Agarwal, L.-K. Liu, and D. Bader. Financial modeling on the cell broadband engine. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2008)*, Miami, FL, Apr. 2008.

[2] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[3] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`, Nov. 2005.

[4] B. Flachs, S. Asano, S. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processor unit for a Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 134–135, San Fransisco, CA, USA, February 2005.

[5] R. George. Algorithm 200: normal random. *Communications of the ACM*, 6(8):444, 1963.

[6] H. Hofstee. Cell Broadband Engine Architecture from 20,000 feet. `http://www-128.ibm.com/developerworks/power/library/pa-cbea.html`, Aug. 2005.

[7] H. Hofstee. Real-time supercomputing and technology for games and entertainment. In *Proc. SC*, Tampa, FL, Nov. 2006. (Keynote Talk).

[8] IBM Corporation. Cell Broadband Engine technology. White paper.

[9] IBM Corporation. The Cell project at IBM Research. White paper.

[10] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[11] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[12] D. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery*, pages 141–146, Cambridge, MA, May 1951.

[13] G. Marsaglia. A current view of random numbers. In *Computer Science and Statistics, Proceedings of the Sixteenth Symposium on The Interface*, pages 3–10, North-Holland, Amsterdam, 1985.

[14] M. Mascagni. Parallel linear congruential generators with prime moduli. *Parallel Computing*, 24(5–6):923–936, 1998.

[15] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.

[16] M. Mascagni, A. Srinivasan, S. Ceperley, and F. Saied. *Scalable Parallel Random Number Generators (SPRNG) Library.* Florida State University, 2.0 edition, 1995. `sprng.cs.fsu.edu`.

[17] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[18] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer, 2000.

[19] D. Pham, E. Behnen, M. Bolliger, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, B. Le, Y. Masubuchi, S. Posluszny, M. Riley, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. The design and implementation of a first-generation Cell processor. In *International Solid State Circuits Conference*, volume 1, pages 184–185, San Fransisco, CA, USA, February 2005.

[20] M. Saito and M. Matsumoto. Simple and Fast MT: A Two times faster new variant of Mersenne twister. 2006.

[21] Sony Corporation. Sony release: Cell architecture. White paper.

[22] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. In *International Solid State Circuits Conference*, pages 29–41, Lille, France, 2007.

[23] C. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, EC-13(1):14–17, Feb. 1964.