Noname manuscript No.
(will be inserted by the editor)

Financial Applications on Multi-CPU and Multi-GPU Architectures

Emilio Castillo · Cristóbal Camarero · Ana Borrego · Jose Luis Bosque

Received: date / Accepted: date

Abstract The use of high performance computing systems in order to help to make the right investment decisions in financial markets is an open research field where multiple efforts have being carried out during the last few years. Specifically, the Heath-Jarrow-Morton (HJM) model has a number of features that make it well suited for implementation on massively parallel architectures. This paper presents a Multi-CPU and Multi-GPU implementation of the HJM model that improves both the performance and energy efficiency. The experimental results reveal that the proposed architectures achieve excellent performance improvements, as well as optimize the energy efficiency and the cost/performance ratio.

Keywords Heterogeneous Computing · Multi-GPU · Financial Applications

1 Introduction

Simulation is getting increasingly important in financial markets as one of the best techniques to improve the accuracy of investments. The duration of these simulations is critical as a microsecond or nanosecond faster than the rest of the players can lead to create market instead of just being on the market. That is why most of the key players are as close as possible to the stock exchange markets, thus reducing the communication time between orders and deals.

Financial simulations based on Monte Carlo methods have been used for many years thanks to their intrinsic parallelism. A Monte Carlo method is

Emilio Castillo · Cristóbal Camarero · Jose Luis Bosque
Department of Ingeniería Informátimca y Electrónica
Universidad de Cantabria
E-mail: emilio.castillo,cristobal.camarero,joseluis.bosque@unican.es

Ana Borrego
Technology Division
Grupo Santander - Produban
E-mail: aiborrego@produban.com

an algorithm that solves a problem through the use of statistical sampling to obtain numerical results [1]; typically it is necessary to run many simulations in order to obtain the distribution of an unknown probabilistic entity.

Monte Carlo methods have a number of properties that make them especially suitable for implementation on massively parallel architectures [2,3]. These include the data independence, that enable domain-based parallelization, with a high degree of parallelism. Hence these methods can generate a large number of fine-grained tasks or a few coarse-grain tasks. This property greatly favours the application scalability, while allowing an adequate distribution of the workload in both homogeneous and heterogeneous systems, which has a large impact on the performance. Also the overhead due to synchronization or communication between processes or threads is minimised.

This work addresses the optimization of financial applications that allow a prediction of risk over time, for financial derivative products, particularly in multi-value environments. The selected model is the Heath-Jarrow-Morton (HJM) framework, which has a high computational cost [4,5]. This paper presents a new and efficient implementation of the HJM Model that can run on heterogeneous and massively parallel architectures. In particular optimization and parallelization code techniques will be used, for homogeneous (multi-core architectures) as well as heterogeneous environments (Multi-GPU). Furthermore, it shows that this implementation provides excellent results in performance, scalability and energy efficiency. The last point is essential, as these applications running on large data centres where improving the energy efficiency is one of his greatest challenges today.

2 Related Work

Several works with the same objectives can be found in the bibliography. For instance, Swaptions is an Intel implementation of the HJM model present in the parsec benchmark suite [6]. This implementation is restricted to swaps, while the work presented in this paper is a large application that is able to work on many different derivatives, specified in input files as trees of large vectorial operations. The Swaptions Parsec benchmark has been ported to a GPU architecture [7] with a very poor performance. The source of this problem is the great amount of data copy needed and the thread divergence happening in the GPU. With the use of newer CUDA versions, we can keep the data in the GPU all the time and execute basic SIMD kernels in an asynchronous manner. This allows fast execution without data transfers between each kernel execution and eliminates thread divergency. The original Swaptions version was a large monolithic computational kernel performing many memory allocations within, and confusing memory access patterns. The HJM implemented in this work relies on the decomposition into smaller kernels.

Moreover, [8] presents an implementation of a Monte Carlo model to estimate the current value of an European option for future purchase in the financial derivatives market, based on the Black-Scholes model. The imple-

mentation was done in four very different computer systems: A shared memory multi-core, a MPI cluster, a CUDA program over a GPU and a cluster of FPGAs where the most time consume computations were implemented in VHDL. Similarly [9] presents the design and implementation of a parallel version of a Monte Carlo method in a FPGA-based supercomputer, called Maxwell, of Edinburgh University [10]. The FPGA-based implementation is compared to other environments with various GPUs and conventional processors.

In addition, [11] also uses clusters of CPUs and GPUs to implement the calculation of the price of European options. They compare different systems and implementations in terms of performance and power consumption. Many financial applications rely on solving systems of sparse linear equations. For example, [12] proposes the design of a number of iterative methods for solving equations, based on the Krylov subspace, on GPU architectures. In this work, the proposed approaches are validated by solving the partial differential equations of the Black-Scholes model.

As far as we know this is the first paper where the HJM model is implemented on a massively parallel architecture, like a Multi-GPU and Multi-CPU system. Additionally, this paper proposes a study of the performance of this kind of applications in heterogeneous environments, from two different points of view: the improvement of performance (both response time and throughput) and scalability, as both are important in financial applications. Finally, a study on the power consumption and cost of these architectures is also shown.

3 Interest Rate Models

During the past three decades, derivatives have become increasingly important in the world of finance. A derivative is defined as a financial instrument whose value depends on the values of other, more basic underlying variables. Very often the variables underlying derivatives are the prices of traded assets. Some major developments have occurred in the theoretical understanding of how derivative asset prices are determined and how these prices change over time. This led to the use of advanced mathematical methods. Models based on the original Black-Scholes assumptions [13] are straightforward. However, they have simplistic approaches and assumptions when tackling exotic options.

Therefore, a number of alternative new models have since been introduced in an attempt to solve this problematic. These models, such as the Hull White, the Vasicek, the Cox Ingersoll and Ross model, incorporate a description of how interest rates change through time. For this reason, they involve the building of a term structure, typically based on the short term interest rate r_t. The main advantage of these methods lies in the possibility of specifying r_t as a solution to a Stochastic Differential Equation. This allows, through Markov theory, to work with the associated Partial Differential Equation and to subsequently derive a rather simple formula for bond prices. This makes them widely suited for valuing instruments such as caps, European bond options and European swap options.

However, they have some limitations and all lead to the same drawback when solving interest rate products: the fact that they only use one explanatory variable (r_t) to construct a model for the entire market. It proves insufficient to realistically model the market curve, which appears to be dependent on all the rates and their different time intervals. Consequently, these models cannot be used for valuing interest rate derivatives such as American-style swap options and structures notes, as they introduce arbitrage possibilities.

3.1 Heath-Jarrow-Morton (HJM) framework

The most straightforward solution to the problem mentioned above should include the use of more explanatory variables: long and medium term rates. The Heath-Jarrow-Morton framework (HJM) uses one representative short term rate, a middle term rate, and finally a long term interest rate [4,5]. It chooses to include the entire forward rate curve as a theoretically infinite dimensional state variable. Unlike other models, this one can match the volatility structure observed in the market today, as well as in the future.

The HJM is a general framework to model the evolution of interest rates. It describes the behaviour of the future price (in time) of a zero coupon bond paying one unit of currency at time T, and it provides a consistent framework for the pricing of interest rate derivatives. The model is directly calibrated to the currently observed yield curve, and is complete in the sense that it does not involve the market price of interest rate risk.

The key aspect of HJM lies in the recognition that the drifts of the no-arbitrage evolution of certain variables can be expressed as functions of their volatilities and the correlations among themselves, so no drift estimation is needed. HJM-type models capture the full dynamics of the entire forward rate curve. In practice, we will not work with a complete, absolutely continuous discount curve. Instead, we will construct our curve based on discrete market quotes, and will then extrapolate the data to make it continuous. Given the zero-coupon curve $B(t,T)$, there exists a forward rate $F(t,u)$ such that:

$$dF(t,T) = \mu(t,T)dt + \sigma(t,T)dWt_t^P \tag{1}$$

The HJM model has the serious disadvantage that it cannot be represented as recombining trees. In practice, this means that it must be implemented using Monte Carlo simulations. Therefore, it is important to use high performance architectures in order to minimize response times.

4 Optimization of the HJM Model

4.1 Analysis and Optimization of Sequential Code

The starting point is a sequential code that implements a multivalued prediction risk values model based on HJM [14], using a Monte Carlo method.

This code was implemented in C++ language with the Intel MKL library. On this version a code profile using *gprof* and *Valgrind* has been done. The profile has been performed both with and without MKL to verify the impact of this library on performance. This profile shows that the 41.67% of the runtime is spent in exponential function of MKL. The remaining time is consumed mainly in other vector operations. Specifically the *operators*, a set of functions that performs simple operations on all the elements of several vectors that are calculated in a step of the simulation. The use of MKL library has a significant effect on performance, provided that use Intel processors. The execution time is reduced by 48%, reaching a speedup of 1.92 compared to the version without MKL. This improvement comes from both the optimizations performed in the own library and the fact that it uses multi-threading.

Valgrind revealed a large number of memory conflicts. These conflicts produce *Cache Jamming*, consisting of two variables constantly overwritten in the cache, resulting in a large number of replacements and thereby causing a strong performance degradation. By changing the memory allocation scheme of the variables involved, this effect has been eliminated.

Finally, since the application uses several arrays with a large number of double-precision data, the effect of the cache in the application performance has been studied. Thus, the runtime has been measured on a processor with the same features, but a size of second-level cache (L2), which is 3 times higher per core. The results shows that the improvement in the response time for commonly used sizes is around 6%.

4.2 Replacing MKL Library

The use of the MKL library has a strong impact on application performance. However, it has two major problems: the economic cost is very high and only takes advantage on Intel processors, so that limits code portability. For these reasons, it is proposed to search for an open source solution, to replace the functions of MKL used in this application. The solution proposed in this paper is an approach based on SLEEF[15] Library and AVX instructions are proposed. Two different approaches have been developed, the first based on a single thread and using vector instructions, and the second using multi-threading.

The application consists of several simple kernels that perform basic operations in large vectors. Hand-coded AVX instructions can be used in almost all kernels. However, some kernels uses transcendental functions, such as the exponential, which are not implemented on the vectorial instruction set. SLEEF ,is an open-source library that implements transcendental functions using SSE or AVX. Hence, it is the perfect replacement for the remaining kernels.

The version that uses only SLEEF and hand-coded AVX instructions, reduces time consumption with respect to the original version without MKL, but nonetheless it consumes around 20% more time than MKL. This is because SLEEF is single-threading, while MKL adjusts the number of threads to data

size. If a version SLEEF and multi-threading in the same areas of MKL is used, a very similar result is obtained (only 3% worse). MKL uses highly optimized routines with details of the processor architecture that are not public, so to get exactly the same performance is a non trivial task. This analysis leads to an interesting conclusion: MKL can be replaced by an alternative open source without losing performance, thus allowing to generate a more portable and less economic cost code.

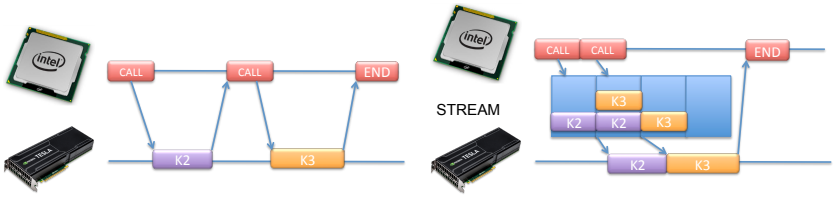
4.3 CUDA Implementation

A first aspect to analyse is the communication between CPU and GPU as it is one of the main bottlenecks in the Host-Device programming model. The application is iterative, that is, it performs a set of calls to CUDA kernels, one for each step of the simulation. The kernels execute the most computational cost operations, such as exponential and division of double-precision numbers, on large vectors which are independent from each other.

A detailed analysis of the data dependencies between successive iterations shows that the results of the partial vector operations, performed at each simulation step, are not needed until the end of the execution, and hence they are always stored in the memory of the GPU. This has a double impact on performance: synchronization points between CPU-GPU are avoided and the transfer of information between the two devices is minimized. This is implemented through the use of *CUDA streams*. Each call to a CUDA operation is queued into a stream and the application can continue executing on the CPU asynchronously. The stream manages the execution of CUDA kernels while the CPU is computing the control structures and queues new CUDA kernels operations. The CPU waits for the GPU only when reading the final results, rather than once per transaction.

Figure 1(a) shows the execution flow of the synchronous case. When a CUDA operation is running on the GPU, the processor remains idle waiting until it ends. During this idle time the processor could submit new work to the GPU or perform independent CPU tasks which would not require pending results. On the other hand, Figure 1(b) shows how the CPU queues, a CUDA task in a special buffer, the *stream* and continues running other part of the code. The synchronization is only needed when a transfer of data is essential.

Another overhead is the CUDA initialization, in which the CUDA driver creates the memory maps, initializes registers and contexts and finally loads the code into the GPU. All these steps, with exception of the last one, may be made prior to the execution of the application. To solve this problem a *Client-Server* architecture has been designed, based on UNIX Domain Sockets. The *Server* initializes the GPU and is listening to a socket which is mapped in the file system, awaiting execution requests coming from the *Clients*. The *Clients* connect to the *Server* using this socket passed as a parameter to the application, and they send the name of the file containing the kernel to be run. The *Server* runs the kernel on the GPU and returns the result to the



(a) Synchronous CPU-GPU Execution (b) Asynchronous CPU-GPU Execution

Fig. 1 Synchronous vs. Asynchronous CPU-GPU execution with CUDA

Client. With this architecture, initialization is performed only once, at boot time of the machine. Thus the individual processes prevent overload time. The NVIDIA K20 has an initialization time of about 100 ms. The execution times of kernels in this application is about 40 ms. Hence, it is clear that the initialization overhead has an strong impact on the application response time.

Finally, it is important to highlight that the implementation is Multi-CPU and Multi-GPU, i.e. it supports the execution of a single job on multiple GPUs in parallel. An input parameter determines the maximum number of GPUs that can be used in each run. In the case of using more than one GPU, workload is distributed statically, at the beginning of the execution. Moreover, the workload is evenly distributed among all the GPUs in the system.

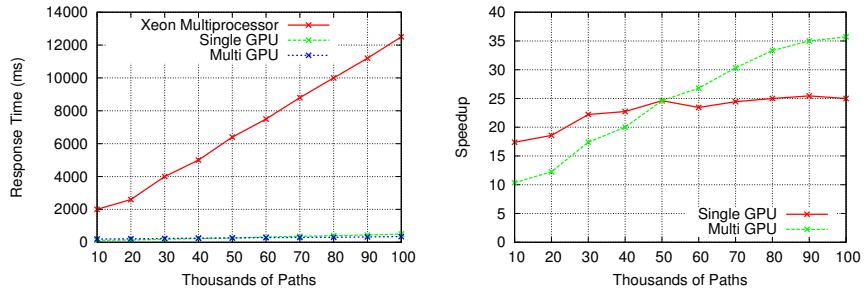
5 Experimental Results

This section presents a set of experimental results. The main objectives of these experiments are to perform an in depth study of the performance of the proposed approaches, varying the number of paths executed, in terms of response time and throughput. On the other hand a detailed analysis of the energy efficiency for all the tests is reported.

The experiments have been developed on a Intel server with a dual Intel Sandy Bridge E5-2620 2 GHz processor with 6 cores each one. The server has 15 MB of L3 cache memory and 16 GB of DDR3 main memory. The system runs a Ubuntu 10.04 Linux operating system, and has the CUDA 5 and Intel MKL Library. The server comprises two NVIDIA Kepler K20 GPUs with 2496 cores and 5 GB of DDR5 memory. Each GPU has its own dedicated PCI-express 3.0 bus between the GPU and the CPU, to avoid collisions in the access to the bus.

To evaluate our approach three different environments and implementations have been developed:

- **Multi-Thread** application running on a multiprocessor with 12 cores. Its TDP is 270 watts.
- **Single-GPU**, a heterogeneous Host+CUDA application running on a single GPU, with a TDP of 495 watts.
- **Multi-GPU**, a heterogeneous Host+CUDA application, running on two GPUs, with a TDP of 720 watts.



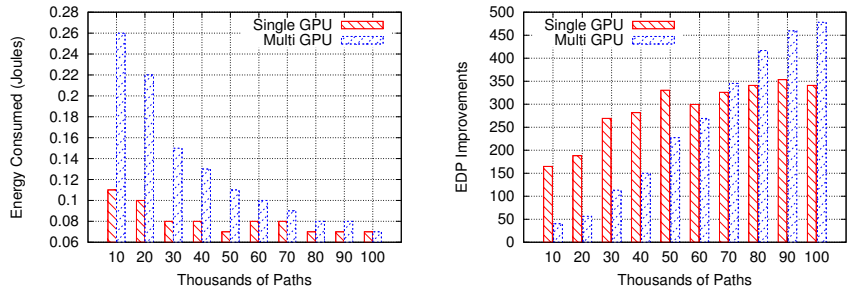
(a) Response Times (b) Speedups

Fig. 2 Response Times and Speedups of the proposed architectures

All results presented in this section refer to the implementation in double precision. The metric used is the *response time*, in milliseconds, defined as the total execution time since the application is launched until results are obtained. Therefore, it includes both computing and communication between CPU and GPU times. Then, the speedup is calculated as the ratio between the response time of the Multi-Thread version with respect to the response time of the different GPUs implementations. The results presented are always the average obtained from 10 independent runs. The energy studies are based on the Thermal Design Power (TDP). In the systems with GPUs the values of TDP, take also into account the power consumed by the server. The metrics used for analysing the energy efficiency are the Energy consumption and the Energy-Delay-Product (EDP). Then, the improvements in EPD are obtained as the ratio between the EDP of the Multi-Thread version and the EDP of the different GPUs implementations.

The first result that is important to highlight is the large reduction in response time that occurs when using the heterogeneous system, as can be seen in Figure 2(a). On the other hand, Figure 2(b), presents the behaviour of the speedups of the heterogeneous environments compared to the multiprocessor as problem size increases. In that figure, it can be seen that below 50.000 paths the speedup of the single GPU is significantly higher than Multi-GPU. This behaviour is due to the fact that the workload is too small and therefore the advantages of using two GPUs simultaneously can not improve the overhead to manage them. However, as the workload grows, the benefits of using two GPUs in parallel outperform this overhead, and this behaviour becomes more noticeable as the problem size grows.

Energy efficiency is one of the key problems in this type of applications, since financial institutions have large data centres mainly devoted to the execution of such models. Reducing the energy footprint is one of the greatest challenges of data centres. In this regard, Figure 3(a) plots the relative energy consumption of the Single-GPU and Multi-GPU systems, normalized to the energy consumed by the Xeon server. As can be seen, for medium and large size problems, the energy efficiency of heterogeneous configurations is very good,



(a) Energy Consumption (b) EDP improvements

Fig. 3 Energy Efficiency for the heterogeneous architecture with respect to the Xeon Server

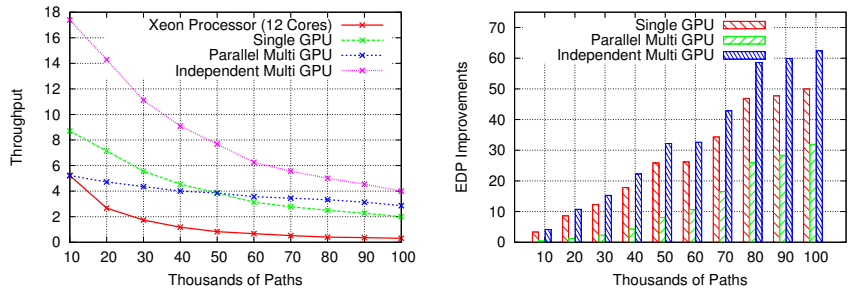
both yielding a energy consumption of around 7% of the Xeon Server. The Single-GPU system soon reaches the minimum and stabilizes around it. The Multi-GPU system is penalized in small problems by two factors: increased overhead in managing the data (lower speedup in performance) and higher energy consumption.

On the other hand, Figure 3(b) shows the energy efciency of each platform by plotting the gain of the EDP calculated with respect to the Xeon Server. The higher the value, the more energy efcient. It is necessary to highlight that these gain values are very high, confirming the excellent behaviour of the GPUs, in terms of energy efficiency. It can be observed that for small problem sizes the EDP of the Single-GPU platform is better, because the improvement in time achieved by introducing a second GPU does not outperform its energy consumption. However, as the problem size grows, the overheads of managing the second GPU are mitigated by the gain in time, and therefore also in EDP.

In the second experiment, the behaviour of the heterogeneous architectures is evaluated in terms of throughput and EDP per task. The metric used is not the response time of a single instance of the problem, but the throughput of the system, i.e. the number of tasks that can be completed in a certain time interval. Therefore multiple independent instances of the same problem are running simultaneously, while time and energy have been measured. In the case of Xeon multiprocessor these instances are not parallelized (i.e. each job runs on a single core, so it can run 12 instances simultaneously).

Figure 4(a) shows the throughput results obtained in all these available systems, as the number of paths increases. It can to be noticed that the environment with higher throughput is the Multi GPU, but using each GPU independently on a single instance of the problem. Furthermore, it is interesting to highlight that only one K20 GPU performs more operations per second than 12 Xeon cores running independent simulations.

With respect to energy, Figure 4(b) presents the gains of the EDP per task with respect to the Xeon Server. The higher the value, the more energy efcient. Based on these results it can be concluded that the implementation that uses the two GPUs independently is the most efficient from the energy point of



(a) Throughput of different configurations (b) Gains of the EDP per task

Fig. 4 Throughput and EDP improvements of the platforms varying the number of paths

view. It is interesting to point out that in the second place is the Single-GPU architecture whose values of EDP are much higher that the Parallel GPU architecture.

Finally, the results achieved in the throughput with the GPUs, have a significant economic impact on the cost of the system. By using a single server with two E5-2620 processors and two GPU cards K20 performing simulations in parallel, it is possible to replace 10 servers without GPU.

6 Conclusions and Future Work

The most important and general conclusion to highlight is that the financial models based on Monte Carlo methods, such as the HJM, have qualities that make them especially suitable for implementation on massively parallel architectures, especially in Multi-GPU platforms. Indeed, the massive data parallelism along with data independence allows to squeeze the full potential out of the GPUs. Furthermore, these model minimize communication between CPU and GPU, that is one of the major bottle-necks in this architecture. Finally, this data independence also allows a balanced distribution of workload and offers excellent properties regarding scalability.

This suitability is proven in the experimental results of performance and energy efficiency presented in this paper. To summarize, it is noteworthy that a heterogeneous architecture with an NVIDIA Kepler K20 GPU can achieve a speedup of more than 35 over the best version on CPU. Furthermore it has been shown that this architecture provides excellent scalability: the higher the workload, the better the speedup is. Finally, it is worth mentioning the excellent energy efficiency of these architectures, with great EDP improvements, as well as its excellent cost/performance ratio.

In Multi-GPU environments, the workload is the key parameter when deciding whether the application runs on a single GPU or uses several in parallel. The experimental results for the HJM model show that the use of the two GPUs in parallel is profitable from a workload paths 50,000. In more complex models with a higher cost of computation, this value can vary substantially.

Future work includes providing the Multi-GPU environment with a load balancing mechanism that allows a heterogeneous distribution between GPUs with different performance. Likewise, other accelerator architectures such as Intel Xeon Phi will be explored.

Acknowledgements The authors would like to express their gratitude to **François Friggit** of Banco Santander who inspired and motivated this challenge as a real business case and provided all necessary assistance to carry out this work. This work has been supported by the Spanish Science and Technology Commission (CICYT) under contract TIN2010-21291-C02-02, the European Unions FP7 under Agreements ERC-321253 (RoMoL) and ICT-288777 (Mont-Blanc) and by the European HiPEAC Network of Excellence. This project has also been partially funded by the JSA no. 2013_119 as part of the IBM/BSC Technology Center for Supercomputing agreement.

References

1. Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003.
2. Gareth W. Morris, Matthew Aubury. Design space exploration of the European option benchmark using hyperstreams. In *International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands*, 2007.
3. Virat Agarwal, Lurng-Kuo Liu, David A. Bader. Financial modelling on the cell broadband engine. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–12, 2008.
4. David Heath, Robert Jarrow, Andrew Morton. Bond pricing and the term structure of interest rates: A discrete time approximation. *Journal of Financial and Quantitative Analysis*, 25(04):419–440, December 1990.
5. David Heath, Robert Jarrow, Andrew Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica*, 60(1):77–105, January 1992.
6. Christian Bienia, Sanjeev Kumarand, Singh Jaswinder Pal Singh, Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications *Proceedings of the 17th Int. Conf. on Parallel Architectures and Compilation Techniques*, 72–81, 2008.
7. Matthew Sinclair, Henry Duwe, Karthikeyan Sankaralingam. Porting CMP Benchmarks to GPUs *Computer Sciences Department, Technical Report 1693* , University of Wisconsin, Madrison, June 2011.
8. Javier Castillo, José Luis Bosque, Emilio Castillo, Pablo Huerta, and José Ignacio Martínez. Hardware accelerated montecarlo financial simulation over low cost fpga cluster. In *IPDPS*, pages 1–8, 2009.
9. Xiang Tian, Khaled Benkrid. High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):26:1–26:22, November 2010.
10. Rob Baxter, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, and Arthur Trew. Maxwell - 64 fpga supercomputer. *Engineering Letters*, 16(3), 2008.
11. L.A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. High dimensional pricing of exotic European contracts on a gpu cluster, and comparison to a cpu cluster. In *Parallel Distributed Processing, IEEE International Symposium on (IPDPS)*, pages 1–8, 2009.
12. A. Gaikwad and I.M. Toke. Parallel iterative linear solvers on gpu: A financial engineering case. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 607–614, 2010.
13. Fishcer Black, Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.
14. Martingale Musiela. *Methods in Financial Modeling.* Springer-Verlag, 2 Ed., 2004.
15. Naoki Shibata. Efficient evaluation methods of elementary functions suitable for SIMD computation. Computer Science - Research and Development, vol 25, issue 1-2, pages 25-32, 2010.