High Dimensional Pricing of Exotic European Contracts on a GPU Cluster, and Comparison to a CPU Cluster

Lokman A. Abbas-Turki*, Stephane Vialle†‡, Bernard Lapeyre*, Patrick Mercier†
*ENPC-CERMICS, Applied Probability Research Group, 77455 Champs-sur-Marne, France
†SUPELEC, IMS group, 2 rue Edouard Belin, 57070 Metz, France
‡AlGorille INRIA Project Team, 615, rue du Jardin Botanique 54600 Villers-les-Nancy France, France

Abstract

The aim of this paper is the efficient use of CPU and GPU clusters for a general path-dependent exotic European pricing, and their comparison in terms of speed and energy consumption. To reach our goal, we propose a parallel random number generator which is well suited to the parallelization paradigm, then, we implement a multidimensional Asian contract as a benchmark using g++/OpenMP/OpenMPI on CPUs and CUDA-nvcc/OpenMPI on GPUs. Finally, we give the detailed results of the two architectures for different size problems using 1-16 GPUs and 1-256 dual-core CPUs.

1. Introduction and Objectives

Looking for the highest performant architecture, financial institutions spend considerable amounts of money in their Information Systems. The performance involved can be evaluated using various criteria such as the precision of the results obtained, the speed of simulation and the energy consumed by the machines during the simulation. The goal of this work is, on the one hand, to design and test a pricer for European Exotic contracts on GPU cluster which is effective even in very high dimension. On the other hand, it is to compare the performance of a cluster of 16 GPUs with a cluster of 256 dual-core CPUs.

After familiarizing the reader with Monte Carlo (MC) applied to contract pricing in the second section, we will give, in the third section, details on the way of building a Parallel Random Number Generator (PRNG) which satisfies some quality criterions and which achieves high performance on parallel architectures. In the fourth section, we will show the algorithms and software architectures designed to exploit CPU and GPU clusters. In order to compare the speedup and energy consumption between GPU cluster and CPU cluster, we implement in the fifth section a benchmark application on both clusters and we analyse numerous experiments. Finally, section six summarizes our results and introduces some future works.

2. Monte Carlo in Financial Applications

The most widely used method in financial field, MC simulation gets its popularity in financial community for three reasons: (1) the possibility to use MC for the majority of problems encountered in finance, (2) the easiness of implementation and parallelization, and (3) contrary to finite element methods, MC still efficient in dimension greater than 4 which is appropriate in portfolio management.

In some low dimension cases (see contribution [1]) other method can be used, but we can expect a much bigger speedup with MC on GPU and this whatever the dimension of the problem. Note also that the binomial lattice that is presented in [2] can be adapted to GPU. However, the problem of this last method is its limitations with respect to dimension increase. This makes them less and less used by financial institutions. Consequently, because MC could give a very high speedup when an appropriate PRNG is used and because MC is the only method which is stable in very large dimension, in addition to the fact that MC is the method which is widely used in financial institutions, leads us to explore its effectiveness on a cluster of GPUs.

Because of the numerical similarity between the problem of pricing and hedging contracts when using MC, we present here only the general aspect of pricing contracts. For a detailed presentation on MC in financial applications, we send the reader to [3].

The general MC method is articulated on two theorems that constitute the two pillars of probability theory. The first one is the Strong Law of Large Numbers (SLLN) that announces the convergence of a certain series 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.

Pricing European contracts with MC is no more than using the result of SLLN and CLT on a certain kind of random functions like those presented in table 1, where $(x)_+ = max(x, 0)$. The variable ε is a normal random variable and we suppose generally that the price of the stock S_t is log-normally distributed. Thus, the first stage of using MC is to simulate the Gaussian distribution of ε through a set of samples ε_i. The assumption in SLLN and CLT

Table 1. 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	$(f(S_T(\varepsilon))1_{max_T<L})$
Asian Floating Strike	$(mean_T S_t(\varepsilon) - S_T(\varepsilon))_+$

theorems that makes MC more attractive than other scientific computation tools for GPGPU is the **'independence'** of ε_i. The main concern of using MC on GPUs is how to spread the independence of the random variables on the stream processor units. Fortunately, the independence involved takes part only in the choice of the PRNG.

3. Parallel RNG for SIMD Architecture

The article [4] proposes on the one hand, two efficient PRNGs for GPU: one based on the Combined Tausworthe Generator which generate uniform distribution and the other one being the Wallace Gaussian Generator. On the other hand, we find in [4] algorithms for Asian and Lookback pricing. The authors choose PRNGs which are well suited to the GPU architecture and which have good sequential property, but they do not prove the parallel independence of the different streams of the PRNG.

The simplest theoretical solution in parallelizing random number generator (RNG) is to split the period of a good sequential one into different sub-streams. Because, we are going to split the whole period we need to have a long period to split. For exemple, we cannot split a period of a standard LCG on a high number of sub-streams[1] because it reduces considerably the period of each sub-stream. One of the most suited RNG to GPUs architecture and which has a very good random behavior is the CMRG given in the exemple 4 of [5]. Indeed, once we determine the number of sub-streams[2], we compute the power of the companion matrix associated to the recurrence of CMRG and which allows us to initialize the different sub-streams at the different points of the period. For more precision, we send the reader to [6].

The adaptation of the CMRG on GPUs will be done in further works. In this work we present the adaptation on GPUs of another RNG which is more efficient than CMRG but less strong. Indeed the goal of this work is to exhibit a comparison of speedup and energy consumption on two parallel architectures using the most adapted method to this kind of architecture. However, we prove in 3.2 that the quality of our PRNG is sufficient for our application and for the majority of financial applications. Also the European path-dependent contract considered here is significantly representative of all path-dependent European contracts used in the market. The authors of [7] and [8] parallelize very strong RNGs[3] on GPUs and Cell processor but the the speedup remains small when compared to the number of stream processor units. The PRNG that we are going to present is suited to the GPU architecture and generally to the parallelization paradigm which will lead us to a very high speedup.

3.1. Parallel-RNG from Parameterization of RNGs

As it is mentioned by the authors of the Mersenne Twister RNG in [9], an acceptable way to parallelize RNGs is to parameterize them. Thus, we will apply here this method to a cluster of GPUs/CPUs. Our choice of PRNG is based on two articles: the first one [10] deals with sequential random behavior of an LCG and the second one [11] deals with the parallel random behavior of LCGs that is used in the library SPRNG (see [12] for more details). These two articles allow us to propose a PRNG made of 1024 RNGs which have, two by two, good parallel independence property and each of these RNGs has a good sequential independence property. The total period of our PRNG is $2^{41} = 2^{31} \times 1024$, which is sufficient in our tests and generally for the most MC simulations in the financial field. Indeed, because in real financial applications we use less than 2^{30} random numbers.

An LCG is based on the relation:

$$x_n = ax_{n-1} + c \; mod(m) \tag{1}$$

This recurrence mimics a uniform distribution on the interval $(0, m)$. With special criterions we can reach the maximum period of (1) (m if $c \neq 0$ and $m - 1$ if $c = 0$). Also according to m we can consider two cases: $m = 2^{32 \; or \; 64}$ or $m = 2^{31 \; or \; 61} - 1$. An $m = 2^{32 \; or \; 64}$ shows a deficiency in the random behavior of the bits. Also because of the single precision of the GPUs used in our application, we take $m = 2^{31} - 1$ and $c = 0$. This choice is good enough to obtain convincing results in term of precision (see section 3.2). Besides, this generator possesses a very simple method of parallelization based on the parameterization of multipliers [11]. We have to choose appropriate multipliers a in (1) and launch different generators at the same time parameterized with these multipliers.

This idea of parallelization using parameterization is stated in [11]. We summarize this idea with algorithm 1:

1. for exemple 1024 sub-streams
2. for instance equals to the number of trajectories simulated or equals to the number of processors involved in GPUs

3. In the sense that verified various statistical tests

```
Initialization: $A[0] = a_b$; $r = 1$; $i = 1$;
while $r < r_0$ do
    if $r$ is a prime number with $m - 1$ then
        $a = a_b^r \ mod \ m$;
        $A[i] = a$;
        $i$++;
    end
    $r$++;
end
```

Algorithm 1: RNG parallelization using parameterization

We use in this algorithm the multiplier $a_b = 62089911$ which is the best one according to [13]. The constant r_0 expresses how long is the distance that separates a_b and the most distant multiplier $a_b^{r_0} \ mod(m)$ in the multiplicative group LCG($2^{31} - 1$). Mascagni M. [11] established that the biggest r_0 is the worst are the multipliers of the vector A in the inter-correlations of the random numbers generated by them. According to [11], the criterion for parallel independence is:

$$r_0/\sqrt{m} << 1 \qquad (2)$$

In our application $\sqrt{m} \sim 46341$, so we will choose $r_0 \sim 2000$, which leads to the maximal number of RNGs that have a good parallel independence property.

Once we have the vector of multipliers A that satisfy a good parallel independence property, we must choose among these multipliers those that satisfies a good sequential independence property. This last point was not treated by [11]. However, the work of Knuth [10] brings us a detailed arithmetic study of good multipliers. He particularly mentioned in [14] that the best multipliers are those that minimize the expressions bellow:

$$\left\{ \begin{array}{c} \left| -\frac{1}{2} \sum_{1 \le j \le t}^{j \ odd} a_j - \sum_{1 \le j \le t}^{j \ even} a_j + \frac{1}{2} \right| , \\ \left| \sum_{1 \le j \le t}^{j \ odd} a_j + \frac{1}{2} \sum_{1 \le j \le t}^{j \ even} a_j - \frac{1}{2} \right| \end{array} \right\} \qquad (3)$$

Where $a_j =$ are partial quotients obtained by successive divisions using Euclid's algorithm between $a \in A$ and m. This criteria ensures a low discrepancy, at least, in one and two dimensions and a good sequential independence property.

Finally we summarize the algorithm 1, the criteria of parallel independence (2) and the criteria of sequential independence (3) in the algorithm 2:

```
Perform Algorithm 1 with $r_0 = 2000$;
Keep only the 1024 best multipliers that minimizes the
    expressions in (3);
```

Algorithm 2: High quality parameterization

After choosing the 1024 best multipliers $A_{1 \le i \le 1024}$ and thus 1024 random generators according to the method exposed above, we can associate each generator with each stream processor used in the cluster of GPUs. If the number of stream processors is not much bigger than 1024, for example 2048, we can split the period of each LCG between two stream processor units which does not reduce considerably the orignal period.

Table 2. Simulation vs Explicit for: $\sigma_i = 0.2$.

K	Put_{MC}	ϵ_{MC}	$Put_{explicit}$
80	0.0621	0.0023	0.0610
90	0.5115	0.0078	0.5068
100	2.1862	0.0178	2.1723
110	5.9433	0.0299	5.9208
120	11.9023	0.0407	11.8810

Table 3. Simulation vs Explicit for: $K_i = 100$.

σ	Put_{MC}	ϵ_{MC}	$Put_{explicit}$
0.1	0.2729	0.0027	0.2704
0.2	2.1862	0.0178	2.1723
0.3	4.9271	0.0315	4.9024
0.5	8.0521	0.0445	8.0184
0.6	11.4249	0.0565	11.3833

3.2. Testing our Parallel RNG on a high dimensional problem

Once we built our PRNG, we will test it on a high dimensional[4] European problem having an explicit solution. For this we simulate a Put contract on a geometric average of 40 stocks that have the following expressions:

$$S_t^i = S_0^i \exp \left[\left(r - d_i - \frac{\sigma_i^2}{2} \sum_{k=1}^{i} \rho_{ik}^2 \right) t + \sigma_i \sum_{k=1}^{i} \rho_{ik} W_t^k \right] \qquad (4)$$

where:

S_0^i	is the initial price of the sotck i,
r	is the rate,
d_i	is the dividend of the stock i,
σ_i	is the volatility of the stock i,
W_t^k	is the k^{th} Brownian motion,
ρ_{ik}	is the coefficient of the stock i in the correlating matrix that weights the Brownian W_t^k.

Let us associate a strike K_i with each stock, then the price of the Put on geometric average of 40 stocks is given by:

$$Put_{MC} = E \left[e^{-rT} \left(\left(\prod_{i=1}^{40} K_i \right)^{\frac{1}{40}} - \left(\prod_{i=1}^{40} S_T^i(\varepsilon) \right)^{\frac{1}{40}} \right)_+ \right] \qquad (5)$$

This particular option has been chosen because its price is given by an explicit formula, namely:

$$Put_{explicit} = e^{-rT} \left[\left(\prod_{i=1}^{40} K_i^{\frac{1}{40}} \right) N_1 - \left(\prod_{i=1}^{40} (S_0^i)^{\frac{1}{40}} \right) N_2 \right] \qquad (6)$$

4. The dimensionality of the problem is equal to the number of stocks simulated

where:

$$N_1 = N(\alpha), \quad N_2 = N(\alpha - \sqrt{TV})e^{T\beta}$$

$$V = \sum_{i=1}^{40}\left(\frac{1}{40}\sum_{k=i}^{40}\sigma_k\rho_{ik}\right)^2, \quad \alpha = \frac{1}{\sqrt{V}}\left(\sum_{i=1}^{40}\gamma_i\right)$$

$$\gamma_i = \frac{1}{40\sqrt{T}}\left[\ln\left(\frac{K_i}{S_0^i}\right) - \left(r - d_i - \frac{\sigma_i^2}{2}\sum_{k=1}^{i}\rho_{ik}^2\right)T\right]$$

$$\beta = \frac{V}{2} + \frac{1}{40}\sum_{i=1}^{40}\left(r - d_i - \frac{\sigma_i^2}{2}\sum_{k=1}^{i}\rho_{ik}^2\right)$$

$$N(x) = \int_{-\infty}^{x}\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{t^2}{2}\right)dt$$

This fact enables to check the random equidistribution of the sequence generated by the PRNG given previously.

In tables 2 and 3, we give the comparison between the simulation according to (5) and the real value according to (6) for different values of K_i and σ_i. In those tables we use the parameters: $r = 0.1$, $T = 1$, $d_i = 0$, $S_0^i = 100$ and we choose the 95% confidence interval. Besides, the matrix of correlation that we choose is the square root of a matrix (in the sense of Cholesky factorization) that is filled by 0.5 except its diagonale which contains ones. In order to have a Gaussian distribution $N(0,1)$, we normalize by m the recurrence (1) to have uniform distributed variables on $(0,1)$, and we finally apply the well-known Box-Muller transformation [3]. Although we are integrating on a very high dimension we notice that table 2 and 3 show that the simulations are always included in the confidence interval, in other words: $|Put_{explicit} - Put_{MC}| < \epsilon_{MC}$.

4. Multi-Paradigm Parallel Algorithm

4.1. Parallelization strategy

In order to take advantage of various and heterogeneous architectures like multi-core CPUs, GPUs, cluster of multi-core CPUs and cluster of GPUs, 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 in some threads on a multi-core CPU, or in many light-threads on a GPU, communicating through a shared memory. Figure 1 illustrates this algorithm.

Input data files are read in step 1 on processing node 0, and input data are *broadcasted* to all other nodes in step 2. Then each node locally achieves its initializations in step 3, function of the common input data and its node number. Some of these initialization have been parallelized at fine grained level, and our parallel RNG is initialized on each node according to specifications of section 3.1.

In step 4 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.

In step 5 each node computes the sum of its computed prices and the sum of its square prices. Then all nodes participate to a global *reduction* of these P_N couples of results: at the end of step 6 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 (step 7).

Broadcast and *reduction* are classic communication routines, efficiently implemented in the standard MPI communication library [15] that we used. At the opposite, 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.

4.2. Fine grained parallelization

On a multi-core CPU node we have used standard OpenMP directives [16] to split computing loops and to create threads on the different cores. An OpenMP parallelization based on *orphan directives* has been easy to develop, and has achieved a speedup close to 2 on dual-core processors. An orphan directive adds just one line in the source code and can create threads, spread computations, and synchronize threads at the end of their execution. We tried to improve performances creating a large *parallel region*, including all split loops and avoiding to create and kill threads too frequently. But performances remained unchanged and the source code had become harder to understand. So, we have kept the parallelization based on OpenMP orphan directives.

On many-core GPU nodes we used the CUDA nVIDIA environment [17] to implement and run *kernels*: light threads running on GPU processors. Each computing C-routine of the CPU multi-core version still exists in the GPU version, but just defines and runs a *grid of blocks of CUDA threads* (a structured set of light threads). The computation codes of the previous C-routines are implemented in CUDA code in associated *kernel routines*, and main computing loops are replaced by the run of grids of light-threads. We have experimented different implementations of our CUDA kernels. A solution leading both to a clear source code and to fast execution has consisted in defining light-threads achieving just one iteration (i.e. processing just one MC trajectory). This strategy has led to quick development of the GPU version, and to the design of identical data-structures adapted to both multi-core CPU and many-core GPU architectures.

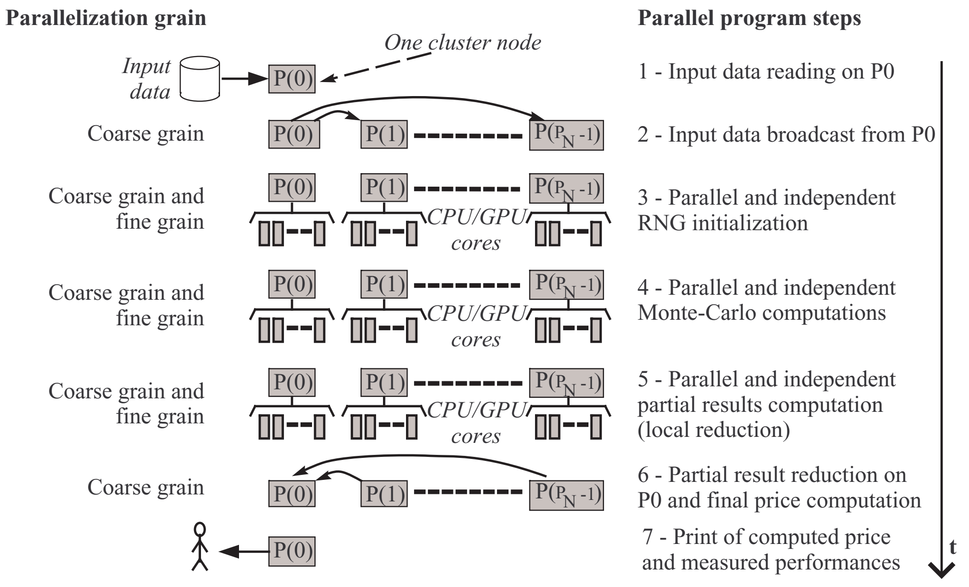
Parallelization grain
Parallel program steps
One cluster node
Input data
P(0)
Coarse grain
P(1)
P(P_N -1)
Coarse grain and fine grain
CPU/GPU cores
Coarse grain and fine grain
CPU/GPU cores
Coarse grain and fine grain
CPU/GPU cores
Coarse grain
1 - Input data reading on P0
2 - Input data broadcast from P0
3 - Parallel and independent RNG initialization
4 - Parallel and independent Monte-Carlo computations
5 - Parallel and independent partial results computation (local reduction)
6 - Partial result reduction on P0 and final price computation
7 - Print of computed price and measured performances
t

Figure 1. Global parallelization strategy, including coarse and fine grained parallelism

4.3. Source files and compilation

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 still the `OpenMPI 1.2.4` library. All these development environments appeared compatible.

Our CPU version is composed of `.h` and `.cc` source files, compiled with the classic following commands:

```
g++ -O3 -fopenmp -I/opt/openmpi/include
    -c X.cc
g++ -O3 -fopenmp -L/opt/openmpi/lib
    -o pricer X.o Y.o .... -lmpi -lm
```

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 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. Experiments and Validation

5.1. Benchmark introduction

In order to explore the performance of MC on a cluster GPUs, we are going to deal with European high dimensional contracts that depend on the trajectory of the stocks' prices (*path-dependant contracts*). We take here the example of an homogenous Asian option in 40 dimensions[5]. However, the procedure that we are going to illustrate can be easily generalized for all European path-dependant contracts.

Asian option is a contract whose price depends on the trajectory average. For example, we estimate a floating strike Asian option using:

$$E\left[e^{-rT}(S_T(\varepsilon) - \overline{S}_T)_+\right],\ with\ \overline{S}_T = mean_{0\leq t\leq T} S_t(\varepsilon)$$

Algorithm 3 introduces a recursive algorithm for computing the mean. The `While` loop is used for time discretization, in our application we take $\delta t = T/100$. This algorithm uses the well known rectangle approximation of an integral. However, in order to have a faster convergence, in our work we use the trapezium approximation which is presented in [18] and characterized by the same easiness as the rectangle one.

Next sections introduce results of three benchmark programs, implementing the algorithm 3 and computing 0.25, 0.5 and 1 million of MC trajectories (corresponding to different pricing accuracies).

5. It 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 CAC 40 index.

```
Initialization: $\overline{S}_0^i = S_0^i$; time step $t = 1$;
while $t \in \{0, \delta t, 2\delta t, \ldots T\}$ do
    for $i = 1 : 40$ do
        Actualization of $S_t^i$ using (4);
        $\overline{S}_t^i = ((t-1)/t).\overline{S}_{t-1}^i + (1/t).S_t^i$;
    end
end
$\overline{S_T} = \frac{1}{40} \cdot \sum_{i=1}^{40} \overline{S}_T^i$;
```

Algorithm 3: Computes $mean_{0 \leq t \leq T}$ on each trajectory

5.2. Testbed introduction

We have experimented and evaluated our parallel and distributed application on a CPU and a GPU clusters of SUPELEC (from CARRI Systems company). The first was a 256 CPU-node PC cluster with a total of 512 cores. Each node hosts one bi-core processor: INTEL Xeon-3075 at 2.66 GHz, with a front side bus at 1333MHz, 4 GB of RAM and 4MB of cache, and the interconnection network is a Gigabit Ethernet network built around a large and fast CISCO 6509 switch. The second was a 16 GPU-node PC cluster. Each node hosts one bi-core processor: Intel E8200, with a front side bus at 1333MHz, 4GB of RAM and 6MB of cache, and one GPU card: nVIDIA GeForce8800 GT with 512MB of RAM. On each node the CPU and the GPU are connected by a classical PCI Express bus, and the 16 nodes are interconnected across a Gigabit Ethernet network built around a small DELL Power Object 5324 switch (24 ports). The 256-CPU cluster has been built in December 2007 with up to date processors, and the 16-GPU cluster has been built in March 2008 with GPU cards appeared a little bit earlier. So, we can consider these two clusters have different technologies appeared at close dates.

5.3. Computing performances

Size up and speedup:. Each CPU node of our cluster can process any of our benchmarks, but our GPU nodes have only 512MB of memory and can process only 0.25 million of MC trajectories. To process more we have to adapt our program to chain several computations of 0.25 millions of trajectories, or to use several GPU nodes to achieve *size up*. We have adopted this last strategy, representative of a realistic usage of a GPU cluster. First we use enough GPU nodes to achieve size up and process the required problem, second we use more nodes to achieve speedup.

So, the performance curves relative to our CPU cluster (see figure 2) spread over a range from 1 to 256 nodes. But the performance curves on our GPU cluster start on 1, or 2 or 4 nodes and finish on 16 nodes, depending on the problem size (the number of Monte-Carlo trajectories depends on the required accuracy).

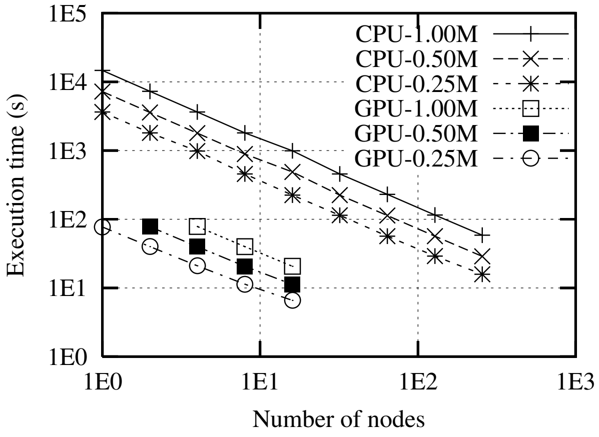

Figure 2. Pricing execution time

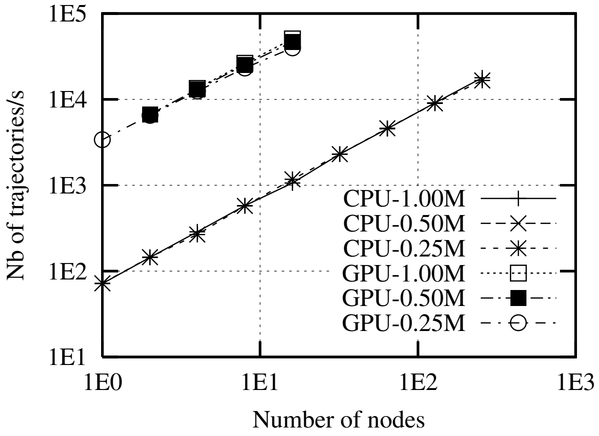

Figure 3. Computed trajectories per second

Performance scaling:. Figure 2 shows the execution times of the three benchmarks on each testbed decrease very regularly: using 10 times more nodes divide 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 2 shows our parallelization *scales* and efficiently uses a cluster of 256 bi-core CPUs and a cluster of 16 GPUs.

We process our largest benchmark (1 million of MC trajectories) in $58.7s$ on 256 bi-core CPUs, while it requires $78s$ on 4 GPUs (minimal number of GPUs to process this problem size) and $20.7s$ on 16 GPUs. Figure 2 shows N GPUs run 46 times faster than N CPUs, so the speedup of our GPUs compared to our bi-core CPUs is close to 46. This speedup becomes close to 100 if we use only 1 CPU core, but using only 1 core of a CPU has no real sense.

Processing speed:. The figure 3 shows the number of trajectories computed per second on the different clusters. This figure confirms the speedup of N GPUs compared to N bi-core CPUs is close to 46, and shows the processing speed of both clusters increases regularly with the number of used

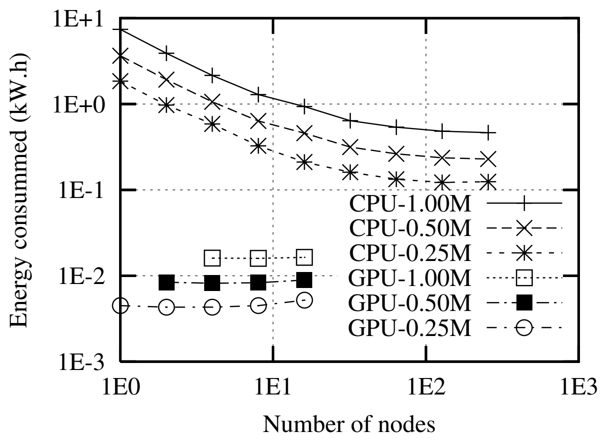

Figure 4. Global energetic consumption

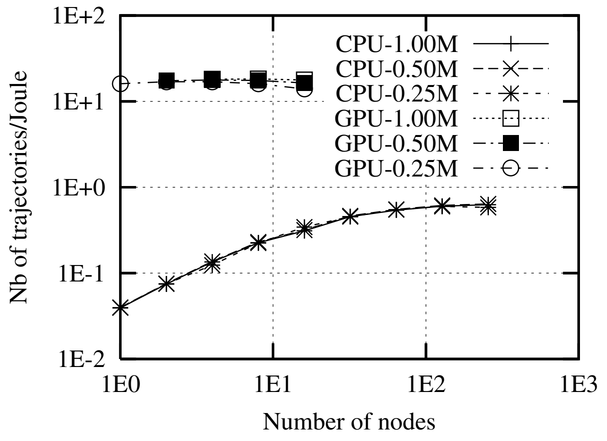

Figure 5. Computing energetic efficiency

nodes, but is not sensitive to the problem size, excepted when running the smallest benchmark on the full GPU cluster. Computation time of 0.25 million trajectories is very small on 16 GPUs, so the initialization time becomes significant and limits the global processing speed.

5.4. Energetic performances

Energetic consumption:. We have measured the energetic power required by the different computing nodes and cluster switches when running our application. To achieve these measures we started to install some *wattmeters* on these switches and cluster nodes, to measure their entire power consumption. However we did not measure the power required by the air conditioned system, that is used to refresh different technical rooms and a lot of machines. One node of our CPU cluster requires 104.5 Watts and the large switch of this cluster requires 1720.0 Watts, while one node of our GPU cluster requires 176.0 Watts and the small switch of the this cluster requires only 34.4 Watts. Considering these energetic power requirements, the

Table 4. MC Precision: CPUs vs GPUs

Number of trajectories	GPU pricing values	GPU pricing errors	CPU pricing values	CPU pricing errors
1024^2	5.87082	0.01288	5.87586	0.01290
512^2	5.86341	0.02572	5.85949	0.02572

application execution times and the number of used nodes, we can compute the energy consumed by each run. Figure 4 summarizes these energetic consumptions.

The cluster switch consumptions remain constant, independently of the number of used nodes. So, a big switch increases drastically the cluster consumption when using only few nodes. On a production cluster other applications would run on the other nodes, and the energetic switch consumption should be distributed among all these runs. So, we consider only the right part of the CPU cluster curves of figure 4, when using at less half of the nodes.

Figure 4 shows our GPU cluster consumes approximately $0.016\ kW.h$ to run our larger benchmark on 4 to 16 nodes, while our CPU cluster consumes approximately $0.464\ kW.h$ to compute the same option pricing on 128 to 256 nodes. It means we achieve the same computation consuming 28.3 times less energy on our GPU cluster than on our CPU cluster.

Computing energetic efficiency:. To make easier the comparison of the energetic efficiency of our implementations and clusters, we have computed the respective *computing energetic efficiency*: the number of MC trajectories computed per Joule (*i.e.* the number of trajectories computed per second and per Watt). Figure 5 shows the obtained computing energetic efficiencies. Again, only the right part of the CPU cluster curves have to be considered. Our MPI+CUDA implementation running on our GPU cluster computes up to 17.8 trajectories per Joule, while our MPI+OpenMP implementation running on our CPU cluster computes only up to 0.64 trajectories per Joule. Our GPU cluster solution appears again 28.3 times more efficient than our CPU cluster solution from an energetic point of view.

Complete balance sheet:. Finally, using 16 GPU nodes we run our larger benchmark in $20.7s$ consuming $0.016\ kW.h$, in place of $58.7s$ and $0.464\ kW.h$ on 256 nodes of our CPU cluster. It means we can achieve our computation 2.8 times faster and consume 28.3 times less energy on our GPU cluster than on our CPU cluster. If we consider the product of the speedup per the energetic efficiency improvement, our GPU solution is globally $2.8 \times 28.3 \approx 80$ times more interesting than our CPU solution.

6. Conclusion and Future Works

The main results of this research work are the followings:

- We have designed a highly parallel RNG, adapted to large parallel and distributed architectures and thus adapted for a comparison between the two parallel architectures considered in this work.
- When running MC simulations, the precision obtained with a cluster of GPUs using single precision is similar to the one obtained with a cluster of CPUs using double precision (see table 4).
- Mixed coarse and fine grain parallelization of MC simulations, using MPI and OpenMP on 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.

Algorithms introduced in this paper remain adapted to the new multi-core CPUs and the new generation of graphic cards launched by nVIDIA (including the GeForce GTX 295, significantly more performant than our GeForce 8800 GT).

Even if the parametrized LCG used is generally sufficient for financial applications, we will present soon the results obtained when we adapt the CMRG for GPU simulations.

Previously, we have implemented on one GPU using Cg language the multidimensional *lookback* and *barrier* contracts, and the multidimensional *target* and *ratchet* contracts in collaboration with CALYON Bank. We achieved important speedup (like for Asian contracts), and as a continuation of the work presented in this article, we aim to port these implementations in MPI+CUDA environment to run again on clusters of GPUs. Besides, two other problems are now studied on GPU clusters which are: American contracts and calibration problems. The goal of this study is to get as efficient implementations as European pricing.

Acknowledgment

This research is part of the ANR-CIGC GCPMF project, and is supported both by ANR (French National Research Agency) and by Region Lorraine.

References

[1] V. Surkov, "Parallel option pricing with fourier space time-stepping method on graphics processing units," *First Workshop on Parallel and Distributed Computing in Finance, in IEEE International Parallel & Distributed Processing Symposium*, April 2008.

[2] C. Kolb and M. Pharr, "Option pricing on the GPU," *GPU Gems2, Addison-Wesley*, 2005.

[3] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. 2003, Springer.

[4] L. Howes and D. Thomas, "Efficient random number generation and application using CUDA," *GPU Gems3, Addison-Wesley*, 2007.

[5] P. L'Ecuyer, "Combined multiple recursive random number generators," *Operations Research*, vol. 44, no. 5, 1996.

[6] P. L'Ecuyer, R. Simard, E. J. Chen, and K. W. D., "An object-oriented random-number package with many long streams and substreams," *Operations Research*, vol. 50, no. 6, 2002.

[7] V. Podlozhnyuk, "Parallel Mersenne Twister," *Part of CUDA SDK documentation, nVIDIA*, june 2007.

[8] V. Agarwal, L. Liu, and D. Bader, "Financial modeling on the Cell Broadband Engine," *IEEE International Parallel & Distributed Processing Symposium*, April 2008.

[9] M. Matsumoto, M. Saito, H. Haramoto, and T. Nishimura, "Pseudorandom number generation: Impossibility and compromise," *Journal of Universal Computer Science*, vol. 12, no. 6, 2006.

[10] D. E. Knuth, "Notes on generalized dedekind sums," *ICM, Acta Arithmetica*, vol. XXXIII, 1977.

[11] M. Mascagni, "Parallel linear congruential generators with prime moduli," *ACM Transactions on Mathematical Software*, vol. 24, no. 5-6, 1997.

[12] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A scalable library for pseudorandom number generation," *ACM Transactions on Mathematical Software*, vol. 28, no. 3, 2001.

[13] B. F. Fishman and L. R. Moore III, "An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$," *SIAM Journal on Scientific Computing*, vol. 7, 1986.

[14] D. E. Knuth, *The Art Of Computer Programming*. Addison-Wesley Publishing, 1981.

[15] P. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[16] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[17] *NVIDIA CUDA Compute Unified Device Architecture. Programming Guide. version 1.1*, nVIDIA, november 11 2007.

[18] B. Lapeyre and E. Temam, "Competitive monte carlo methods for the pricing of asian options," *Journal of Computational Finance*, vol. 5, no. 1, 2001.