Pricing American options with least squares Monte Carlo on GPUs

MASSIMILIANO FATICA, NVIDIA, Santa Clara, CA, United States

EVERETT PHILLIPS, NVIDIA, Santa Clara, CA, United States

Pricing American Options with Least Squares Monte Carlo on GPUs

Massimiliano Fatica and Everett Phillips
NVIDIA Corporation
Santa Clara, CA 95050

ABSTRACT

This paper presents an implementation of the Least Squares Monte Carlo (LSMC) method by Longstaff and Schwartz [1] to price American options on GPU using CUDA. We focused our attention to the calibration phase and performed several experiments to assess the quality of the results. The implementation can price a put option with 200,000 paths and 50 time steps in less than 10 ms on a Tesla K20X.

1. INTRODUCTION

Pricing American style options is very important since these financial instruments are found in all financial markets. American options can be exercised at any time between the present date and the time to maturity. Europen options, on the other hand, can only be exercised at maturity. The early exercise feature is what makes their simulation and pricing challenging, the holder will exercise the option as soon as it is more profitable to do so rather than wait until expiration. While there are some analytical approximations for particular classes of early exercise options, there are no general closed form solutions and numerical simulation is the only way to accurately price them.

Algorithms for American option pricing typically belong to two categories. Grid based algorithms (finite differences or binomial/trinomial trees), where a discrete grid in space and time is generated and iterated backward in time, belong to one category. In the other category we found Monte Carlo algorithms. The Longstaff-Schwartz algorithm [1] is one of the most popular Monte Carlo methods for the pricing of American style options on more than one underlying asset. These simulations are quite computationally demanding.

The use of Graphics Processing Units (GPU) in high performance computing is becoming very popular due to the high computational power and high memory bandwidth of these devices coupled with the availability of programming languages like CUDA C [3] and tools (from libraries to debuggers and profilers). For most problems in computational finance, there is the need to improve the time to solution and to increase the accuracy of the discretization. By using GPUs, it is possible to address both requirements, as we will show in this paper.

There are several papers on option pricing on GPU ([4, 5, 6]), but this is the first one that implements the standard Least Squares Monte Carlo (LSMC) method by Longstaff and Schwartz. This method, at first sight, seems very unfriendly to such massively parallel architectures, but as we will show in this paper, it is possible to efficiently map it to GPUs and achieve very high performance.

WHPCF '13 Denver, Colorado USA
Copyright 2013 ACM 978-1-4503-2507-3/13/11 ...$15.00
http://dx.doi.org/10.1145/2535557.2535564 .

2. LEAST SQUARE MONTE CARLO ALGORITHM

In a standard Monte Carlo method, the paths of the state variables are simulated forward in time. Given a predetermined exercise policy, a price is determined for each path. A unbiased estimate of the option price is computed by taking the average of these prices. On the other hand, the pricing methods for American options are generally backward in time. The optimal exercise price is determined at maturity and then by recursively propagating it backward in time using dynamic programming, the price at the current time is estimated.

During the simulation, the exercise times are restricted to a fixed set of times. The objective is to provide an approximation of the optimal stopping rule that maximizes the value of the American option.

For $t = T$, the holder will exercise the option if it is in the money. For any other time t_k, the holder needs to choose whether to exercise or to continue to hold. The value of the option is maximized if the exercise happens as soon as the immediate exercise cash flow is greater than or equal to the continuation value. The continuation value at time t_k is not known and needs to be estimated. In the seminal paper by Longstaff and Schwartz[1], the continuation value C is estimated by ordinary least-squares regression using a cross section of simulated data. Since the decision to exercise the option is only relevant when the option is in-the-money, Longstaff and Schwartz [1] regress only paths that are in-the-money. This choice improves the efficiency of the algorithm.

If N is the number of paths and M the number of time intervals in each path, the basic steps for the LSMC algorithm are:

1. Generate a matrix Z(N,M) of normal random numbers
2. Compute the underlying asset prices S(N,M+1)
3. Compute the value of the option at time T, since the

exercise policy is known.

4. For each time step t_k, going backward from $t = T - \Delta t$ ($k = M$) to $t = \Delta t$ ($k = 2$):
   - 4.1 Select the paths that are in the money
   - 4.2 Compute the matrix A and right hand side b of the least square system to approximate the continuation function. This involves the asset prices at time k and the cash flow at time $k+1$
   - 4.3 Compare the value for immediate pay off and continuation value and decide if early exercise
5. Discount the cash flow to time $t = 0$ ($k = 1$) and average over the paths.

2.1 Selection of the basis functions

In the regression model, the continuation function is approximated as a linear combination of basis functions:

$$F(., t_k) = \sum_{k=0}^{p} \alpha_k L_k(S(t_k)) \qquad (1)$$

Simple powers of the state variable and several orthogonal polynomial familes (Laguerre, Hermite, Legendre and Chebyshev) have been suggested. In the original paper, simple monomial:

$$L_k(S) = S^k$$

or weighted Laguerre functions:

$$\begin{aligned} L_0(S) &= e^{-S/2} \\ L_1(S) &= e^{-S/2}(1-S) \\ L_2(S) &= e^{-S/2}(1-2S+S^2/2) \\ L_k(S) &= e^{-S/2}\frac{e^S}{k!}\frac{d^k}{dS^k}(S^k e^{-S}) \end{aligned}$$

were proposed.

The LSMC algorithm is quite robust to the choice of the basis functions. In our code we have implemented both the monomial and the weighted Laguerre polynomials.

3. GENERATING RANDOM NUMBERS ON THE GPU

For the random number generation we relied on CURAND, the library included in the CUDA Toolkit[3]. The basic operations needed in CURAND to generate a sequence of random numbers are:

- Create a generator using *curandCreateGenerator()*
- Set a seed with *curandSetPseudoRandomGeneratorSeed()*
- Generate the data from the required distribution. The available distributions are: uniform, normal or log-normal.
- Destroy the generator with *curandDestroyGenerator()*

In CUDA 5.5, there are 4 random number generators included in the CURAND library:

1. XORWOW, xor-shift added with Weyl sequence introduced by Marsaglia[7].
2. MTGP32, a member of the Mersenne Twister family of pseudorandom number generators with parameters customized for operation on the GPU (Saito and Matsumoto [8])
3. MRG32K3A, a member of the Combined Multiple Recursive family of pseudorandom number generators.
4. PHILOX4-32, a *counter-based* parallel RNG introduced by Salmon et al. [9].

We generated the normal distribution needed for the path generation in two ways: calling directly the normal distribution generator or calling a uniform distribution generator and then applying a Box-Muller transform to obtain a normal distribution. We used the original formulation of the Box-Muller transform to avoid branching. If u_0 and u_1 are two samples from the uniform distribution, by applying:

$$n_0 = \sqrt{-2\log(u_1)}\sin(2\pi u_0)$$

$$n_1 = \sqrt{-2\log(u_1)}\cos(2\pi u_0)$$

we generate two numbers n_0 and n_1 with normal distribution.

For all the simulations, we used the CURAND host API, where the library calls happen on the host but the actual work of random number generation occurs on the device. The resulting random numbers are stored in global memory on the device. This is very convenient when pricing multiple options, the random number generation is done only once and the random data are used multiple times.

Some generators have an optional driver API, where the generation could be embedded in kernels running directly on the device. While this could result in a lower memory footprint, it makes applying techniques like antithetic variables or moment matching harder. The random number generation inside the kernels could also increase register pressure and result in reduced performance. Also, the main focus of the paper is not on the random number or path generation, but on the regression phase.

3.1 Moment matching

The code has the option to preprocess the data for moment matching. Since we are working with a finite sample of the whole population, we may have some sample bias. We can match the desidered mean and variance for a standard normal distribution by calculating μ, the sample mean, and σ, the standard deviation of the generated sequence and by scaling each random variable by:

$$n_i^* = \frac{(n_i - \mu)}{\sigma}$$

The new variable n^* has a mean and standard deviation matching the desired distribution.

4. PATH GENERATION

The underlying stock price, $S(t)$ is assumed to follow a geometric Brownian motion. If $S0$ is the initial price, r is the interest rate, σ the stock price volatility, for each path the evolution of the stock price over a sequence of time steps $0 = t_0 < t_1 < ... < t_M = T$ is given by the formula:

$$\begin{aligned} S_i(0) &= S0 \\ S_i(t+\Delta t) &= S_i(t)e^{(r-\frac{\sigma^2}{2})\Delta t + \sigma\sqrt{\Delta t}Z_i} \end{aligned}$$

4.1 Antithetic variables

Standard Monte Carlo methods have a convergence to the solution that is proportional to the inverse square root of the number of samples N. The antithetic variance reduction technique reduces variance by introducing a negative or "antithetic" relation between pairs of paths. When working with normally distributed random variables, the variables Z and $-Z$ form an antithetic pair. The path generated with $-Z$ simulates the reflection about the origin of the path generated with Z. The advantage of this technique is twofold: it reduces the number of samples needed to generate N paths, and it reduces the variance of the sample paths, improving the accuracy. For each path S_i:

$$S_i(t+\Delta t) = S_i(t)e^{(r-\frac{\sigma^2}{2})\Delta t+\sigma\sqrt{\Delta t}Z_i}$$

there is a correspondent antithetic path S^*:

$$S_i^*(t+\Delta t) = S_i^*(t)e^{(r-\frac{\sigma^2}{2})\Delta t-\sigma\sqrt{\Delta t}Z_i}$$

This technique is also ensuring the first-moment matching, since we are effectly using a new sequence with elements having a symmetry across the origin.

5. CUDA IMPLEMENTATION

The complete algorithm has been implemented on GPU using CUDA C and the CURAND library. The only data transferred to the GPU are the input parameters for the simulation, all the rest is generated on the GPU. At the end of the computation the mean value is transferred back to the CPU.

The path generation is implemented with a very simple kernel in which each thread is assigned one or more paths and than iterates in time to generate the full path. The current simple implementation is very fast, it can generate 100,000 paths with 50 time points in half a microsecond.

The mean and standard deviation computations, needed for both the moment matching and the final pricing, are implemented with a single pass kernel using an atomic lock, as proposed by Giles[2].

The generation and solution of the Least Squares system in the regression phase is the main contribution of this work. The regression phase for each time step is typically implemented on traditional CPU in the following way:

- If ITM denotes the number of paths in the money at time t, a new vector $S'(t)$ of dimension ITM is selected from the original asset vector $S_i(t)$ of dimension N.
- The matrix A for the Least Squares system is constructed from the vector $S'(t)$ using a particular set of basis functions as explained in 2.1. This matrix is of dimensions (ITM,p+1).
- The right hand side b is constructed from the discounted cash-flow at time $t+1$ of the paths in the money at time t. Also this vector has size ITM.
- The system $Ax = b$ is then solved with a QR or SVD method. The solution will provide the $p+1$ coefficients of the linear combination of basis functions (Eq. 1).
- For all the paths in the money, the continuation value is computed with Eq. 1 and the cash flow at time t is set as the maximum of the immediate exercise and the continuation value.

For the GPU implementation, our approach is based on the normal equation. Multiplying both sides of $Ax = b$ by A^T, we build a new system $A^TAx = A^Tb$. A^TA is a matrix of dimension (p+1,p+1) and A^Tb is a vector of size p+1.

Instead of selecting the paths in the money (operation that could be performed with a stream compaction algorithm), forming the matrix A and the right hand side b with this subset, computing A^TA and A^Tb, and then solving the normal equation, we merge all these steps.
The element (l, m) of A^TA, with $l = 0, .., p$ and $m = 0, .., p$ can be expressed in term of the basis functions as:

$$\sum_{j\in ITM} L_l(j)L_m(j)$$

while the element l of A^tb is:

$$\sum_{j\in ITM} L_l(j)b(j)$$

In the first stage each block selects the paths in the money and adds their contribution to two partial sums, one for A^TA and one for A^Tb. The matrix A is never formed, the only inputs are the asset vector S(t), needed to compute the basis functions, and the discounted cash-flow vector at time $t+1$. The second stage performs the final reductions with a single block and once the final matrix A^TA and the the right hand side A^Tb are in shared memory, it solves the system and determine the coefficients for the linear regression. A third kernel computes the conditional continuation value and decides if exercise immediately or keep waiting. These steps could also be combined in a single kernel, but we preferred to have them in three different kernels to keep the code simpler and to experiment with different approaches (for example, solve the system using extended precision).

Since we are solving the least squares problem with a normal equation approach, we estimate the conditional expectation function in a renormalized space, dividing the stock price by the initial or the strike price. This will eliminate underflow in the weighted Laguerre and also avoid overflow when using a high number of basis functions. It will also reduce the condition number of the matrix, fact that is particularly important since we are using a normal equation that squares the condition number of original system The first kernel can also perform the partial sums using a compensated algorithm to improve the accuracy.

Since the amount of memory on the GPU is currently limited to $6GB$, we optimized the code memory footprint. The code is using two matrices, one for the random numbers of dimension $N \times M$ and one for the paths of dimension $N \times (M+1)$. We only stores two cashflow vectors, one for the previous time step $(k+1)$ and one for the current time step (k) and swap them. The total amount of memory required is $N \times (2M+3) \times 8$ bytes, assuming that all the variables are stored in double precision. We also implemented a version of the code where the random numbers and the paths are stored in single precision.

6. RESULTS

In this section, we will analyze the different steps of the algorithm. The results have been obtained on a Tesla K20X GPU, a card of the Kepler generation with 6GB of memory. The complete algorithm is implemented on the GPU using CUDA.

6.1 Performance of the random number generators

We analyzed the performance and quality of the random number generators. For each generator, we performed the simulation calling directly the normal distribution or by applying the Box-Muller transform to a uniform distribution.

In general, it is sligthly faster to generate the uniform distribution and then apply the Box-Muller transform than to call directly the normal distribution. The only exception is the MTGP32 generator, for which CURAND uses internally an inverse cumulative distribution function to generate the normal distribution from uniform data. It is also important to mention that XORWOW spends most of the time in the generation of the seeds. Changing the way in which the random numbers are stored in global memory by setting the order to *CURAND_ORDERING_PSEUDO_SEEDED* will bring the execution time down to 3.2ms, more in line with the other generators. This seeding method reduces state setup time but may result in statistical weaknesses of the output for some seed values.

Generator	Distribution	Time (ms)	Mean	Std dev
XORWOW	Normal	12.99	2.5449e-04	9.9934e-01
XORWOW	Unif.+BM	12.65	8.3393e-05	9.9966e-01
MTGP32	Normal	3.48	5.5573e-05	9.9984e-01
MTGP32	Unif.+BM	3.92	4.8088e-04	9.9980e-01
MRG32K	Normal	4.46	5.0794e-05	1.0001e+00
MRG32K	Unif.+BM	4.02	-1.4576e-04	1.0001e+00
PHILOX	Normal	2.89	1.5379e-04	9.9955e-01
PHILOX	Unif.+BM	2.53	1.0618e-05	9.9945e-01

Table 1: CURAND performance and quality of the random numbers for $N = 10^7$ samples on a Tesla K20X for different generators.

The statistical quality of the random number is in line with expectations (the mean of N normals is zero on average but has a standard deviation of $1/\sqrt{N}$, so for $N = 10^7$ it should be close to 3×10^{-4}).

If we increase the sample size by an order of magnitude as shown in table 2, we see that while MTGP32 and PHILOX run times scale linearly with the number of paths, for the XORWOW and MRG32K the scaling is sublinear.

Generator	Distribution	Time (ms) $N=10^7$	Time (ms) $N=10^8$
XORWOW	Normal	12.99	34.03
XORWOW	Unif.+BM	12.65	30.93
MTGP32	Normal	3.48	32.95
MTGP32	Unif.+BM	3.92	37.53
MRG32K	Normal	4.46	26.44
MRG32K	Unif.+BM	4.02	22.02
PHILOX	Normal	2.89	27.40
PHILOX	Unif.+BM	2.53	24.12

Table 2: CURAND performance for $N = 10^7$ and $N = 10^8$ samples on a Tesla K20X for different generators.

By applying moment matching, we can bring the mean close to roundoff (10^{-17}) and the standard deviation to 1. The moment matching requires two kernels, a first one to compute the mean and standard deviation and a second one to apply the scaling. The additional time to run these two kernel is 1.88ms, .08ms for the first one and 1.04ms for the second one with $N = 10^7$ points.

6.2 Option value

Table 3 compares the original results from the Longstaff and Schwartz paper with the current GPU implementation. The finite difference results are from an implicit scheme with 40,000 time steps per year and 1,000 step for the stock price. The GPU results are with the PHILOX generator with a random seed and 4 terms in the monomial expansion.

S	σ	T	Finite Difference	Longstaff paper	GPU result
36	.20	1	4.478	4.472	4.473
36	.20	2	4.840	4.821	4.854
36	.40	1	7.101	7.091	7.098
36	.40	2	8.508	8.488	8.501
38	.20	1	3.250	3.244	3.248
38	.20	2	3.745	3.735	3.746
38	.40	1	6.148	6.139	6.138
38	.40	2	7.670	7.669	7.663
40	.20	1	2.314	2.313	2.309
40	.20	2	2.885	2.879	2.877
40	.40	1	5.312	5.308	5.305
40	.40	2	6.920	6.921	6.909
42	.20	1	1.617	1.617	1.616
42	.20	2	2.212	2.206	2.204
42	.40	1	4.582	4.588	4.578
42	.40	2	6.248	6.243	6.233
44	.20	1	1.110	1.118	1.112
44	.20	2	1.690	1.675	1.684
44	.40	1	3.948	3.957	3.944
44	.40	2	5.647	5.622	5.627

Table 3: Comparison of the finite difference and simulation values. The strike price for the put is 40, the interest rate is 0.06. The simulation from Longstaff paper and the GPU results are with 100000 paths and 50 time steps per year.

6.3 Effect of the number of basis function

We have studied the effect of increasing the number of basis function on both the runtime and accuracy. We went from 2 to 16 terms in the expansion. As we can observe from Table 4, the result is quite accurate starting with 4 terms. The execution time, even increasing the number of function from 2 to 16, only increase by a factor of two.

6.4 Comparison with QR solver

To check the quality of the results from the normal equation approach, we have compared them with the results coming from a QR solution on the CPU. At each time step, we

Basis functions	Value	Time (ms)
2	4.416784418196301	5.614208
3	4.467512662141085	6.933216
4	4.476203385666065	7.421696
5	4.477696553155241	7.648160
6	4.478052613348710	8.013440
7	4.477953374455391	7.490336
8	4.477901680786058	7.791392
9	4.478551179372646	8.410880
10	4.478770739443920	9.278496
11	4.479134192770500	9.646816
12	4.479558727600991	8.940992
13	4.478892208874470	9.844864
14	4.479228441448461	10.778688
15	4.479248510520764	11.077408
16	4.478665388251865	10.968608

Table 4: Results for a put option with strike price=40, stock price=36, $\sigma = .2$, $r = .06$ and $T = 1$. The reference value is 4.478. The number of paths is $204,800$ with 50 time steps. The time reported is for the least square portion of the algorithm.

transfer the column of the stock price S(:,t) and cash flow CF(:,t+1) to the CPU, select the paths in the money, build the matrix A and the right hand side and solve the system using a QR solver. Table 5 reports the final price varying the number of basis functions:

While we can see small differences in the solutions, for example when using 4 basis functions, the coefficents α_i on GPU at the last stage are

```
{155.156982160074, -397.156557357517,
 353.391768458585, -108.545827892269}
```

versus

```
{155.157227263422, -397.157372674635,
 353.392671772303, -108.546161230726}
```

on the CPU, there is pretty much no effect on the final price. Only for a large number of basis functions there is a small difference in the final value. This high count is quite unusual to price a single option, when a large number of basis functions is used they are usually related to different assets or to control variates.

6.5 Single vs. Double precision

The single precision performance of the GPU is quite remarkable, so it is desiderable to use single precision variables as much as possible. In our case, the use of single precision will also benefit the memory footprint, doubling the size of the maximum solvable problem. It is important to note that CURAND generates two complete different sequences when using single vs double precision variables, so there will be a difference in the results, similar to the case when two different seeds are used. The two different implementations can also help us to assess the kernels that are bandwidth limited versus the kernel that are compute limited.

When we run the code through the profiler *nvprof* using single precision, we will get a similar output:

Basis	Normal equation	QR
functions	(GPU)	(CPU)
2	4.740095193796793	4.740095193796793
3	4.815731393932048	4.815731393932048
4	4.833172186198728	4.833172186198728
5	4.833251309474664	4.833251309474664
6	4.835805059904685	4.836251721596485
7	4.837584550853037	4.837803730367345
8	4.838283073214879	4.839358646526560

Table 5: Comparison of the normal equation approach on GPU with the results from a classical QR on CPU for a put option with strike price=40, stock price=36, $\sigma = .2$, $r = .06$ and $T = 2$. The reference value is 4.840.

```
nvprof ./american_sp -g3
American put option N=524288 (LDA=524288)  M=50 dt=0.020000
Strike price=40.000000 Stock price=36.000000 sigma=0.200000 r=0.060000 T=1.000000

Generator: MRG
BlackScholes put = 3.844
Normal distribution
RNG generation time = 5.920544 ms
Path generation time = 1.882912 ms
LS time = 6.319168 ms, perf = 162.617 GB/s
GPU Mean price  =4.475582e+00


 Time     Calls   Avg       Min       Max      Name
 3.5837ms   1  3.5837ms  3.5837ms  3.5837ms  gen_sequenced<curandStateMRG32k3a
 3.0940ms  49  63.142us  61.984us  64.256us  tall_gemm
 2.2367ms  49  45.646us  44.608us  46.688us  second_kernel
 1.9065ms   1  1.9065ms  1.9065ms  1.9065ms  generate_seed_pseudo_mrg
 1.8345ms   1  1.8345ms  1.8345ms  1.8345ms  generatePath
 505.38us  49  10.313us  10.144us  10.656us  second_pass
 127.71us   1  127.71us  127.71us  127.71us  redusum
 12.544us   3  4.1810us  3.8080us  4.3840us  [CUDA memset]
 7.8080us   1  7.8080us  7.8080us  7.8080us  BlackScholes
 5.7280us   2  2.8640us  2.7840us  2.9440us  [CUDA memcpy DtoH]
```

If we compare this output with the one obtained from the code where all the variables are in double precision:

```
nvprof ./american_dp -g3
American put option N=524288 (LDA=524288)  M=50 dt=0.020000
Strike price=40.000000 Stock price=36.000000 sigma=0.200000 r=0.060000 T=1.000000

Generator: MRG
BlackScholes put = 3.844
Normal distribution
RNG generation time = 8.763488 ms
Path generation time = 3.288832 ms
LS time = 7.512192 ms, perf = 136.792 GB/s
GPU Mean price  =4.476522e+00


Time     Calls   Avg       Min       Max      Name
6.4127ms    1  6.4127ms  6.4127ms  6.4127ms  gen_sequenced<curandStateMRG32k3a
3.4666ms   49  70.746us  69.632us  71.680us  second_kernel
3.2417ms    1  3.2417ms  3.2417ms  3.2417ms  generatePath
3.0826ms   49  62.909us  61.856us  63.840us  tall_gemm
1.9224ms    1  1.9224ms  1.9224ms  1.9224ms  generate_seed_pseudo_mrg
480.03us   49  9.7960us  9.5360us  10.208us  second_pass
126.24us    1  126.24us  126.24us  126.24us  redusum
12.384us    3  4.1280us  3.7760us  4.3200us  [CUDA memset]
11.936us    1  11.936us  11.936us  11.936us  BlackScholes
5.7920us    2  2.8960us  2.8480us  2.9440us  [CUDA memcpy DtoH]
```

we can infer that the first phase of the Least Square is more compute limited (it takes the same time when the input arrays are in single or in double precision). Other kernels like the path generation or the computation of the continuation value are bandwidth limited. From the profiler output, that reports the average time plus the minimum and maximum, we can also see that the divergence caused by the selection of the paths in the money has a minimal effect on performances.

7. CONCLUSION

This paper presents a GPU implementation of the Least Square Monte Carlo method. The whole algorithm has been implemented on the GPU using CUDA C resulting in a significant speed-up while still maintaining high accuracy. On a Tesla K20x, we can simulate an American put option with 200,000 paths and 50 time steps in less than $10ms$. Even on a laptop, with a GPU with low double precision performance, the simulation completes in $200ms$.

8. REFERENCES

[1] F.A. Longstaff and E.S. Schwartz, *Valuing American Options by Simulation: A Simple Least-Squares Approach*, Review of Financial Studies vol. 14 ,2001.

[2] M. Giles, *Reduction and Scan Operation*, http://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec4.pdf

[3] CUDA Toolkit, *http://developer.nvidia.com/cuda-toolkit*

[4] D. Egloff, *Pricing financial derivatives with high performance finite difference solvers on GPUs*, GPU Computing Gems Jade Edition, Editor W. Hwu, Morgan Kaufmann, 2011.

[5] G. Pages and B. Wilbertz, *GPGPUs in computational finance: massive parallel computing for American style options*, Concurrency and Computation: Practice and Experience, Vol. 24-8, 2011.

[6] L.A. Abbas-Turki, S. Vialle, B. Lapeyer and P. Mercier, *Pricing derivatives on graphics processing units using Monte Carlo simulation*, Concurrency and Computation: Practice and Experience, John Wiley & Sons, 2012

[7] G. Marsaglia, *Xorshift RNGs*. J. Stat. Soft., 8:1âĂŞ6, 2003.

[8] M. Saito and M. Matsumoto, *Variants of Mersenne Twister Suitable for Graphic Processors*, ACM Transactions on Mathematical Software (TOMS), Volume 39 Issue 2, 2013.

[9] J. Salmon, M. Moraes, R. Dror, D.E. Shaw, *Parallel Random Numbers: As Easy as 1, 2, 3* ,Supercomputing 11, Nov 12-18, 2011, Washington, USA.