CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2862

SPECIAL ISSUE PAPER

Pricing derivatives on graphics processing units using Monte Carlo simulation

L. A. Abbas-Turki [1,*,†], S. Vialle [2,3], B. Lapeyre [4] and P. Mercier [2]

[1] *Université Paris-Est, Laboratoire d'Analyse et de Mathématiques Appliquées, 77454 Marne-la-Vallée Cedex2, France*
[2] *SUPELEC - IMS group, 2 rue Edouard Belin, 57070 Metz, France*
[3] *AlGorille Project Team, INRIA, Vandoeuvre-les-Nancy, France*
[4] *Université Paris-Est, CERMICS, Projet MathFi, ENPC-INRIA-UMLV, 77455 Marne La Vallée Cedex 2, France*

SUMMARY

This paper is about using the existing Monte Carlo approach for pricing European and American contracts on a state-of-the-art graphics processing unit (GPU) architecture. First, we adapt on a cluster of GPUs two different suitable paradigms of parallelizing random number generators, which were developed for CPU clusters. Because in financial applications, we request results within seconds of simulation, the sufficiently large computations should be implemented on a cluster of machines. Thus, we make the European contract comparison between CPUs and GPUs using from one up to 16 nodes of a CPU/GPU cluster. We show that using GPUs for European contracts reduces the execution time by ~ 40 and diminishes the energy consumed by ~ 50 during the simulation. In the second set of experiments, we investigate the benefits of using GPUs' parallelization for pricing American options that require solving an optimal stopping problem and which we implement using the Longstaff and Schwartz regression method. The speedup result obtained for American options varies between two and 10 according to the number of generated paths, the dimensions, and the time discretization. Copyright © 2012 John Wiley & Sons, Ltd.

Received 31 December 2009; Revised 9 February 2011; Accepted 1 May 2012

KEY WORDS: Monte Carlo; GPU; parallel random number generator; pricing derivatives; European options; American options

1. INTRODUCTION AND OBJECTIVES

Monte Carlo (MC) simulation, the most widely used method in transport problems, owes its popularity in the scientific community to its three features: (1) the possibility to use MC for complex transport problems that cannot be interpreted in deterministic language; (2) the ease of implementation and parallelization; and (3) contrary to deterministic methods such as finite element or finite difference methods, MC remains efficient in a dimension greater than four, which is appropriate for systems requiring high DOFs.

In this article, it is shown that although MC is theoretically very efficient for multi-core architectures, the methods based on MC vary according to their effectiveness on these architectures. In this work, we will present the practical point of view of the pricing methods based on Monte Carlo and implemented on graphics processing units (GPUs). This practical study will provide the comparison between CPUs and GPUs on pricing the two major derivative classes found in the financial field, which are European contracts (ECs) and American contracts (ACs). As in practice, one multi-core

*Correspondence to: L. A. Abbas-Turki, Analysis and Applied Mathematics Laboratory, University Paris-Est MLV, 77454 Champs-sur-Marne, France.

†E-mail: lokman.abbas-turki@laposte.net

card is generally insufficient for the execution of high-dimensional applications within seconds; we will compare, on the same cluster, multi-core GPUs with four-core CPUs for pricing ECs. Moreover, we will explain how we can generalize this kind of cluster comparison for pricing ACs.

After the introduction of MC and its applications for pricing ECs and ACs in Section 2, in Section 3, we present two different methods of parallelizing random number generation that aim at the highest adaptability on GPUs, and we give an example for each method. In Section 4, we give details on the implementation of a typical multidimensional EC on a multi-core CPU/GPU cluster. Section 5 presents a detailed study of the accuracy of the results, the speedups and the energy consumed during the simulation of ECs. Once the concept of pricing parallelization on ECs is understood through Sections 4 and 5, we devote the final sections to pricing ACs, which is known as one of the most challenging problems in financial applications. Thus, in Sections 6 and 7, we aim at reducing the running time of ACs simulation using GPUs, and we will propose means of parallelizing it on a cluster of machines.

Before going into the detail of this work, the main specifications of the machines on which we implement our benchmark applications are as follows:

M1: is the XPS M1730 laptop composed of Intel Duo Core CPU with a clock rate of 2.50 GHz and contains 2 nVIDIA 8800 M GTX connected with SLI.

M2: is a cluster of 16 nodes. Each node is a PC composed of an Intel Nehalem CPU, with four hyperthreaded cores at 2.67 GHz, and a nVIDIA GTX285 GPU with 1 GB of memory. This cluster has a Gigabit Ethernet interconnection network built around a small DELL Power Object 5324 switch (with 24 ports). Energy consumption of each node is monitored by a Raritan DPXS20A-16 device that continuously measures the electric power consumption (in Watts) up to 20 nodes (in Watts). Then a Perl script samples these values and computes the energy (Joules or WattHours) consumed by the computation on each node and on the complete cluster (including the interconnection switch).

2. MONTE CARLO AND MULTI-CORE PROGRAMMING

This section is divided into two parts: the first part goes over the general aspects of parallelizing MC simulations and the benchmark model used. The second part gives some details on pricing ECs and ACs. Indeed, in Markovian models, pricing ACs basically adds one step to the pricing algorithm. Thus, based on what is known on ECs, we will present the problem of pricing ACs.

2.1. An introduction to Monte Carlo methods

The general MC method is articulated by two theorems that constitute the two pillars of the probability theory [1]. The first one is the Strong Law of Large Numbers (SLLN) that announces the convergence of a certain series of independent random variables that have the same distribution to a value of an integral. The second one is the Central Limit Theorem (CLT), which determines the speed of the convergence revealed by SLLN. These two classic theorems can be found, for instance, in [2].

The assumption that makes MC more attractive than other methods for GPU is the independence of the random variables. The main concern of using MC on GPUs is how to spread this independence on the stream processor units. Unlike pricing ACs, pricing ECs with MC is no more than using the result of SLLN and CLT on random functions such as those presented in Table I. In Table I, $(x)_+ = \max(x, 0)$ and $\max_T$, mean_T, respectively stand for the maximum value and the average value on the trajectory of the stock $S_t(\varepsilon)$ on the time interval $[0, T]$. On the one hand, in this article, we suppose that $\varepsilon = (\varepsilon^1, \ldots, \varepsilon^d)$ is a Gaussian vector and the coordinates $\varepsilon^1, \ldots, \varepsilon^d$ are independent. On the other hand, we denote by S_t the price of a basket of stocks $S_t^1, \ldots, S_t^d$ and to describe the behavior of each stock, we will use the following standard geometric Brownian motion:

$$S_t^i = S_0^i \exp\left[\left(r_i - d_i - \frac{\sigma_i^2}{2}\sum_{k=1}^{i}\rho_{ik}^2\right)t + \sigma_i \sum_{k=1}^{i}\rho_{ik} W_t^k\right] \tag{1}$$

DOI: 10.1002/cpe

Table I. Contracts and associated payoffs.

Name of contracts	Payoffs
Put	$(K - S_T(\varepsilon))_+$
Call	$(S_T(\varepsilon) - K)_+$
Lookback	$(\max_T S_t(\varepsilon) - S_T(\varepsilon))$
Up and out barrier	$(f(S_T(\varepsilon))1_{\max_T S_t(\varepsilon)<L})$
Floating Asian put	$(\text{mean}_T S_t(\varepsilon) - S_T(\varepsilon))_+$

which is *equal in density* to

$$S_t^i = S_0^i \exp\left[\left(r_i - d_i - \frac{\sigma_i^2}{2}\sum_{k=1}^{i}\rho_{ik}^2\right)t + \sigma_i \sum_{k=1}^{i}\rho_{ik}\sqrt{t}\varepsilon^k\right]$$

where:

S_0^i is the initial price of the asset i,
r_i is the rate of the asset i,
d_i is the dividend of the asset i,
σ_i is the volatility of the asset i,
$\sqrt{t}\varepsilon^k$ simulates the Brownian motion distribution W_t^k,
$(\rho_{ik})_{1\leqslant i,k\leqslant d}$ is a given matrix correlating the assets.

Thus, the first stage of using MC is to simulate the Gaussian distribution of ε through a set of samples ε_i. For a detailed presentation on MC in financial applications, we refer the reader to [1]. In order to parallelize pricing ECs, we implement algorithms that can be executed similarly on all the trajectories at the same time. With MC methods, the best way to perform this similarity task is to discretize the time interval then run the same tasks sequentially for the whole current trajectories at the same step of the discretization. For instance, we can simulate the log-normal evolution of the stock $S_t^i(\varepsilon)$ at each time $t_k \in [0, T]$ using the two following steps:

(1) The simulation of normal distribution variable ε_q associated with the trajectory q.
(2) The actualization of the stock value using the recurrence relation

$$S_{t_k}^{iq} = S_{t_{k-1}}^{iq} \exp(f(\varepsilon_q))$$

where f is an affine function.

In the example demonstrated in Figure 1, we carry out the two steps sequentially. The parallelization takes part in performing the same step on different trajectories. Thus, a subset of the whole set of trajectories can be associated with one processor unit and carry out each step independently from the other subsets. Moreover, parallelizing the simulation on a cluster of multi-cores CPUs/GPUs is no more than parallelizing or enlarging the set of trajectories to add all the contributions of the different machines.

2.2. *Pricing European and American options*

An EC is one that can be exercised only on the maturity T, unlike AC, which can be exercised anytime before the maturity T. Among ECs and ACs, the 'options' contracts are those that are the most studied. The option payoff is generally given using the function $(x)_+ = \max(x, 0)$ which expresses the fact that options are contracts, which 'allow, without obligation', to buy or to sell an asset at a fixed price. For example, the put and call contracts given in Table I are options.

If r is the risk neutral rate and $\Phi(S_t)$ the payoff of a given contract, the price of a European version of this contract, at each time $t \in [0, T]$, is defined by the following expression:

$$P_t^{\text{Euro}}(x) = \mathbb{E}_{t,x}\left(e^{-r(T-t)}\Phi(S_T)\right), \tag{2}$$

DOI: 10.1002/cpe

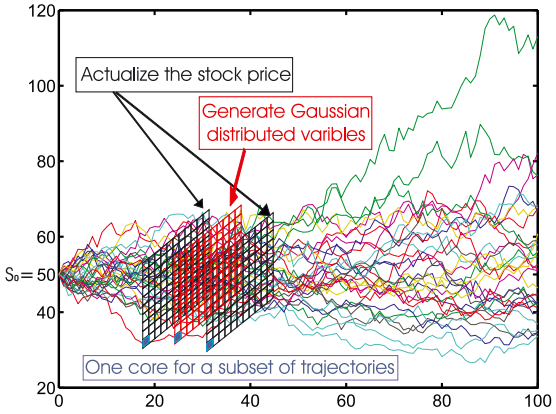

Figure 1. Parallelizing the same task on different trajectories.

where $\mathbb{E}_{t,x}$ is the expectation associated to the risk neutral probability knowing that $S_t = x$.

Using the previous notations, ACs can be exercised at any trading date until maturity and their prices are given, at each time t, by

$$P_t^{\text{Amer}}(x) = \sup_{\theta \in \mathcal{T}_{t,T}} \mathbb{E}_{t,x}\left(e^{-r(\theta - t)}\Phi(S_\theta)\right), \tag{3}$$

where $\mathcal{T}_{t,T}$ is the set of stopping times in the time interval $[t, T]$.

To simulate (3), we first need to approach stopping times in $\mathcal{T}_{t,T}$ with stopping times taking values in the finite set $t = t_0 < t_1 < \ldots < t_n = T$. When we do this approximation and use the dynamic programming principle [1], we obtain the following induction for each simulated path:

$$\begin{aligned} P_T^{\text{Amer}}(S_T) &= \Phi(S_T), \quad \forall k \in \{n-1, \ldots, 0\}, \\ P_{t_k}^{\text{Amer}}(S_{t_k}) &= \max\{\Phi(S_{t_k}), C(S_{t_k})\} \end{aligned} \tag{4}$$

$C(S_{t_k})$ in (4) represents the continuation value and is given by

$$C(S_{t_k}) = \mathbb{E}\left(e^{-r(t_{k+1} - t_k)} P_{t_{k+1}}(S_{t_{k+1}}) \middle| S_{t_k}\right). \tag{5}$$

Thus, to evaluate the price of (3), we need to estimate $C(S_{t_k})$. Longstaff and Schwartz consider the stopping times formulation of (4), which allows them to reduce the bias by using the actual realized cash flows. We refer the reader to [3] for a formal presentation of the Longstaff and Schwartz Regression (LSR) algorithm.

Algorithms devoted to American pricing and based on MC, differ essentially on the way they estimate and use the continuation value (5). For example, the authors of [4] perform a regression to estimate the continuation value, but unlike [5], they use $C(S_{t_k})$ instead of the actual realized cash flows to update the price. Other methods use Malliavin Calculus [6] or quantization methods [7] for $C(S_{t_k})$ estimation. In addition to these methods based on MC, there is a profusion of algorithms for American option pricing. However, the one that is gaining widespread adoption in the financial industry is the LSR method. This widespread adoption and the fact that LSR is based on Monte Carlo simulation leads us to choose LSR implementation on GPU.

The LSR method approximates the continuation value by projecting the cash flow $P_{t_{k+1}}(S_{t_{k+1}})$, generated at t_{k+1}, on a set of functions $\psi(S_{t_k})$ that depend only on the asset price at time t_k. However, contrary to an ordinary regression method, the LSR uses the drawings satisfying $\Phi(S_{t_k}) > 0$, the 'in the money' drawings. Even if Longstaff and Schwartz give partial convergence results in their original paper [5], the authors of [3] proved the convergence and analyzed the speed of this convergence according to the number of simulated paths. This convergence analysis has been refined in [8] by studying the problems due to the degree of regression.

DOI: 10.1002/cpe

Strictly speaking, if we consider the regression vector A and we denote by $\overline{C}(S_{t_k}) = A^t \psi(S_{t_k})$ the estimated continuation value, one must find the vector A that minimizes the quadratic error

$$\left|\left| P_{t_{k+1}}(S_{t_{k+1}}) - \overline{C}(S_{t_k}) \right|\right|_{L^2}. \tag{6}$$

We can easily check that the regression vector that minimizes (6) is given by

$$A = \Psi^{-1} \mathbb{E}\left(P_{t_{k+1}}(S_{t_{k+1}}) \psi(S_{t_k}) \right), \tag{7}$$

where $\Psi = \mathbb{E}(\psi(S_{t_k}) \psi^t(S_{t_k}))$.

Consequently, once we approximate the expectations in (7) by an arithmetic average using MC, we must invert the matrix Ψ. One of the most used and most stable methods of inversion is the one based on a singular value decomposition [9]. However, this method and other methods of inversion are not efficient to parallelize on GPUs for relatively small and not sparse matrices. In the sixth section, we will explain how the GPU implementation can be used to slightly (x1.2 to x1.4) accelerate this part of the algorithm.

Without loss of generality, we use the basis $\psi(S_{t_k})$ of monomial functions to perform our regression. Also, in the case of geometric Brownian motion, the convergence study given in [8] shows that the number of polynomials $K = K_N$ for which accurate estimation is possible from N paths is $O(\sqrt{\log(N)})$. Consequently, we use monomials of degrees less than or equal to 2 for one dimension and we will use affine regression in the multidimensional simulation. Finally, we subdivide the algorithm of pricing ACs in three parts as in [10]:

(1) Paths generation (PG) phase.
(2) Regression (REG) phase.
(3) Pricing (PRC) phase.

As the 'calibration phase' [10] can be a source of confusion with the model calibration activity in finance, we preferred to rename it by the Regression phase'.

3. PARALLEL RANDOM NUMBER GENERATION FOR SINGLE INSTRUCTION MULTIPLE DATA ARCHITECTURE

The parallelization on the GPU of random number generation (RNG) is essential in GPU implementation of MC. As a matter of fact, the GPU programmer must reduce the CPU/GPU communication if he aims at a good speedup. Indeed, although we can simulate random numbers on the CPU parallel to executing other tasks on the GPU, the communication time CPU/GPU makes this solution less efficient. We also have the same communication time problem when using true random number generators (TRNG), this is why we adopt the traditional solution of using pseudo random number generators as RNGs instead of using TRNGs.

In RNG literature, we find considerable work on sequential RNGs, but much less on parallel RNGs. The authors of [11] use the Mersenne Twister (MT) generator [12] even though this generator is relatively slow on GPUs. Indeed, as it is already mentioned in [13], because the cache memory does not exist on the GPU[‡], MT presents problems caused by the multiple accesses per generator and thus per thread to the global RAM in order to serially update the large state needed by MT. The two paradigms of generating random numbers that we are going to use are suited to the GPU architecture and generally to architectures that do not possess a large cache memory. The first one is based on period splitting, and the second one is based on parametrization, which is used in the scalable parallel pseudo random number generators [14] library and which is also recommended by the authors of MT in [15]. For each parallelization method, we will give an example, and we will compare, at the end of the second subsection, these two examples to the optimized implementation of MT and the Niederreiter Quasirandom (NQ) generator given in [16]. Because our work was implemented on GPU cards that basically compute in single precision, the two examples of random generators that we are going to present in the next subsections are based on single precision. However the reader can easily extend our constructions to the double precision cards.

[‡]This work was done before the Fermi architecture that includes a cache memory.

DOI: 10.1002/cpe

Our first goal is to have an efficient random number generator for GPU architecture, which also provides sufficient good results. Thus, in the following, two methods that are proven to be sufficiently good on CPU clusters are adapted on GPU and GPU clusters.

3.1. Parallel-random number generation from period splitting of one random number generation

The simplest theoretical solution to parallelizing RNG is to split the period of a good sequential one into different random number streams. On the one hand, although we are going to split the whole period, we need to have a long one to split. For example, we cannot split a period of a standard $\sim 2^{31}$ linear congruential generator (LCG) because it considerably reduces the period of each stream. On the other hand, although we use the RAM memory of the GPU, we should limit the parameters of the RNG in order to reduce the access of each thread to this memory. To explain the method of period splitting, we are going to take the example of an RNG whose random behavior had already been studied in [17], that has a long period to split and relatively few parameters. This RNG is the combined multiple recursive generator (CMRG) given in example 4 of [17], and it is obtained from a judicious combination of two MRGs, and each MRG has the following general expression:

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} \mathrm{mod}(M)$$

The main goal of combining two MRGs is to reduce the memory storage of the past values without really compromising the quality of the random numbers. To define the different streams of CMRG, we determine the number of these streams[§], then we compute the power of the companion matrices associated to the recurrence of CMRG, which allows us to initialize the different streams at the different points of the period. Also, the length of the streams should be chosen carefully so that a vector formed by the first number from each stream, for example, should have relatively independent coordinates. For further details, we refer the reader to [18].

Because splitting the period of CMRG implies the computation of huge[¶] powers of 3×3 matrices, the operation of launching MC on an increasing number of machines can consume a considerable amount of time. As a result, even though computing the powers of matrices uses the efficient divide-and-conquer algorithm [19], we should precompute the jump-ahead matrices once and for all. Thus, the best way of implementing CMRG is as follows:

- to fix the maximum of streams associated with the maximum of multi-cores used,
- then compute the matrices of transition between streams only once before launching the application.

For instance, let us consider the companion matrices of the CMRG given in example 4 of [17]. Each matrix is associated with one MRG from the combination:

$$A_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -183326 & 63308 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -539608 & 0 & 86098 \end{pmatrix}$$

The period of the matrix A_1 is $p_1 = m_1^3 - 1$ and the period of the matrix A_2 is $p_2 = m_2^3 - 1$, which means:

$$A_1^{p_1+1} \mathrm{mod} m_1 = A_1 \mathrm{mod} m_1, \quad A_2^{p_2+1} \mathrm{mod} m_2 = A_2 \mathrm{mod} m_2$$

For example, if we want

- to associate an RNG stream with each trajectory;
- to perform the evolution of about 2^{18} trajectories by each multi-core GPU; and
- to use a maximum of 16 GPUs.

[§]Which is for instance equal to the number of trajectories simulated or to the number of processors involved in GPUs.
[¶]Proportional to the length of the period.

DOI: 10.1002/cpe

We divide the total period of the CMRG $(p_1 \times p_2)/2 \sim 2^{205}$ by $2^{18} \times 16 = 2^{22}$ to obtain $\mathrm{InitPower} = 2^{205}/2^{22} = 2^{183}$ and perform the powers:

$$A_1^{\mathrm{init}} = A_1^{\mathrm{InitPower}} \mathrm{mod} m_1, \quad A_2^{\mathrm{init}} = A_2^{\mathrm{InitPower}} \mathrm{mod} m_2$$

As shown in Figure 2, if we initialize the stream 0 with the seed vectors: $X_1^0 = (x_1^1, x_1^2, x_1^3)^T$ for the first MRG and $X_2^0 = (x_2^1, x_2^2, x_2^3)^T$ for the second MRG of the combination, the seed values associated to the ith stream are: $X_1^i = (A_1^{\mathrm{init}})^i \times X_1^0 \mathrm{mod} m_1$ for the first MRG and $X_2^i = (A_2^{\mathrm{init}})^i \times X_2^0 \mathrm{mod} m_2$ for the second MRG of the combination. Finally, the algorithm of the CMRG on a single precision architecture is detailed in full in Figure 1, page 12 of [17].

3.2. Parallel-random number generation from parameterization of random number generations

As mentioned by the authors of the Mersenne Twister RNG in [15], an acceptable way to parallelize RNGs is to parameterize them. Besides, if we consider the seeds of the RNGs as the parameters of RNGs, the method based on period splitting can also be regarded as a parametrization of these RNGs. In this article, we prefer to separate the two methods and to concentrate on the parameterizations given in [14]. One of the generators that is really efficient in implementing on double precision GPUs is the parameterized prime modulus LCG $(2^{61} - 1)$ that allows us to specify each RNG with only one parameter, which is the multiplier a of (8). According to [14], this parameterization provides about 2^{58} streams. The prime modulus LCG $(2^{61} - 1)$ is based on the following relation:

$$x_n = a x_{n-1} \mathrm{mod}(2^{61} - 1) \tag{8}$$

In [20], we use a parameterized prime modulus LCG $(2^{31} - 1)$, which is a single precision version of (8), and we implement it on single precision GPUs to compare two clusters of GPUs and CPUs. Because of its short period and its random behavior, the LCG $(2^{31} - 1)$ should be taken as a benchmark and not used for standardized applications. In Table II, we compare the effectiveness of an optimized implementation of MT and NQ given in [16] with our sufficiently optimized implementation of the CMRG detailed in the previous subsection and the parameterized LCG (PLCG) $(2^{31} - 1)$. The results presented in Table II are obtained by averaging on various simulations performed on the GPU of *M1*.

Even if NQ is not an RNG, but a quasi-random generator, which is based on a different theory from the RNG one, we consider it interesting to compare its effectiveness with RNGs. According to Table II, we remark that CMRG is about 1.8 times faster than MT, and that PLCG is about five times faster than CMRG. Nevertheless, in order to be more confident about the quality of the random numbers, *we will use the CMRG in our next applications*.

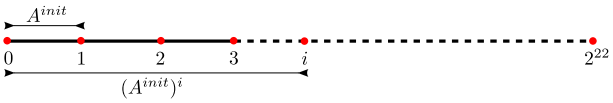

Figure 2. Splitting the period of combined multiple recursive generator.

Table II. Comparison of the effectiveness of random number generations on *M1*.

Name of the RNG	PLCG	CMRG	MT	NQ
Mega samples generated per second	16.33	3.30	1.86	1.21

RNG, random number generation; PLCG, parameterized linear congruential generator; CMRG, combined multiple recursive generator; MT, Mersenne Twister generator; NQ, Niederreiter Quasirandom generator.

DOI: 10.1002/cpe

4. MULTI-PARADIGM PARALLEL ALGORITHM AND IMPLEMENTATION

4.1. Support application

In order to explore the effectiveness of pricing ECs on a cluster of GPUs, we are going to process a typical high dimensional EC whose price depends on the whole simulation of the trajectories of the stocks' prices (*path-dependent contracts*). Here, we take the example of a homogenous Asian option in 40 dimensions; this means that our contract is an Asian option on a homogenous weighting basket of 40 stocks. We can find this kind of contract, for instance, when managing the CAC 40 index. In the financial markets, we can find other contracts on high-dimensional indices such as S&P 500, DAX 30, FTSE 100. The procedure that we are going to illustrate can be easily generalized for all European path-dependent contracts like the look back or barrier options whose payoffs are given in Table I.

The Asian option is a contract whose price depends on the trajectory average. We compute the price of a floating Asian call option using:

$$E\left[e^{-rT}(S_T(\varepsilon) - \overline{S}_T)_+\right] \tag{9}$$

$$\overline{S}_T = \text{mean}_{0 \leqslant t \leqslant T} S_t(\varepsilon) \tag{10}$$

In expressions (9) and (10), S_T represents the price of a homogenous weighting basket of 40 stocks at maturity T: $S_T = \frac{1}{40} \cdot \sum_{i=1}^{40} S_T^i$. Each stock has the log-normal distribution given in (1). Besides, according to (10) $\overline{S}_T$ represents the average price of S during the life time of the contract. The third step of Algorithm 1 introduces a recursive method for computing this average price. In Algorithm 1, the exterior time loop is used for time discretization, and in our application, we take $\delta t = T/100$. Inside the time loop, we put another loop associated to the number of stocks S^i that take part in the pricing problem. The loops on trajectories are those that we parallelize on the different stream processor units.

The third step of Algorithm 1 uses the well-known rectangle approximation of an integral. However, in order to have a faster convergence, we use in the implemented version the trapezoidal approximation, which is presented in [21] and characterized by the same implementation ease as the rectangular one.

In order to take advantage of various and heterogeneous architectures such as multi-core CPUs, GPUs, CPUs cluster and GPUs cluster, we have designed a multi-paradigm parallelization of our option pricer. First, a coarse-grained parallelization splits the problem in P_N big tasks (one per processing node), communicating by message-passing. Second, a fine grained parallelization splits each big task into some threads on a multi-core CPU, or in many light-threads on a GPU, communicating through a shared memory.

Input data files are read on processing node 0, and input data are *broadcast* to all other nodes. Then, each node locally achieves its initializations, function of the common input data and its node number. Some of these initializations have been parallelized at fine-grained level, and the parallel CMRG RNG is initialized on each node according to the specifications of Section 3.

Afterwards, each node processes its subset of MC trajectories, using its fine-grained level of parallelism. This is an *embarrassingly parallel* computing step, without any communications. Then, each node computes the sum of its computed prices and the sum of its square prices. All nodes participate in a global *reduction* of these P_N pairs of results: at the end of this step, the global sum of prices and global sum of square prices are available on node 0. Finally, node 0 computes the final price of the option and the associated error, and prints these results.

Broadcast and *reduction* are classic communication routines, efficiently implemented in the standard message passing interface (MPI) communication library [22] that we used. Conversely, reading some input files concurrently from many nodes is not always supported by a file system. So, we prefer to read input files from node 0 and to broadcast data to other nodes using an MPI routine. This strategy is highly portable and scalable.

DOI: 10.1002/cpe

Input: Model parameters and CMRG initialization
Output: $Call_{Asian} = \mathbb{E}\left(e^{-rT}(S_T(\varepsilon) - \overline{S}_T(\varepsilon))_+\right)$

for $t \in \{\delta t, 2\delta t, \ldots, T\}$ **do**
  for $i \in \{1, \ldots 40\}$ **do**
    for *each trajectory* $k \in \{1, 2, \ldots, N\}$ **do**
      `/* First step: generating a normal distributed variable using CMRG and a distribution transformation as a Box-Muller one */`
      $u_k \longleftarrow CMRG$;
      $\varepsilon_k \longleftarrow Box - Muller(u_k)$;
    end
    for *each trajectory* $k \in \{1, 2, \ldots, N\}$ **do**
      `/* Second step: price actualization during the discretized time interval` $[0, T]$ `*/`

$$S_t^i(\varepsilon_k) = S_{t-\delta t}^i \exp\left[\left(r - d_i - \frac{\sigma_i^2}{2}\sum_{k=1}^{i}\rho_{ik}^2\right)\delta t + \sigma_i \sum_{k=1}^{i}\rho_{ik}\varepsilon_k\sqrt{\delta t}\right]$$

    end
    for *each trajectory* $k \in \{1, 2, \ldots, N\}$ **do**
      `/* Third step: recursive implementation of the trajectory average using rectangle approximation */`
      $\overline{S}_t^i(\varepsilon_k) \longleftarrow ((t - \delta t)/t)\overline{S}_{t-\delta t}^i(\varepsilon_k) + (1/t)S_t^i(\varepsilon_k)$;
    end
  end
end
for *each trajectory* $k \in \{1, 2, \ldots, N\}$ **do**
  `/* Fourth step: a homogeneous weighing of 40 stocks */`
  $\overline{S}_T(\varepsilon_k) = \frac{1}{40}.\sum_{i=1}^{40}\overline{S}_T^i(\varepsilon_k)$;
  $S_T(\varepsilon_k) = \frac{1}{40}.\sum_{i=1}^{40}S_T^i(\varepsilon_k)$;
end

$$Call_{Asian} \longleftarrow \frac{1}{N}\sum_{k=1}^{N}\left(e^{-rT}(S_T(\varepsilon_k) - \overline{S}_T(\varepsilon_k))_+\right)$$

Algorithm 1: 40 Dimension Floating Asian Call

4.2. Fine-grained parallelization on the central processing unit and the graphics processing unit

The implementation on multi-core CPU clusters *M2* has been achieved using both MPI, to create one process per node and to insure the few inter-node communications, and OpenMP to create several threads per core and take advantage of each available core. The OpenMP parallelization has been optimized to create the required threads (inside a large parallel region) only once, and to load

DOI: 10.1002/cpe

balance the work among these threads. Inside each thread, data storage and data accesses are implemented in order to optimize cache memory usage. GPU implementation: Again, MPI is used to create one process per node, to distribute data and computations on the cluster and to collect results, whereas compute unified device architecture (CUDA) is used to send data and MC trajectory computations on the GPU of each node. In order to avoid frequent data transfers between CPU and GPU, we have ported our RNG on the GPU: each CUDA thread computes random numbers and all node computations are executed on the GPU. Moreover, we have minimized the accesses to the global memory of the GPU, each GPU thread uses mainly fast GPU registers. This strategy leads to a very efficient usage of the GPUs, and achieves a high speedup on GPU clusters compared with a multicore CPU cluster.

To develop the CPU cluster version, we used `g++ 4.1.2` compiler and its native and included `OpenMP` library, and the `OpenMPI 1.2.4` library. To develop the GPU cluster version, we used the `nvcc 1.1 CUDA` compiler and the `OpenMPI 1.2.4` library. All these development environments appeared compatible.

Our GPU version is composed of `.h` and `.cu` files, compiled with the following commands:

```
nvcc --host-compilation C++
     -O3 -I/opt/openmpi/include
     -DOMPI_SKIP_MPICXX -c X.cu
nvcc -O3 -L/opt/openmpi/lib
     -o pricer X.o Y.o .... -lmpi -lm
```

On our machines, the OpenMPI library is installed in the `/opt/openmpi/` directory. The `-DOMPI_SKIP_MPICXX` flag allows us to avoid the exception mechanisms implemented in the OpenMPI library (according to the MPI 2 standard), which are not supported by the `nvcc`[‖] compiler. The `-host-compilation C++` flag helps `nvcc` to understand the C++ code of the non-kernel routines.

5. CLUSTER COMPARISON FOR PRICING EUROPEAN CONTRACTS

The following subsections introduce results of three benchmark programs, implementing Algorithm 1 and computing ~ 0.25, ~ 0.5 and ~ 1 million MC trajectories (corresponding to different pricing accuracies).

5.1. The accuracy of the result

Table III represents the results of the same simulation using an increasing number of trajectories. Each one of these simulations is either done on a CPU cluster or a GPU cluster. 'Value' is the price of our Asian option and ϵ measures the accuracy of the results using 95% CI. ϵ is related to the standard deviation 'std' of the simulation with the following relation: $\epsilon = 1.96 * \mathrm{std}/\sqrt{\mathrm{Numberoftrajectories}}$. We notice very slight differences between CPU and GPU simulations. The differences between GPU and CPU results are included in the 95% CI. Although we repeated the experiments with different parameters, we obtain the same similarity between GPU pricing and CPU pricing. This fact demonstrates that the single precision on GPUs does not affect the results of our simulations.

The parameters of the simulations are the following: Maturity $T = 1$, the time discretization $\delta t = 0.01$, $S_0^i = 100$, $r_i = r = 0.01$, $d_i = 0$, $\sigma_i = 0.2$ and the 40×40 correlating matrix $(\rho_{ik})_{1 \leqslant i,k \leqslant 40}$ is equal to the square root of a matrix (in the sense of Cholesky factorization) that is filled by 0.5 except on its diagonal which contains ones.

[‖]Next versions of nvcc are more compatible with OpenMPI.

DOI: 10.1002/cpe

Table III. Pricing results: central processing unit versus graphics processing unit.

Number of	CPU pricing		GPU pricing	
trajectories	Value	ϵ	Value	ϵ
2^{18}	5.8614	0.0258	5.8616	0.0257
2^{19}	5.8835	0.0183	5.8836	0.0183
2^{20}	5.8761	0.0129	5.8763	0.0129

CPU, central processing unit; GPU, graphics processing unit.

5.2. Computing efficiency

Effectiveness and speedup scaling:
Figure 3 shows that the execution times of the three benchmarks on each testbed decrease very regularly: using 10 times more nodes divides the execution time by ~ 10. This result is due to the *embarrassingly parallel* feature of our algorithm, (communications are limited to input data broadcast and result reduction). So, Figure 3 shows our parallelization *scales* and efficiently uses the CPUs and GPUs of the cluster *M2*.

We process our largest benchmark (one million MC trajectories) in 213.8 s on 16 multi-core CPUs, whereas it requires 61.3 s on one GPU and 4.9 s on 16 GPUs. Figure 3 shows N GPUs run about 45 times faster than N CPUs, so the speedup of our GPUs compared with our multi-core CPUs is close to 50. This speedup becomes close to 200 if we use only one CPU core, but using only one core of a CPU has no real sense. Also, according to Figure 3, when running the smallest benchmark of of 0.25 million trajectories on 16 GPUs, the computation time is 1.9 s, where it takes 53.0 s to run this simulation using 16 CPUs. The speedup on 16 GPUs of the 0.25 trajectories benchmark is small when compared with the one million trajectories benchmark, so the initialization time becomes significant and limits the global processing performance.

5.3. Energy efficiency

We only consider the energy consumption of the nodes that are actually used during the computation, as it is easy to remotely switch off unused nodes of our GPU cluster. However, it is not possible to switch off the GPU of one node when using only its CPU, and we have not yet tried to reduce the frequency and the energy consumption of the CPU when using mainly the GPU. Besides, we have not included the air conditioning energy consumption because the energy consumed depends on the type of air conditioning facility.

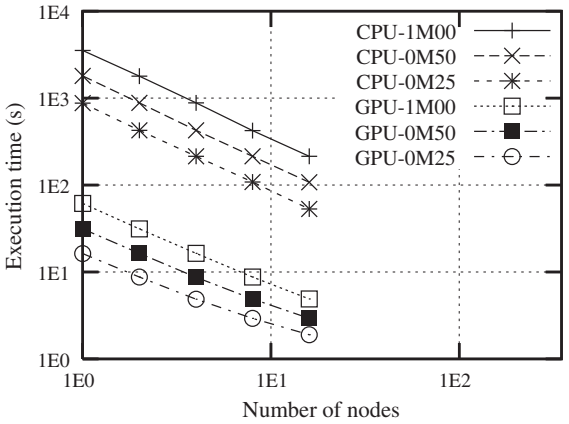

Figure 3. The execution time of pricing European options.

DOI: 10.1002/cpe

Effectiveness of computing energy:
The cluster switch consumption of *M2* remains constantly independent of the number of nodes used. Figure 4 shows that the GPU computations of *M2* consume on average 0.0046 kW.h to run our largest benchmark on 1 to 16 nodes, whereas the CPU computations consume on average 0.228 kW.h to run the same option pricing on 1 to 16 nodes.

Complete balance sheet:
Finally, using 16 GPU nodes, we run our largest benchmark in 4.9 s consuming 0.004 kW.h, in place of 213.8 s and 0.211 kW.h on 16 CPUs of the same *M2* cluster. It means we can perform our computation 43 times faster and consume 53 times less energy on our GPU cluster than on our CPU cluster. If we roughly consider the product of the speedup per the energy efficiency improvement, our GPU solution is globally $43 \times 53 \approx 2279$ times better than our CPU solution. As far as the smallest benchmark is concerned, we obtain a GPU solution, which computes 27 times faster and consumes 53 times less energy.

6. THE PARALLEL IMPLEMENTATION OF LONGSTAFF AND SCHWARTZ ON GRAPHICS PROCESSING UNIT

Although a lot of work has been done in variance reduction techniques, here, we prefer the implementation of a basic LSR, which will help a better understanding of the CPU/GPU comparison. In more standard applications, one can also implement the importance sampling method [23] or a European price as a control variable to accelerate the convergence. First of all, we detail the different steps of LSR in Algorithm 2. Afterwards, we are going to present the GPU version of Algorithm 2 in Algorithm 3. In Algorithms 2 and 3, we use the parameter l as a path index and i as a dimension index; we also denote n as the number of simulated paths, m as the total dimension and δt the time discretization. In addition, we use the set $\Gamma_l = \{\overline{C}(S_t^{(l)}) < \Phi(S_t^{(l)})\}$ that tests the continuation for each trajectory l using the indicator application 1.

6.1. Parallel path generation on graphics processing unit

This part of the algorithm depends on whether the random number generator can be parallelized or not. But, as presented in Section 3.1, we use the CMRG that is parallelized by period splitting. Consequently, the PG phase is an embarrassingly parallel part of the simulation, and we generate independently the random numbers for each path, then we use the Brownian bridge technique [1] to generate the Brownian motions and the asset prices at each time step according to (1).

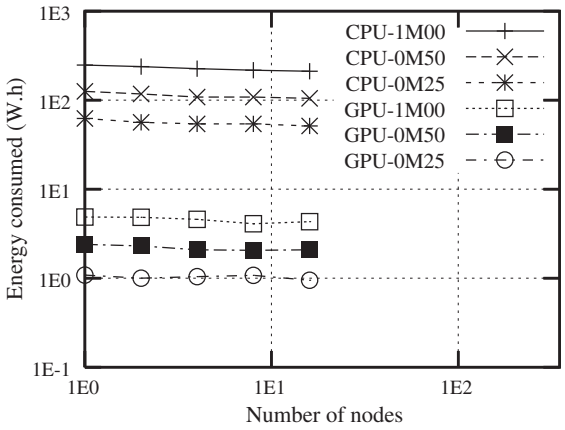

Figure 4. Energetic consumption.

DOI: 10.1002/cpe

Input: Model parameters and CMRG initialization.
Output: $P_0(S_0)$
for $t \in \{T, \ldots, 2\delta t, \delta t\}$ **do**
  `/* Computations performed during the PG phase */`
  for $i \in \{1, \ldots m\}$ **do**
    for $l \in \{1, \ldots n\}$ **do**
      - Draw $W_t^{i,(l)}$ using CMRG and the Brownian bridge induction
      - Use (1) to update the asset price $S_t^{i,(l)}$

    end
  end
  if $(t < T)$ *and* $l \in \{\Phi(S_t^{(l)}) > 0\}$ **then**
    `/* Computations performed during the REG phase */`
      - Approach the expectations: (12) and (13)
      - $A = \Psi^{-1}\mathbb{E}\left(P_{t+\delta t}(S_{t+\delta t})\psi(S_t)\right)$

    `/* Computations performed during the PRC phase */`
    for $l \in \{1, \ldots n\}$ **do**
      - $\overline{C}(S_t^{(l)}) = A^t\psi(S_t^{(l)})$
      - Compute the payoff $\Phi(S_t^{(l)})$
      - $P_t(S_t^{(l)}) = 1_{\Gamma_l}\Phi(S_t^{(l)}) + 1_{\Gamma_l^c}e^{-r\delta t}P_{t+\delta t}(S_{t+\delta t}^{(l)})$

    end
    if $(t = \delta t)$ **then**
      `/* `$P_0(S_0)$` is the price of the option */`
      $P_0(S_0) = \max\left(\Phi(S_0), \frac{e^{-r\delta t}}{n}\sum_{l=1}^{l=n} P_{\delta t}(S_{\delta t}^{(l)})\right)$
    end
  end
  else
    `/* Computations performed during the PRC phase */`
    for $l \in \{1, \ldots n\}$ **do**
      $P_t(S_t^{(l)}) = 1_{t=T}\Phi(S_T^{(l)}) + 1_{\Phi(S_t^{(l)})\leq 0}e^{-r\delta t}P_{t+\delta t}(S_{t+\delta t}^{(l)})$
    end
    `/* We have, of course, `$\forall l \in \{1, \ldots n\}\ P_{T+\delta t}(S_{T+\delta t}^{(l)}) = 0$` */`
  end
end

Algorithm 2: LSR algorithm for an American put option

6.2. The regression phase on graphics processing unit + central processing unit

As mentioned previously, the convergence study given in [8] shows that the number of trajectories needed to approximate the expectation (13) is more than exponentially proportional to the degree of regression. Thus, for the REG phase we use monomials of degrees less than or equal to two for one dimension, and we will use affine regression in the multidimensional simulation. We do not

DOI: 10.1002/cpe

use monomials of degrees up to 2 in the multidimensional simulation because it does not improve significantly the numerical results either.

Besides, the inversion of matrices cannot be done in parallel. Subsequently we need to transfer all the following values for each path l, from GPU to CPU:

$$P_{t_{k+1}}\left(S_{t_{k+1}}^{(l)}\right)\psi\left(S_{t_k}^{(l)}\right) \quad and \quad \psi\left(S_{t_k}^{(l)}\right)\psi^t\left(S_{t_k}^{(l)}\right) \tag{11}$$

where l is the path index.

When the values of (11) are in the CPU memory, we approach expectations (12) and (13) with an arithmetic average, then we perform the singular value decomposition) method according to [9].

$$\mathbb{E}\left(P_{t_{k+1}}(S_{t_{k+1}})\psi(S_{t_k})\right) \approx \frac{1}{n}\sum_{l=1}^{n} P_{t_{k+1}}\left(S_{t_{k+1}}^{(l)}\right)\psi\left(S_{t_k}^{(l)}\right) \tag{12}$$

$$\mathbb{E}\left(\psi(S_{t_k})\psi^t(S_{t_k})\right) \approx \frac{1}{n}\sum_{l=1}^{n} \psi\left(S_{t_k}^{(l)}\right)\psi^t\left(S_{t_k}^{(l)}\right) \tag{13}$$

with n representing the number of paths.

In this simulation phase, the GPU plays a role in the computation of the different products. For example in the case of three assets $\psi\left(S_{t_k}^{(l)}\right) = \left(1, S_{t_k}^{1,(l)}, S_{t_k}^{2,(l)}, S_{t_k}^{3,(l)}\right)$, one has to compute on GPU the following products associated to each path l:

$$\begin{array}{c}\left(\left(S_{t_k}^{1,(l)}\right)^2, \left(S_{t_k}^{2,(l)}\right)^2, \left(S_{t_k}^{3,(l)}\right)^2, S_{t_k}^{1,(l)}S_{t_k}^{2,(l)}, S_{t_k}^{1,(l)}S_{t_k}^{3,(l)}, S_{t_k}^{2,(l)}S_{t_k}^{3,(l)}\right) \\ \left(P_{t_{k+1}}\left(S_{t_{k+1}}^{(l)}\right)S_{t_k}^{1,(l)}, P_{t_{k+1}}\left(S_{t_{k+1}}^{(l)}\right)S_{t_k}^{2,(l)}, P_{t_{k+1}}\left(S_{t_{k+1}}^{(l)}\right)S_{t_k}^{3,(l)},\right)\end{array} \tag{14}$$

As will be demonstrated in Section 7.1, performing these computations on the GPU compensates for the loss caused by the data transfer between GPU and CPU.

6.3. The parallel pricing on graphics processing unit

Once we compute the regression vector A, the backward induction (4) can be done independently for each path of the simulation. At the final step time, we transfer the different price values from GPU to CPU and estimate the expectation using the arithmetic average of all prices.

7. PRICING AMERICAN CONTRACTS USING GRAPHICS PROCESSING UNITS

In this work, we were not able to directly compare our results with those presented in [10] as the authors do not give precise enough information on the digital procedure. Note that we also study here the running time of a multidimensional case, which is more representative of real American option challenges. The results presented in this section are evaluated by computing an average value of the different simulation times

We divide this section in three parts. The first part includes the comparison between our GPU implementation on *M1* using the NVIDIA Cg Toolkit and the QuantLib open-source library [24] implementation on the same machine as the Longstaff and Schwartz algorithm. The second part studies the dependance of the running time on *M1* of a multidimensional American option according to the number of paths simulated and the dimension of the contract. Using the results of the previous subsections, in the last subsection, we discuss a possible parallelization of pricing ACs on a cluster and which introduces one prospective work related to this article.

7.1. The running time comparison between graphics processing unit and central processing unit

The QuantLib open-source library is a highly object-oriented library. In order to make a fair comparison between the GPU and the CPU, we need to avoid overheads, which are unrelated to our algorithm. Thus, we only concentrate on the execution time of the main three phases of the

DOI: 10.1002/cpe

Table IV. Running time (s): central processing unit versus graphics processing unit.

Simulation phases	50 time steps		100 time steps		300 time steps	
	CPU	GPU	CPU	GPU	CPU	GPU
PG	0.671	0.047	1.278	0.079	4.386	0.162
PRC	0.484	0.064	1.315	0.116	8.864	0.359
REG	0.266	0.222	0.557	0.447	1.919	1.324

CPU, central processing unit; GPU, graphics processing unit; PG, paths generation phase; PRC, pricing phase ; REG, regression phase.

simulation. Moreover, we only consider the original one-core implementation of QuantLib implementation, and we do not parallelize the simulation on the two cores of *M1*. In Table IV, we compare the execution time between our GPU implementation and the QuantLib one-core CPU implementation of ACs for an increasing number of time steps. We perform the simulation of one-dimensional American put on $2^{14} = 16,384$ trajectories. According to Table IV, the REG phase is faster on the GPU than it is on the CPU. The two other phases are significantly improved when using the GPU, which reduces the total time of the simulation. It is also noticeable that when we increase the number of time steps, we make the simulation more complex, and this provides a higher speedup.

Input: The same as in Algorithm 2
Output: $P_0(S_0)$
GPU initialization.
for $t \in \{T, \ldots, 2\delta t, \delta t\}$ **do**
  `/* Computations performed during the PG phase */`
  Distribute the n trajectories on stream processors + Perform the same operations as Algorithm 2.
  if $(t < T)$ *and* $l \in \{\Phi(S_t^{(l)}) > 0\}$ **then**
    `/* Computations performed during the REG phase */`
    - Perform the products (14) on GPU.
    - Transfer (11) from GPU to CPU.
    - Same operations as Algorithm 2.

    `/* Computations performed during the PRC phase */`
    Distribute + Perform the same operations as Algorithm 2.
    if $(t = \delta t)$ **then**
      - Transfer from GPU to CPU: $(P_{\delta t}(S_{\delta t}^{(l)}))_l$
      - Compute the price of the option $P_0(S_0)$ as in Algorithm 2.

    end
  end
  else
    `/* Computations performed during the PRC phase */`
    Distribute + Perform the same operations as Algorithm 2.
  end
end

Algorithm 3: GPU version of LSR algorithm for an American put option

DOI: 10.1002/cpe

Table V. Increasing dimensions and trajectories on graphics processing unit (time in seconds)

Simulation	1 asset (2^{14})	4 assets (2^{14})	4 assets (2^{16})	4 assets (2^{18})
Total	1.092	1.591	2.605	7,360
PG	0.047	0.146	0.159	0.171
PRC	0.064	0.114	0.303	1.090
REG	0.222	0.588	1.400	5.387

PG, paths generation phase; PRC, pricing phase ; REG, regression phase.

7.2. Multidimensional American option

In this part, we compare the running times of one-dimensional American put ($2^{14} = 16,384$ trajectories and 50 time steps) with the running times of four assets American put (using 50 time steps) that has the following payoff $\Phi(S_T)$:

$$\Phi(S_T) = \left(K - \prod_{i=1}^{4} (S_T^i)^{1/4} \right)_+ \tag{15}$$

We study this multidimensional payoff because it is easier to check the prices coherence. Indeed, the American put on a geometric average of stocks can be approximated very well when using the one-dimensional equivalence and a tree method. Besides, unlike [25], to reduce the complexity of the REG phase, we restrict ourselves to the constant and linear monomials regression for the multidimensional benchmark.

In Table V, the first line provides the total running time that includes initialization and CPU/GPU data transfer[**]. The three columns on the right show the running times of the four assets American put associated to an increasing number of trajectories: 2^{14}, 2^{16}, 2^{18}.

According to Table V, the running time of the PG phase increases linearly with the number of assets and is slightly modified when we increase the number of trajectories. Conversely, the PRC phase is rather sensitive to the number of trajectories. Like the PRC phase, the running time of REG is approximately linear with the number of trajectories, and this is also the case when we increase the dimensionality of the problem[††]. Finally, even if pricing multidimensional ECs on GPUs allows better overall speedup, we obtain very short running times for a multi-asset American option pricing using a large number of trajectories.

When comparing the phases in Table V, we see that the total running time on GPU is $\sim$ 70% dominated by the running time of the REG phase. We will see, in the next subsection, a method that can reduce the execution time of the REG phase using a cluster of machines.

7.3. About pricing American contracts on graphics processing unit cluster

Pricing multidimensional ACs remains one of the most challenging problems in financial applications. The popularity of methods based on MC that use regression is due to the fact that they provide, in a sufficiently short time, relatively good solutions to dimensions included between one and three or one and five (It depends on the variance and the regression basis). Knowing the strengths and the weaknesses of these methods, we only tried to take advantage of the parallel architecture of the GPU to reduce the execution time of the Longstaff and Schwartz algorithm. As a result of the previous subsections, we show that we can efficiently execute on GPUs the phases PRC and PG. In this subsection we present how to reduce the execution time of the REG phase using a cluster of machines.

[**]The initialization and the data transfer takes at most 0.8 s.

[††]Because in the one-dimensional benchmark, we use $(1, S_1, S_1^2)$ as a regression basis, and we use $(1, S_1, S_2, S_3, S_4)$ for the four-dimensional benchmark.

DOI: 10.1002/cpe

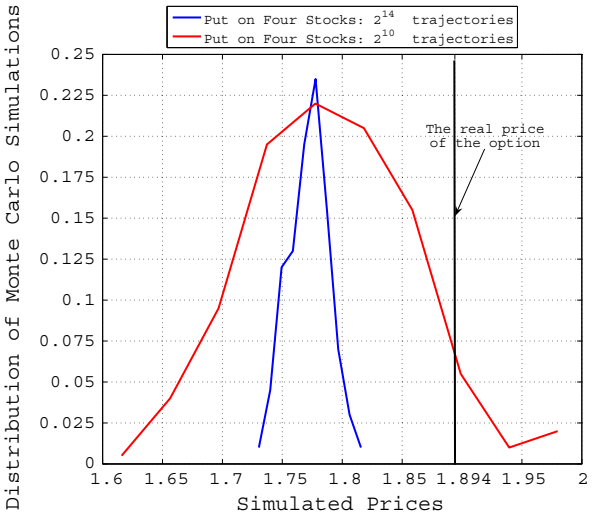

Figure 5. The histogram of simulated prices according to the number of trajectories.

In Figure 5, we sketch the histogram of 200 simulated prices of the four dimensional American put whose payoff is given by (15). The parameters of the simulations are the following: Maturity $T = 1$, time discretization $\delta t = 0.02$, $S_0^i = 100$, $r_i = r = 0.0953$, $d_i = 0$, $\sigma_i = 0.2$ and the 4×4 correlating matrix $(\rho_{ik})_{1 \leqslant i,k \leqslant 4}$ is equal to the identity matrix. As said above, we choose this multidimensional benchmark because we can have a good approximation of the price of the option using the one-dimensional equivalence and the tree method. In Figure 5, we give the real value[‡‡] of this option and the prices resulting from the MC simulation of 2^{10} and 2^{14} trajectories.

According to Figure 5, the two histograms are centered approximately around the same value, which is different from the real value of the option. This difference is due to regression errors, and even if we use more trajectories (2^{14} instead of 2^{10}), the average of simulated values remains relatively unchanged. Nevertheless, when we increase the number of the simulated trajectories, we shrink the distribution of simulated prices and thus we reduce the difference between the real value and the simulated value. We refer the reader to [3] for a CLT result of ACs.

Consequently, we can parallelize the AC pricing on a cluster of 16 machines using 2^{10} trajectories for each machine then averaging instead of simulating 2^{14} trajectories on only one machine. The former solution will improve the running time of the PG and the PRC phases. Also, according to the results of subsection 7.2, the decrease in the number of trajectories simulated per machine reduces almost linearly the execution time of the REG phase. The overall solution obtained would be more effective on a cluster of machines than it is on only one machine.

In the previous analysis, in order to parallelize our implementation on a cluster of machines, we use the fact that the reduction of the number of simulated trajectories does not affect the errors implied by the regression phase. However, there are limits to this result, indeed [8] recommends to have a number of polynomials $K = K_N \sim O(\sqrt{\log(N)})$ where N is the number of paths. Thus, for a fixed number of polynomials this determines approximately the minimum number of the simulated paths needed for a good regression.

8. CONCLUSION AND FUTURE WORK

The main results of this research work are the following:

- We have analyzed two different methods of parallelizing RNGs for parallel and distributed architectures. The results of this study are two examples of RNGs, which are most suited to the GPU architecture.

[‡‡]It is the value approximated using the tree method.

DOI: 10.1002/cpe

- When running MC simulations, the accuracy of the results obtained with a cluster of GPUs using single precision is similar to the one obtained with a cluster of CPUs using double precision.
- Mixed coarse and fine grain parallelization of MC simulations for pricing ECs, using MPI and OpenMP on multi-core CPU cluster, or MPI and CUDA on GPU cluster, is an efficient strategy and scales.
- Execution time and energy consumption of MC simulations can be both efficiently reduced when using GPU clusters in place of pure CPU clusters.
- In the case of American options that depend on one asset, we compare our GPU implementation with the one given in QuantLib library. Even if the speedup is small compared with pricing ECs, we observe a 2–10 improvement of the execution time and the speedup increases with the complexity of the problem.
- We look into the multi-asset American option and how the execution time can change with the dimension and the number of trajectories. As a result, when using GPUs, the execution time is almost only dominated by the REG phase because it is the only phase that cannot be parallelized on the GPU. Consequently, we give a method that aims at reducing the running time of the REG phase and which is based on a cluster implementation.

Algorithms introduced in this paper remain adapted to the new multi-core CPUs and the new generation of graphic cards, which computes in double precision.

Similar to the European contracts, we are going to extend the ACs pricing on a CPU/GPU cluster using the method presented in the Section 7.3. Subsequently, we will compare the speedup and the energy efficiency of the parallelization on GPUs and CPUs using the coarse-grained and fine-grained paradigms.

Besides, in order to improve the parallelization of the American options pricing, we are studying now the Malliavin Calculus-based algorithms [6] which completely avoid matrix regressions and allow an efficient computation of the hedge.

ACKNOWLEDGEMENTS

This research is part of the ANR-CIGC GCPMF project, and is supported both by ANR (French National Research Agency) and by Region Lorraine. The authors want to thank Professor Pierre L'Ecuyer for his valuable advices on the choice of random number generators.

REFERENCES

1. Glasserman P. *Monte Carlo Methods in Financial Engineering*, Applications of Mathematics. Springer: New York, 2003.
2. Williams D. *Probability with Martingales*. Cambridge University Press.: Cambridge, 1991.
3. Clément E, Lamberton D, Protter P. An analysis of a least squares regression algorithm for american option pricing. *Finance and Stochastics* 2002; **17**:448–471.
4. Tsitsiklis J, Van Roy B. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks* 2001; **12**(4):694–703.
5. Longstaff FA, Schwartz ES. Valuing American options by simulation: A simple least-squares approach. *The Review of Financial Studies* 2001; **14**(1):113–147.
6. Bally V, Caramellino L, Zanette A. Pricing American options by Monte Carlo methods using a Malliavin Calculus approach. *Monte Carlo Methods and Applications* 2005; **11**:97–133.
7. Bally V, Pagès G. A quantization algorithm for solving multidimensional discrete-time optimal stopping problems. *Bernoulli* 2003; **9**(6):1003–1049.
8. Glasserman V, Yu B. Number of paths versus number of basis functions in American option pricing. *The Annals of Applied Probability* 2004; **14**(4):2090–2119.
9. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. *Numerical Recipes in C++: the Art of Scientific Computing*, 2nd ed. Cambridge University Press: Cambridge, 2002.
10. Choudhury AR, King A, Kumar S, Sabharwal Y. Optimizations in financial engineering: the least-squares Monte Carlo method of Longstaff and Schwartz. *IEEE International Parallel & Distributed Processing Symposium*, April 2008; 1–11.
11. Podlozhnyuk V, Harris M. Monte Carlo option pricing. *Part of CUDA SDK documentation, nVIDIA* November 2007.
12. Matsumoto M, Nishimura T. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation* 1998; **8**(1):3–30.

DOI: 10.1002/cpe

13. Howes L, Thomas D. *GPU Gems 3*, Chapter 37. Efficient Random Number Generation and Application Using CUDA. Addison-Wesley: Boston, 2007.
14. Mascagni M, Srinivasan A. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software* 2001; **28**(3):436–461.
15. Matsumoto M, Saito M, Haramoto H, Nishimura T. Pseudorandom number generation: impossibility and compromise. *Journal of Universal Computer Science* 2006; **12**(6):672–690.
16. Niederreiter quasirandom sequence generator and mersenne twister http://www.nvidia.com/content/cudazone/cuda_sdk/computational_finance.html[13 december 2009].
17. L'Ecuyer P. Combined multiple recursive random number generators. *Operations Research* 1996; **44**(5):816–822.
18. L'Ecuyer P, Simard R, Chen EJ, Kelton WD. An object-oriented random-number package with many long streams and substreams. *Operations Research* 2002; **50**(6):1073–1075.
19. Knuth DE. *The Art of Computer Programming*. Addison-Wesley Publishing: Massachusetts, 1998.
20. Abbas-Turki LA, Vialle S, Lapeyre B, Mercier P. High dimensional pricing of exotic European contracts on a GPU cluster, and comparison to a CPU cluster. *Parallel and Distributed Computing in Finance, in IEEE International Parallel & Distributed Processing Symposium*, May 2009; 1–8.
21. Lapeyre B, Temam E. Competitive Monte Carlo methods for the pricing of Asian options. *Journal of Computational Finance* 2001; **5**(1):39–59.
22. Pacheco PS. *Parallel Programming with MPI*. Morgan Kaufmann: San Francisco, 1997.
23. Moreni N. A variance reduction technique for American option pricing. *Physica A: Statistical Mechanics and Its Applications* 2004; **338**:292–295.
24. The QuantLib Library http://quantlib.org/index.shtml, 2008.
25. Abbas-Turki LA, Lapeyre B. American options pricing on multicore graphic cards, July 2009; 307–311.

DOI: 10.1002/cpe