Financial Modeling on the Cell Broadband Engine

Virat Agarwal
virat@cc.gatech.edu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Lurng-Kuo Liu
lkliu@us.ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

David A. Bader
bader@cc.gatech.edu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

High performance computing is critical for financial markets where analysts seek to accelerate complex optimizations such as pricing engines to maintain a competitive edge. In this paper we investigate the performance of financial workloads on the Sony-Toshiba-IBM Cell Broadband Engine, a heterogeneous multicore chip architected for intensive gaming applications and high performance computing. We analyze the use of Monte Carlo techniques for financial workloads and design efficient parallel implementations of different high performance pseudo and quasi random number generators as well as normalization techniques. Our implementation of the Mersenne Twister pseudo random number generator outperforms current Intel and AMD architectures by over an order of magnitude. Using these new routines, we optimize European Option (EO) and Collateralized Debt Obligation (CDO) pricing algorithms. Our Cell-optimized EO pricing achieves a speedup of over 2 in comparison with using RapidMind SDK for Cell, and comparing with GPU, a speedup of 1.26 as compared with using RapidMind SDK for GPU (NVIDIA GeForce 8800), and a speedup of 1.51 over NVIDIA GeForce 8800 (using CUDA). Our detailed analyses and performance results demonstrate that the Cell/B.E. processor is well suited for financial workloads and Monte Carlo simulation.

1. Introduction

The Cell Broadband Engine (or the Cell/B.E.) [20, 17, 18, 35] is a novel high-performance architecture designed by Sony, Toshiba, and IBM (STI), 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. The Cell is used in Sony's PlayStation 3 gaming console, Mercury Computer System's dual Cell-based blade servers, and IBM's QS21 Cell Blades.

High performance computing is critical in the Financial Sector to aid complex analytics in financial models. These models aim to understand and deal with uncertainties prevalent in the market. The use of technologies such as multicore processors, clusters and grid computing is well established in this market, with growing interest in the Cell/B.E., GPUs and FPGAs [15]. The IBM Cell/B.E. is known to perform well for applications that are compute-intensive [36].

Option pricing is a basic financial model and is used extensively throughout the Financial Services sector for pricing European, American, Bermuda and various other options. An option is a contract between two parties (buyer/seller), where the buyer has the right but not the obligation to engage in a future

transaction based on a financial instrument. The literature contains several publications related to parallel option pricing algorithms. Parallel algorithms for pricing various types of options using the multinomial lattice model are discussed in [4, 16, 10]. Optimizing this on the Cell requires communication and synchronization after every stage which leads to degraded performance. Option pricing using parallel algorithms based on Monte Carlo method are presented in [24, 29, 37]. Collateralized Debt Obligation (CDO) is the fastest growing sector of the asset backed securities market. According to the Securities Industry and Financial Markets Association (SIFMA), global CDO issuance increased to $549.2 billion in 2006, over twice the $271.8 billion issued in 2005 [33]. Collateralized Debt Obligation pricing algorithms are discussed in [7, 6].

Monte Carlo simulation is a popular technique used in financial markets to compute stock/asset prices, commodity prices and risk valuation that require estimating losses based on an underlying stochastic process. It also has wide application in computational physics, physical chemistry and computational biology. In this work, we design an efficient parallel pseudo-random number generator based on Mersenne Twister algorithm [25] and a quasi-random number generator based on the Hammersley Sequence [12]. For our implementation of Mersenne Twister, Cell achieves a speedup of over 11 as compared to the performance on current Intel and AMD architectures. We explore and analyze the performance of various normalization techniques such as Low Distortion Map (LDM) [34], Box Mueller transform [3] in Cartesian and Polar forms. Using these routines for Monte Carlo simulation we develop an efficient parallel implementation for pricing European options using the Black Scholes option pricing model. We also present the design of an efficient parallel CDO pricing algorithm. Monte Carlo simulation has been the most popular method for CDO valuation and can be very resource intensive for large CDOs. In our design, we use the Monte Carlo approach with Gaussian Copula. This algorithm consists of several important scientific kernels that are proven to achieve high performance on the Cell/B.E.

Section 5 provides an extensive performance comparison of our implementation over NVIDIA G80 using CUDA SDK and RapidMind Development Platform v2.1 for GPUs. The RapidMind development platform helps developers create high performance applications with less effort and low cost. We also compare the performance of our hand tuned code as com-

pared to using RapidMind Development platform v2.1 for Cell. Our implementation achieves a speedup of 1.51 over NVIDIA G80 (using CUDA), a speedup of over 2 as compared with using RapidMind for Cell and a speedup of 1.26 as compared with using RapidMind for GPU.

Our detailed analyses and performance results suggest that Cell is well suited for financial workloads. Also, the Monte Carlo method is highly scalable among the various SPEs. This is particularly important for the next generation of the Cell processor that may offer 32 SPEs [14].

2. Cell Broadband Engine Architecture

The Cell Broadband Engine (Cell/B.E.) processor is a heterogeneous multi-core chip that is significantly different from conventional multiprocessor or multi-core architectures. It consists of a traditional microprocessor (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. We refer the reader to [30, 8, 21, 13, 5] for additional details.

The PPE runs the operating system and coordinates the SPEs. It is a 64-bit PowerPC core with a vector multimedia extension (VMX) unit, 32 KByte L1 instruction and data caches, and a 512 KByte L2 cache. The PPE is a dual issue, in-order execution design, with two way simultaneous multithreading. Ideally, all the computation should be partitioned among the SPEs, and the PPE only handles the control flow.

Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a micro-architecture designed for high performance data streaming and data intensive computation. It includes a 256 KByte *local store* (LS) memory to hold SPU program's instructions and data. The SPU cannot access main memory directly, but it can issue DMA commands to the MFC to bring data into the Local Store or write computation results back to the main memory. DMA is non-blocking so that the SPU can continue program execution while DMA transactions are performed.
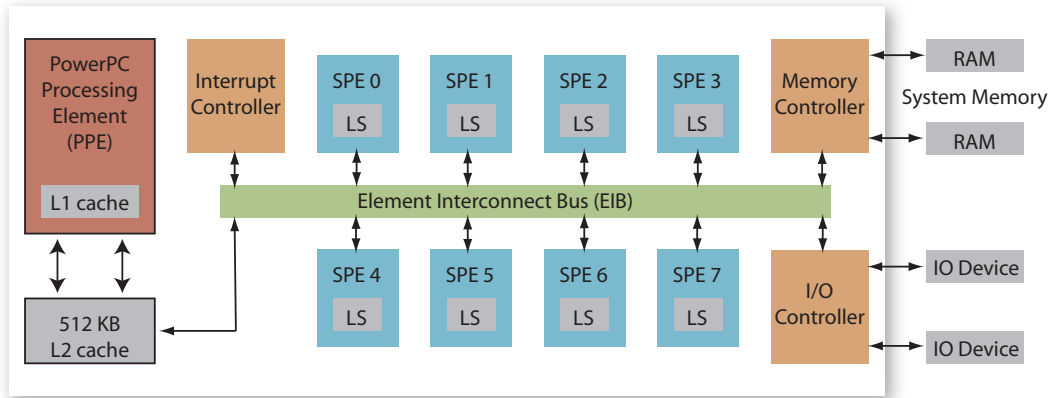
Figure 1. Cell Broadband Engine Architecture.

The SPU is an in-order dual-issue statically scheduled architecture. Two SIMD [19] instructions can be issued per cycle: one compute instruction and one memory operation. The SPU branch architecture does not include dynamic branch prediction, but instead relies on compiler-generated branch hints using *prepare-to-branch* instructions to redirect instruction prefetch to branch targets. Thus branches should be minimized on the SPE as far as possible.

The MFC supports naturally aligned transfers of 1,2,4, or 8 bytes, or a multiple of 16 bytes to a maximum of 16 KBytes. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. Peak performance is achievable when both the effective address and the local storage address are 128 bytes aligned and the transfer is an even multiple of 128 bytes. In the Cell/B.E., each SPE can have up to 16 outstanding DMAs, for a total of 128 across the chip, allowing unprecedented levels of parallelism in on-chip communication. Kistler *et al.* [21] analyze the communication network of the Cell/B.E. and state that applications that rely heavily on random scatter and or gather accesses to main memory can take advantage of the high communication bandwidth and low latency.

With a clock speed of 3.2 GHz, the Cell processor has a theoretical peak performance of 204.8 GFLOP/s (single precision). The EIB supports a peak bandwidth of 204.8 GB/s for intrachip transfers among the PPE, the SPEs, and the memory and I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 GB/s to main memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound.

3. Financial modeling using Monte Carlo simulation

In this section we give a brief overview of Option pricing and Collateralized Debt Obligation pricing. We also discuss the use of Monte Carlo simulation in financial modeling.

3.1. Option pricing

An option is a contract between two parties where one party has the right but not the obligation to engage in a future transaction on an underlying security. Option pricing involves the computation of the option payoff value that depends on the current price of the underlying asset, expiration time, volatility of asset, and risk-free rate. The Black Scholes formula [2] is a celebrated model used to price options that is based on the assumption that the price of the underlying security follows the geometric Brownian Motion described as a continuous time stochastic differential equation. In a situation where there is no arbitrage the price can be calculated as the discounted expected value under the risk-neutral measure. Using the Monte Carlo method [27], this value can be calculated by averaging over a large number of sample values.

The pseudo-code for option pricing using the Monte Carlo method is given in Alg. 1. In the option pricing algorithm, Steps 1 & 2 are the most computationally intensive steps, i.e. generating standard Gaussian (normal) random numbers. These are random numbers that have the following probability density function.

$$f(x) = \frac{1}{2\pi}e^{-\frac{1}{2}x^2}$$
$$\text{mean} = 0, \text{ variance} = 1$$

Algorithm 1: Monte Carlo method for option pricing

Input: Current Price (S, Strike Price (K), Expiration time (T), Volatility (v), Yield rate (r), Number of cycles (N)

Output: $\hat{C}$: Discounted payoff value

1 for $j \leftarrow 1$ **to** N **do**

2   Generate uniform random number x;

3   Transform x to Gaussian (normal) random number $\hat{x}$;

4   Compute $C_j = S * e^{(r-0.5v^2)*T+v*\sqrt{T}*\hat{x}}$;

5   Compute $C_j = \max(0, C_j - K)$ for Call, $C_j = \max(0, K - C_j)$ for Put;

6 Average current option value $\hat{C} = e^{-rT} * \frac{1}{N}\Sigma_{j=1}^{N}C_j$;

Option pricing is fundamental workload to the Financial Services Sector and is widely used to trade stock, commodity, bond and index options.

3.2. Collateralized Debt Obligation (CDO) pricing

A Collateralized Debt Obligation (CDO) [7, 6] is a structured finance product that assembles a portfolio from an underlying diversified pool of defaultable assets such as corporate loans, junk bonds, and mortgages with potential high or low correlation. These defaultable assets introduce risk exposures to the portfolio. A CDO segments the risk into various tranches with a unique risk/return/maturity profile to appeal to a wide variety of investors. The risk can then be transferred to the investors through these various tranches. While pricing a multi-asset portfolio the correlation of defaults (which forms the correlation matrix C) within the portfolio becomes an important factor along with the individual default probability distributions. A higher correlation of defaults within a portfolio implies a higher risk of many assets defaulting at the same time. Thus, for pricing the CDO we need to model the joint distribution of the default times of the assets in the portfolio. Pricing a CDO using Monte Carlo simulation involves creating sample paths of correlated default times of the various assets in the portfolio. We use Monte Carlo simulation with Gaussian Copula approach [23] to sample the default times. Assuming the number of assets in a CDO is N, pricing the CDO using Monte Carlo simulation with Gaussian copula involves the following steps.

1. Generate independent uniform random numbers.

2. From uniform random numbers generate a vector of normal random numbers W (of length N), using Box Mueller transform in Polar form.

3. Perform Cholesky decomposition on the correlation matrix, $R = C \cdot C^T$.

4. Generate correlated normal random numbers with $X = C \cdot W$.

5. Convert this sample to correlated uniform random numbers by applying normal cumulative distribution function. Generate default times from correlated uniforms by inversion.

6. Sort default times to discard the ones that are after the maturity date.

7. Using the default times calculate CDO payments, and discount these payments to get present value.

8. Repeat this procedure for a given input of Monte Carlo paths.

4. Financial Modeling on the Cell/B.E.

From the previous section we observe that one of the most computationally intensive steps in using Monte Carlo simulation for financial modeling is the generation of standard Gaussian (normal) random numbers. To compute these we first compute uniform-pseudo/quasi random numbers and then transform them to standard normal random numbers using standard normalization techniques.

4.1. Random Number Generation

pseudo-random
The Monte Carlo method requires a high quality random number generator. We use the Mersenne Twister algorithm [25] as the pseudo-random number generator in our design. Fig. 2 gives an illustration of the algorithm for ($N=624$, $M=397$). It uses an input seed to

initialize an array of size N. This array is traversed in a round robin manner during the subsequent iterations of the algorithm. During each iteration, element i is updated using element $i + 1$ and $i + M$. A series of shift and bitwise operations on the ith element gives the output random number.
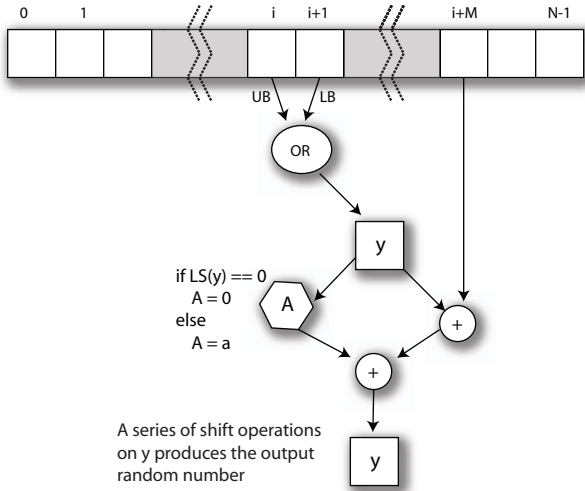

Figure 2. Illustration of the Mersenne Twister Algorithm. The function LS extracts the least significant bit from the input, and a is a constant in the algorithm.

There are 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 [26]. 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 [26] that is based on the Mersenne Twister algorithm. This generates different algorithm parameters for the various SPEs which helps in generating multiple independent streams.

The data access pattern of the algorithm introduces

challenges for optimizing this on the SPEs. For vectorization, the data access should be aligned to a 16 byte boundary. Calculating random number from array element i requires the value from element $i + M$, which may not be aligned in most cases. This requires using shuffle intrinsics within the SPE that degrades performance. In our implementation we use several techniques for optimization such as loop unrolling, branch hints, vectorization and use compare and select instructions to eliminate the branch in the algorithm.

quasi-random
Another technique uses the quasi Monte Carlo simulation for option pricing. This requires a quasi-random number generator.
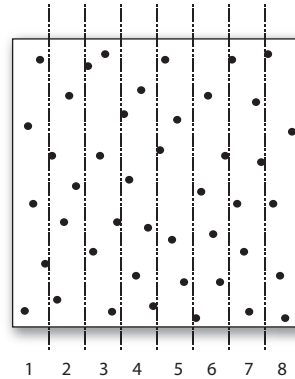

Figure 3. Illustration of the Cell parallelization for quasi-random number generation using Hammersley sequence. In this figure 8 SPEs are used for illustration. Here, SPE i is responsible for generating Hammersley points from the domain i.

In our approach we use the Hammersley sequence [12] that generates points uniformly within a square and has better statistical properties than many pseudo-random number generators. These sequences are known as low discrepancy sequences [28, 11].

The pseudo-code of the algorithm is given in Alg. 2. This algorithm is computationally more expensive than Mersenne Twister, but is less branchy.

For parallelizing this on the Cell processor we divide the square domain into p equal parts (where p is the number of SPEs). SPE i generates Hammersley sequence from block i. Note that this algorithm generates a deterministic sequence and thus it is important to parallelize a single stream of random numbers. This is in contrast to the technique we used for paralleliz-

---

Algorithm 2: Generating Hammersley point set

Input: Number of Simulations: N

Output: $\{(x_i, y_i)\}$, $i \in [1, N/2]$

                        //Divide loop iterations among p SPEs.

1   **for** $j \leftarrow 1$ ***to*** $\frac{N}{2}$ **do**

2 $x_j = \frac{j}{N}$ //Unroll and vectorize for Cell optimization.

3 $y_j = \frac{bit\_reverse(j)}{MAX\_INT}$

Algorithm 3: Box Mueller transform in Cartesian form.

Input: Independent uniform random numbers (x,y)

Output: Normal random numbers $(\bar{x}, \bar{y})$

1 $R = \sqrt{-2 * \ln x}$ //One multiplication, One logarithm, One square root

2 $\theta = 2\pi * y$ //One multiplication

3 $\bar{x} = R * \cos\theta$ //One multiplication, One trigonometric function

4 $\bar{y} = R * \sin\theta$ //One multiplication, One trigonometric function

---

ing the Mersenne twister algorithm. Fig. 3 gives an illustration of this technique. The x-coordinate of the domain is divided into p equal parts, where p is the number of SPEs. For each $x_j \in$ part i, SPE i generates the corresponding y_j.

4.2. Normalization

The random number generators discussed in the previous section generate uniform random numbers (random numbers uniformly distributed in the interval $[0, 1]$). The Monte Carlo approach for financial modeling (Alg. 1) requires a random variable with Gaussian (normal) distribution (range $\in [-1, 1]$, mean $= 0$, variance $= 1$). In this section we analyze three techniques that could be used for transforming a set of uniform random numbers to normalized random numbers, and report their performance on the Cell processor.

Box Mueller transformation in Cartesian form
For every pair of input random numbers, Box Mueller transformation [3] in Cartesian form generates a pair of normalized random numbers. Alg. 3 gives this transformation along with the computational effort required at each step.

For compute intensive operations such as *log*, *sqrt*, *sin* and *cos* we use the latest MASS (Mathematical Acceleration Subsystem) library that is available with Cell SDK 3.0. The MASS library routines take array inputs and give the best performance for large array sizes. We structure our implementation to provide long array inputs to the MASS routines in order to attain high performance.

Box Mueller transformation in Polar form
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 this transformation along with the computation effort required at each step.

In comparison to the Box Mueller transform in Cartesian form, this algorithm discards about one in four pairs of input random numbers, but it prevents 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.

The presence of a branch in the algorithm poses issues during optimization on the Cell. The branch restricts vectorization of the algorithm. Also, due to the absence of a branch predictor on the SPEs it leads to a degradation in performance. This first problem reduces to extracting elements from a long input array A that satisfy a given condition X (let's call these 'good' elements), using vector intrinsics. To solve this problem in an elegant manner we create another array B, that stores the counter value for the corresponding element of A, i.e., If $A[i]$ is 'good', then $B[i]$ is 1 otherwise 0. Using B we extract the 'good' elements of A. The pseudo-code of this technique is given in Alg. 5. Step 4 of the algorithm writes into the array C regardless of the value of $A[i]$. Note that

Algorithm 4: Box Mueller transform in Polar form.

Input: Independent uniform random numbers (x,y)

Output: Normal random numbers $(\bar{x}, \bar{y})$

1   $s = x^2 + y^2$ //Two multiplications, One addition

2   **if** $0 < s \leq 1$ **then**

3      $z = \sqrt{\frac{-2*\ln s}{s}}$ //One multiplication, One division, One square root, One logarithm

4      $\bar{x} = u * z$ //One multiplication

5      $\bar{y} = v * z$ //One multiplication

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

Input: array A, length N

Output: array C, number of extracted elements j

1   **for** $i \leftarrow 1$ **to** N **do**

2      $B[i] = X(A[i])$ //Vectorize and Unroll for Cell optimization.

3   $j \leftarrow 0$

4   **for** $i \leftarrow 1$ **to** N **do**

5      $C[j] = A[i]$ //Unroll for Cell optimization.

6      $j = j + B[i]$

---

this is correct as the array index of C increments only when a 'good' element is written to it. This leads to extra work but prevents branching from a conditional store. Although, Steps 4 & 5 of the algorithm are scalar, branch elimination significantly boosts the performance of the algorithm on the Cell processor. We use the MASS library for compute intensive operations such as *sqrt* and *log*.
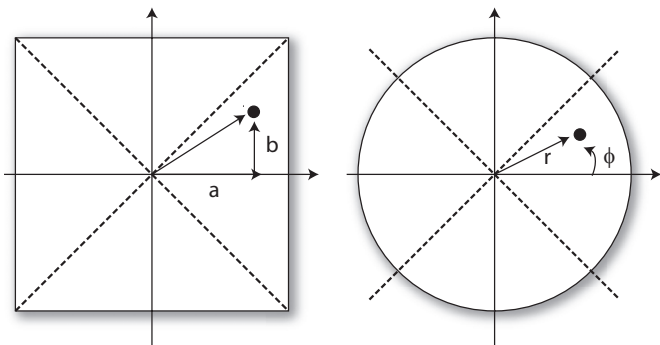
LDM : Low Distortion Map between square and disc

Figure 4. Illustration of the Low Distortion Map transformation. $r = a, \phi = \frac{\pi*b}{4*a}$

Shirley and Chiu [34] describe a low distortion map (LDM) between a unit square and disk. This map preserves fractional area and introduces low distortion in shape. For a given point (a, b) within a unit square this transformation calculates values $r = a$, $\phi = \frac{\pi b}{4a}$. This maps to the output transformed point $(\bar{a}, \bar{b})$, where $a = r \cos \phi$, and $b = r \sin \phi$. Fig. 4 gives an illustration of the algorithm. In our implementation we map every point in the unit square to the first quadrant and apply the LDM transformation. We use various optimization techniques such as vectorization, loop-unrolling and use the IBM MASS library to achieve high performance.

4.3. Optimizing option pricing for Cell

In Monte Carlo Simulation the number of cycles N (Alg. 1) 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 $\frac{N}{p}$ cycles, where p is the number of SPEs. We use our optimized kernels of random number generation and normalization as described earlier. Given the limited local store on an SPE pre-computing the normal random numbers and storing them on the PPE should be avoided. Instead, we calculate these numbers during each Monte Carlo cycle. Using Cell SIMD instructions we simultaneously calculate payoff

values from four standard normal random numbers. Also for efficient pipeline utilization, we unroll the *for* loop in Alg. 1 by a factor of 8.

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, gather the computed payoff value from each SPE, and average them to compute the final payoff value.

4.4. Optimizing Collateralized Debt Obligation (CDO) pricing for Cell

Similar to the option pricing algorithm, the number of Monte Carlo cycles in the CDO pricing algorithm is large. Thus, we equally divide the N cycles among the various available SPEs, and each SPE is responsible for simulating $\frac{N}{p}$ paths, where p is the number of available SPEs.
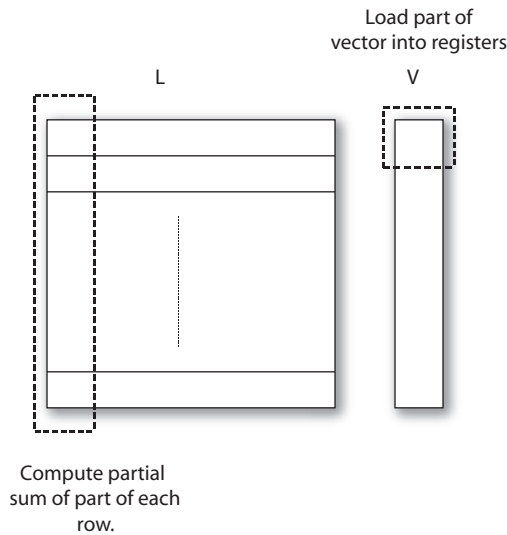

Figure 5. Illustration of the technique for Matrix Vector Multiplication.

For the first two stages of the algorithm we use our Cell optimized implementation of Mersenne Twister, and Box Mueller transform in Polar form as discussed in the previous sections. This accounts for a major part of the running time of the algorithm. Cholesky decomposition accounts for a very small fraction (2-3%) of the profile time for CDOs where the number of assets in the portfolio is less than 150. Since, this is generally the case while pricing CDOs and a matrix of this size fits entirely within the SPE local store, we use a sequential Cholesky decomposition that resides on the SPEs.

To minimize the communication between the PPE and the SPEs, we structure our implementation so that all stages of the CDO pricing algorithm happen within the SPE, the results from one stage are used directly by the next stage within the SPE, and only the final result after stage 7 is transferred back to the PPE.

For Step 4 of the algorithm, the normalized random numbers vector from Step 2 is multiplied with the output of Cholesky decomposition to produce correlated random numbers. This can be achieved by an efficient SPE matrix-vector multiplication implementation. The algorithm suggests the use of the *madd* vector intrinsic within the SPEs that does a multiply and add in 6 clock cycles. This instruction requires 2 loads that require 6 clock cycles each. For achieving high performance for this routine it is important to prevent one of these loads by reusing the registers for several vector dot product computations. This is possible when we know the size of the input vectors at the beginning. For implementing a generic routine we propose a strategy that reuses the loaded registers from earlier vector dot product computation. This technique is illustrated in Fig. 5. A part of the vector (say 8 elements) is loaded into the registers and the corresponding part from each row of the matrix is multiplied to generate partial sums.

Step 6 of the algorithm sorts the vector of default times during each iteration of the Monte Carlo method. This vector is of length N, the number of assets in the portfolio. Since CDO pricing does not involve pricing arbitrarily large portfolios, we can assume that this vector fits within the SPE for practical purposes. This problem reduces to optimizing a sequential sorting algorithm for a single SPE.

The above analysis suggests that the CDO pricing algorithm using Monte Carlo method is a good fit for the Cell/B.E.. The algorithm is compute-intensive and does not require any communication between the SPEs. The SPEs communicate with the PPE during the final stage for storing results. Also, the various stages of this algorithm are popular scientific kernels that have been proven to give high performance on the Cell architecture [36, 22, 1, 9].

5. Performance Results

We report our performance results from actual runs on a IBM BladeCenter QS20, with two 3.2 GHz

Table 1. Time in seconds to generate 100 million random samples in sequential and block pattern on various architectures. For the Cell/B.E. our timings are from a single chip. The performance results on the Intel, AMD and IBM PowerPC processors are from Saito and Matsumoto [32].

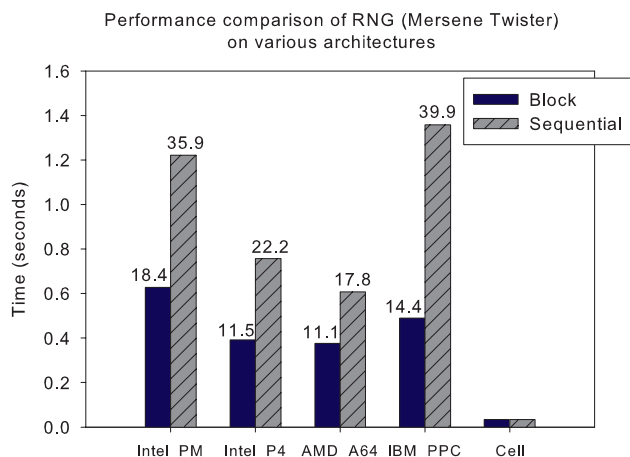| CPU/Compiler	Output	MT	MT(SIMD)
Intel Pentium-M 1.4 GHz	block	1.122	0.627
Intel C/C++ v9.0 [32]	seq	1.511	1.221
Intel Pentium-4 3.0 GHz	block	0.633	0.391
Intel C/C++ v9.0 [32]	seq	1.014	0.757
AMD Athlon 64 3800+	block	0.686	0.376
2.4 GHz, gcc v4.0.2 [32]	seq	0.756	0.607
IBM PowerPC G4 1.33 GHz	block	1.089	0.490
gcc v4.0.0 [32]	seq	1.794	1.358
IBM Cell/B.E. 3.2 GHz	block	-	0.034
xlc	seq	-	0.036

Figure 6. Comparison of running times to generate 100 million random samples in sequential and block pattern on various architectures as reported in Table 1. The number above each bar represents the speedup of Cell/B.E. as compared with the corresponding architecture.

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.

Table 1 lists the running time of our 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 [32]. Fig. 6 plots the performance and reports speedup of our Cell optimized implementation (using one Cell/B.E. processor) as compared to the corresponding architecture. 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.

We use different combinations of random number generators and normalization techniques to compare performance across several platforms. For the remainder of this section we use both of the Cell/B.E. processors available on the blade for measuring performance. Table 2 gives a performance comparison of option pricing using Monte Carlo simulation with these architectures. The performance column reports the number of Monte Carlo experiments that can be performed per second. *EOP-BMP* demonstrates the performance of our implementation that uses Box Mueller in Polar form and *EOP-BMC* uses the Box Mueller in Cartesian form.

For *CUDA-BMC* we use an optimized implementation by Podlozhnyuk [31], that is based on the CUDA Software Development Toolkit, to show performance comparisons with NVIDIA G80. The code generates an array (domain set) of random samples, normalizes the array and uses that for pricing many options. For performance comparisons we aggregate the running time of all stages for pricing a single option. Our Cell optimized implementation (*EOP-BMC*) achieves

Table 2. Performance comparison of option pricing using Monte Carlo simulation with other architectures. Mersenne Twister is used as the pseudo-random number generator.

Version	Platform	Performance	Transformation	Software
EOP-BMP (this paper)	Cell/B.E.	**1040M/s**	Box Mueller /Polar form	Cell SDK 2.1
IBM SDK Sample [17]	Cell/B.E.	190M/s	Box Mueller /Polar form	Cell SDK 2.1
RMCell-BMP	Cell/B.E.	605M/s	Box Mueller /Polar form	RapidMind SDK 2.1
EOP-BMC (this paper)	Cell/B.E.	**1824M/s**	Box Mueller /Cartesian form	Cell SDK 2.1
CUDA-BMC [31]	NVIDIA G80 (GPU)	1209M/s	Box Mueller /Cartesian form	CUDA SDK 1.0

Table 3. Performance comparison of option pricing using quasi-Monte Carlo simulation with other architectures. Hammersley sequence is used as the quasi-random number generator.

Version	Platform	Performance	Transformation	Software
EOP-LDM (this paper)	Cell/B.E.	**1770M/s**	Low Distortion Map (LDM)	Cell SDK 2.1
RMCell-LDM	Cell/B.E.	888M/s	Low Distortion Map (LDM)	RapidMind SDK 2.1
RMGPU-LDM	NVIDIA G80 (GPU)	1400M/s	Low Distortion Map (LDM)	RapidMind SDK 2.1

a speedup of 1.51 over *CUDA-BMC*. The performance number for *RMCell-BMP* is based on an implementation that uses the RapidMind development platform for optimizing option pricing on Cell. We change the normalization technique to optimize this code for performance comparisons. We observe that our hand-tuned code obtains a performance advantage of 1.72 as compared with using the RapidMind SDK for Cell. *IBM SDK Sample* gives the performance of an implementation of this algorithm provided as a sample with Cell SDK 2.1.

Table 3 gives a performance comparison of option pricing using quasi-Monte Carlo simulation with other architectures. *EOP-LDM* shows the performance of our Cell-optimized implementation based on Hammersley quasi-random number generator and the Low Distortion Map (LDM) transformation. *RMCell-LDM* represents the performance of the latest implementation from RapidMind Inc. of this algorithm compiled using RapidMind v2.1. *RMGPU-LDM* shows the performance of the same implementation run on NVIDIA

GeForce 8800. RapidMind's performance on Cell is within a factor of 2 as compared to our hand-tuned implementation, and we obtain a speedup of 1.26 as compared to *RMGPU-LDM*.

6. Conclusion

We use Monte Carlo techniques to design efficient parallel algorithms for European Option and Collateralized Debt Obligation pricing. To achieve high performance, we design, analyze and optimize different high performance pseudo (such as Mersenne Twister) and quasi (such as Hammersley sequence) random number generators as well as normalization techniques, while maintaining high accuracy. Our Cell-optimized EO pricing attains a speedup of 1.51 over NVIDIA GeForce 8800 (using CUDA), a speedup of over 2 as compared to using RapidMind SDK for Cell and a speedup of 1.26 as compared to using RapidMind SDK for GPU. We also present the design of a parallel CDO pricing algorithm. Our detailed analyses and performance re-

sults suggest that the IBM Cell/B.E. is well suited for financial workloads, and Monte Carlo simulation provides high scalability among the SPEs.

Acknowledgments

References

[1] D.A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proc. 14th Int'l Conf. on High Performance Computing (HiPC 2007)*, Goa, India, December 2007.

[2] F. Black and M. Scholes. Monte Carlo methods for solving multivariate problems. *The Journal of Political Economy*, 81(3):637–654, 1973.

[3] 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.

[4] G. Campolieti and R. Makarov. Parallel lattice implementation for option pricing under mixed state-dependent volatility models. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 170–176, Washington, DC, USA, 2005. IEEE Computer Society.

[5] 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/, November 2005.

[6] L. S. Goodman D. J. Lucas and F. J. Fabozzi. *Collateralized Debt Obligations: Structures and Analysis*. John Wiley & Sons, 2006.

[7] D. Darrell and G. Nicolae. Risk and valuation of collateralized debt obligations. *Financial Analysts Journal*, 57(1):41–59, 2001.

[8] B. Flachs and *et al.* 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.

[9] B. Gedik and R. Bordawekar. CellSort: High Performance Sorting on the Cell processor. In *Proc. VLDB*, 2007.

[10] A. V. Gerbessiotis. Architecture independent parallel binomial tree option price valuations. *Parallel Comput.*, 30(2):301–316, 2004.

[11] J. H. Halton. On the efficiency of certain quasirandom sequences of points in evaluating multidimensional integrals. *Num. Math.*, 2(1):84–90, 1960.

[12] J. Hammersley. Monte Carlo methods for solving multivariable problems. In *Proc. of the New York Academy of Science*, volume 86, pages 844–874, 1960.

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

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

[15] HPCWire. Wall Street-HPC Lovefest; Intel's Fall Classic. White paper, September, 2007.

[16] K. Huang and R. K. Thulasiram. Parallel algorithm for pricing American Asian options with multi-dimensional assets. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 177–185, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

[19] C. Jacobi, H.-J. Oh, K.D. Tran, S.R. Cottier, B.W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano. The vector floating-point

unit in a synergistic processor element of a Cell processor. In *Proc. 17th IEEE Symposium on Computer Arithmetic*, pages 59–67, Washington, DC, USA, 2005. IEEE (ARITH '05) Computer Society.

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

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

[22] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113, New York, NY, USA, 2006. ACM Press.

[23] D. X. Li. On default correlation: A copula function approach. *Journal of Fixed Income*, 9(4):43–54, 2000.

[24] J. X. Li and G. L. Mullen. Parallel computing of a quasi-Monte Carlo algorithm for valuing derivatives. *Parallel Comput.*, 26(5):641–653, 2000.

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

[26] 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.

[27] N. Metropolis and S. Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[28] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[29] G. Pauletto. Parallel Monte Carlo methods for derivative security pricing. In *NAA '00: Revised Papers from the Second International Conference on Numerical Analysis and Its Applications*, pages 650–657, London, UK, 2001. Springer-Verlag.

[30] D. Pham and *et al.* 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.

[31] V. Podlozhnyuk. Monte Carlo Option pricing. (NVIDIA CUDA) White paper, v1.0, June, 2007.

[32] M. Saito and M. Matsumoto. Simple and Fast MT: A Two times faster new variant of Mersenne twister. In *Proc. 7th Intl. Conference on Monte Carlo Methods in Scientific Computing*, Germany, 2006.

[33] Securities Industry and Financial Markets Association (SIFMA). Global CDO issuance data, 2007. `http://www.sifma.org/research/global-cdo.html`.

[34] P. Shirley and K. Chiu. A low distortion map between disk and square. *Journal of graphics tools*, 2(3):45–52, 1997.

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

[36] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the Cell processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.

[37] W. Zhu, P. Thulasiraman, R. K. Thulasiram, and G. R. Gao. Exploring financial applications on many-core-on-a-chip architecture: A first experiment. In *ISPA Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 221–230. Springer, 2006.