2012 IEEE 4th International Conference on Cloud Computing Technology and Science

Towards Parallel and Distributed Computing on GPU for American Basket Option Pricing

Michaël Benguigui, Françoise Baude
INRIA Sophia-Antipolis Méditerranée, CNRS I3S, University of Nice Sophia-Antipolis
First.Last@inria.fr

***Abstract* — This article presents a GPU adaptation of a specific Monte Carlo and classification based method for pricing American basket options, due to Picazo. Some optimizations are exposed to get good performance of our parallel algorithm on GPU. In order to benefit from different GPU devices, a dynamic strategy of kernel calibration is proposed. Future work is geared towards the use of distributed computing infrastructures such as Grids and Clouds, equipped with GPUs, in order to benefit for even more parallelism in solving such computing intensive problem in mathematical finance.**

Keywords-component; Distributed and parallel computing, Grid, Cloud, GPU, OpenCL, machine learning, mathematical finance, option pricing

I. INTRODUCTION: HIGH PERFORMANCE COMPUTING IN FINANCE

A growing number of problems can benefit from High Performance Computing. In nuclear physics, for example HPC and Monte Carlo (MC) methods are combined to simulate radiation interaction. In market finance, HPC becomes unavoidable in arbitrage trading or hedging. In 2009, 10% of Top500 supercomputers are employed for financial calculus [1]. Many financial measures are stochastic calculus and require a high number of simulations. High dimensional option pricing with fine time discretization quickly becomes a challenging problem that is suitable to MC methods. Value at Risk calculation for instance may cope with large number of sophisticated assets. Hedging strategies require rapid adaptation and automated trading algorithms need fast execution to make gain opportunities. Code optimization techniques improve program performance such as algorithm improvement, code vectorization for vector processor. However this is not sufficient and adequate for one or many GPU usage for high parallelization degrees. Besides, Cloud usage can be a good reason to outsource technology-intensive trade processing to larger financial company to take benefit from new developments [2]. In other cases, some financial establishments using grid computing with CPU pointed out excessive cost in hardware and electricity consumption. This is the case of Aon Benfield, a world's leading insurance company which for a bond pricing service spent $4 million in a grid architecture using CPUs and $1.2 million in electricity a year [3]. On the contrary, a GPU based pricing engine will only cost $144,000 and $31,000 in electricity a year. For instance, for J.P. Morgan's Equity Derivatives Group, the equity derivative-focused risk computation is performed on hybrid GPU-based systems, increasing performance by 40x compared to only CPU-based systems, for the same electric power [4]. Exploiting heterogeneous CPU and GPU resources tend to be a topical problem for intensive financial activities.

Machine learning covers many scientific domains (image processing, particle physics...) and was introduced in option pricing problems by Picazo [5] giving then a powerful and easy way to parallelize algorithm for distributed architectures such as computing Grids [6]. However, particularity of GPU implementation is that all cores in the same block execute the same instruction at the same time (Single Instruction, Multiple Thread model). The main goal of our present work is to define and implement a GPU approach for this option pricing algorithm based upon the Picazo method, keeping in mind that our future goal is to further parallelize the pricing by relying upon distributed GPUs nodes acquired from computing grids or Clouds. As such, heterogeneity in acquired GPU devices will have to be addressed. We explain in this paper how we handle this heterogeneity requirement within the proposed GPU implementation.

In section II we will present a CPU/GPU implementation of Picazo method and explain our adaptation to tackle the warp divergence of loop condition at the algorithmic level. In section III, we will consider more technical aspects and propose a solution that enables to adapt our code to any NVIDIA GPU device. We will also suggest portability to AMD GPUs. Evolution over heterogeneous distributed Cloud/GPU architectures will be discussed in section IV and some related works presented in section V before concluding.

II. A GPU BASED ALGORITHM OF PICAZO PRICING METHOD

Here we describe Picazo algorithm before focusing on our GPU adaptation. Last section shows experiments which prove benefits of our strategy to tackle the warp divergence of loop condition.

A. Pricing Algorithm

Forecasting financial instrument prices is a challenging task in market finance. High dimensional American basket call/put option is a contract allowing you to buy/sell at a specified strike price K, a possibly high size, (e.g. 40) set of underlying stocks S^i_t at any time until a maturity date T. So a call owner expects underlying asset prices to rise over strike. American basket option pricing is a typical problem requiring a lot of time and memory resources for resolution. There is no analytic solution and some numerical methods such as finite difference methods cost too much computational time to get accurate

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

results. Monte Carlo methods, based on the law of large number and central limit theorem, allow a simplified approach for high complex problems, reaching good accuracy in reasonable time. Consider $S_t^{(s)}$ as independent trajectories of an underlying asset price following Black and Scholes model, $\Psi(f(S_t^{(s)},t))$ as the option pay-off, N as the discrete time number, r as the risk free rate. American option price V at time zero can be estimated as follows:

$$V(S_0,0) \approx \frac{1}{nbMC}\sum_{s=1}^{nbMC} e^{-rT}\Psi\left(f\left(S_t^{(s)}, t \in [0,T]\right)\right)$$

As opposed to European option price, American option price must reflect all possible opportunities to exercise option until maturity date. This possibility is reflected in the mathematical definition as follows:

$$V(S_T,T) = \Psi(S_T,T)$$
$$V(S_{t_m},t_m) = \max\left(\Psi(S_{t_m},t_m), \mathrm{E}\left[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}},t_{m+1})\,|\,S_{t_m}\right]\right)$$

We define $\mathrm{E}\left[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}},t_{m+1})\,|\,S_{t_m}\right]$ as the continuation value. Some methods compute directly a boundary exercise line to decide to continue or not to hold the option [7], before performing final MC simulations as sketched in FIGURE I [phase 2]. Instead of computing an explicit boundary exercise line, Picazo method relies upon continuation/exercise regions. It exposes an efficient way to define these regions by combining a machine learning technique with MC methods (FIGURE I [phase 1]). Consider S_t^i as asset prices with $i = 1..d$, d as the asset number, δ_i as dividend rates, σ_i as volatility rates, *nb_class* is the number of training instances per classifier, *nb_cont* is the number of MC simulations to compute the continuation values.

FIGURE I. PICAZO ALGORITHM RELYING UPON CLASSIFICATION

Require: S_0^i, d, r, δ_i, σ_i, T, N.
Require: number of classification points *nb_class*,
Require: number of trajectories to estimate each continuation value *nb_cont*
Require: number of trajectories to estimate the final option price $nbMC$
1: **[phase 1] :**
2: **for** $m = N-1$ to 1 **do**
3: Generate *nb_class* points of $\{S_{t_m}^{i,(s)} : i = 1,\ldots,d; s = 1,\ldots,nb\_class\}$.
4: **[step 1] :**
5: **for** $s = 1$ to *nb_class* **do**
6: Compute $C^{(s)}(S_{t_m},t_m) = \mathbb{E}\left[e^{-r(t_{m+1}-t_m)}V(S_{t_{m+1}},t_{m+1})|S_{t_m}\right]$ using *nb_cont* trajectories and also compute $\Psi^{(s)}(S_{t_m},t_m)$.
7: **if** $C^{(s)}(S_{t_m},t_m) \le \Psi^{(s)}(S_{t_m},t_m)$ **then**
8: sign = 1
9: **else**
10: sign = -1
11: **end if**
12: **end for**
13: **[step 2] :** Classify $\left\{(S_{t_m},\mathrm{sign})^{(s)} : s = 1,\ldots,nb\_class\right\}$ to characterize the exercise boundary at t_m.
14: **end for**
15: **[phase 2] :** Generate new $nbMC$ trajectories $\{S_{t_m}^{i,(s)} : i = 1,\ldots,d; m = 1,\ldots,N; s = 1,\ldots,nbMC\}$. Using the characterization of the exercise boundary above, we can estimate the final option price.
16: return the final option price.

To compute the final option price using MC simulations in [phase 2], a classifier for each discrete time is needed to ensure the algorithm can decide if the running simulation must be stopped or not, i.e. given simulated asset prices, it forecasts if an exercise opportunity will come. We create all the classifiers during [phase 1]: for every time (line 2), we generate a set of training instances (line 5) to train a new classifier. Each training instance is composed of simulated asset prices and a sign according resulting payoff is over (+1) or not (-1) the continuation value (line 7). Because continuation values require relying upon classifiers during Monte Carlo simulations in [phase 1] [step 1], Monte Carlo simulations are backward computed: at time *(T-1),* classifiers are not needed because simulations reach instantly maturity, and by this way last classifier can be trained (line 6.). Then starting from *(T-2)*, simulated asset prices at *(T-1)* can be classified, and so on backward N times, until *T=1*.

B. *Algorithm Parallelization for a GPU*

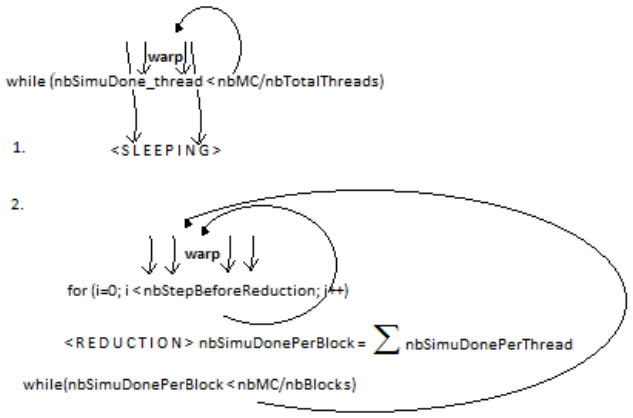
We now consider the critical sections of the algorithm to parallelize. Backward loop cannot be parallelized because classifier at time *m* is used to train classifier at time *(m-1)*. Our previous work [6] proposed however a distributed implementation upon Clouds relying upon a set of a moderate number of slave nodes (up to few hundreds) and one master node. The training instance computations are performed through independent simulations. As a result at every discrete time, each node asynchronously can handle a set of training instances, storing their results back to the master node before going on with new computations if needed. By this way, the nodes having a task completed earlier need not wait for the others, reducing the bottleneck effect. The parallelization degree depends on the number of *nb_class* training instances which is clearly several orders of magnitude lower than the amount of required MC simulations (thousands, millions) for one training instance computation. Simply applying this same design in the context of a single GPU would not allow us to benefit from the GPU high thread number, as this thread number can be much larger than *nb_class*.

A single GPU offers more cores than a common distributed architecture (more than a thousand for NVIDIA Kepler or AMD Tahiti architecture) and MC simulation numbers must be large enough to get a good approximation of the expected value. Thus we propose a new design along the following main idea: the *nb_cont* MC simulations to compute a single training instance are parallelized on the GPU threads. More precisely, each thread performs at every discrete time (during its simulations) the following operations: generates random uniform variables, applies Gaussian transformation, correlates them if specified, simulates asset prices, predicts exercise situation through a classify function, stores actualized payoff. Contrary to a distributed architecture, all cores on the GPU are identical and no load balancing is needed. The main difficulty to adapt American option pricing problem to SIMT architecture comes from the variable length of simulations amongst all the *nb_cont* ones. It prevents to ask each thread to perform predefined number of trajectories that would be fixed at the beginning of line 6. A warp (for NVIDIA architecture or a wavefront for AMD) is the smallest quantity of threads that are

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

issued with a SIMT instruction. Threads of the same warp cannot perform at the same time different instructions, resulting in implicit synchronization barriers. Consider we distribute the same number of simulations per thread, those performing short-length simulations (requiring less time steps because of the American option behavior which dictates to exercise the option as soon as possible) would wait for the others of the same warp. This leads to unwanted synchronizations and low occupancy. Occupancy is the ratio of active warps per multiprocessor (per Compute Unit i.e. CU) to the maximum number of possible active warps. We need to consider performance degradation that occurs if the occupancy is not high enough to hide memory latencies, even if increasing it brings no more performance at a certain level [8]. A way to avoid this termination divergence is to apply a suitable behavior in our design (FIGURE II). Threads inside a warp work synchronously. To reduce internal block (composed of warps) waiting time, we compute after *nbStepsBeforeReduction* time steps and with an intermediate reduction, how many MC simulations have been achieved. This is repeated until at least the total number of MC simulations needed for getting a continuation value has been achieved (FIGURE II 2). Obviously the value of *nbStepsBeforeReduction* is different for each *nb_class* continuation value computation.

FIGURE II. BEHAVIOR OF A WARP IN A BLOCK: 1. WITHOUT INTERMEDIATE REDUCTIONS. 2. WITH INTERMEDIATE REDUCTIONS



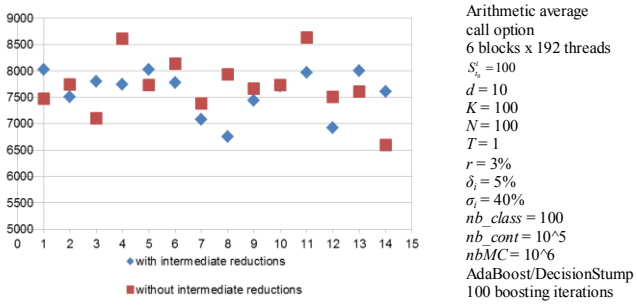
nbStepsBeforeReduction is computed at runtime as follows: after the very first parallel simulations for a given continuation value, each thread of the block provides its stopping time (i.e. given the initial asset prices, at which time the decision to exercise the option is taken). For the subsequent batches of MC simulations to run, we set *nbStepsBeforeReduction* as being the maximum of these stopping times. Indeed, as to compute the continuation value all threads start from same simulated asset prices, we can notice that each thread is more or less as far as the others from exercise regions. This is more significant with bigger drift part than martingale part. So there is no need to perform early intermediate reductions although they are quickly achieved through dynamically allocated shared memory. We have conducted specific tests that reveal that even a high number of reductions do not impact global execution time.

C. *Performance Evaluation*

The exposed approach targets American options that exhibit fine time discretization, and whose pricing features large classification parameters. Indeed coarse discretization does not exhibit strong differences between simulation lengths among threads. So, in the sequel, we only plot some experiments conducted with a fine time discretization (N=100).

Option pricing is typically a stochastic problem, so, for same parameters, computation time can vary significantly. Our strategy rounds up time step numbers to complete simulations (FIGURE II 2) because we insure a fixed number of simulations is at least performed per block before stopping reductions. So overall we may run more MC simulations than needed. However we reduce time peaks due to bottleneck effects of random simulation sizes and we are able to smooth the effects of randomness when calculating each continuation value. As illustrated by FIGURE III, our implementation allows decreasing the stochastic impact on classification durations (i.e. the whole [phase 1] of our algorithm): this is a concrete proof of the effectiveness of our strategy, and more importantly in real situations it could allow to better foresee the total amount of option pricing computation time.

FIGURE III. TOTAL CLASSIFICATION TIMES IN SECONDS FOR 14 EXPERIMENTS



III. GPU SPECIFIC IMPLEMENTATION

Unlike CPUs, GPUs do not allow to work with advanced libraries. This firstly section aims at explaining how to cope with this constraint. Because this work is ultimately dedicated to distribute GPUs which might be heterogeneous, we propose then a strategy to dynamically calibrate kernel parameters.

A. *OpenCL implementation details*

In order to classify values, we aim to rely as much as possible upon an existing Java machine learning library (Weka library [10]). Thus we require that all the non-GPU sequential part of our pricing algorithm be expressed in Java. However we intend to exploit most popular graphic cards (NVIDIA, AMD, Intel) though OpenCL (Open Computing Language [9]), a C-derived language interfaced with the C++ language. So we decided to use OpenCL through JOCL [11] which itself is a JNI wrapper around OpenCL keeping readable end-user Java code on the CPU side (without explicit JNI calls).

We could not however rely upon all the needed Java methods provided by Weka, because a GPU thread is not able

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

to call a method of an external library. So we had to mimic the behavior of some of the needed Weka methods directly in OpenCL as explained below.

The Weka library allows us to create AdaBoost (Adaptive Boosting) or SVM (Support Vector Machines) based classifiers, by training them with training instances. This allows the pricing algorithm to subsequently make classify calls onto a previously built classifier to predict new instances. Indeed, during the computation of continuation values and final option price, we must know at every time until maturity, giving simulated asset prices, if it is suitable or not to exercise option contract, therefore stop or not the current MC simulation. Taking the decision is easy as soon as the code can access to the already trained classifier. Weka allows both to build a classifier (as an object), and subsequently to call a specific method (named *classify*) upon it for the decision making. However making such a call from a GPU thread is impossible. The solution we have developed consists in (a) making a copy of the required classifiers hold by Weka in the GPU global memory at the beginning of each new loop, which requires to use an adequate data structure to represent the classifiers; (b) providing a function in OpenCL which mimics the behavior of the classify Weka method on a classifier, and have GPU threads use this function and not the one provided by Weka in the parallel phase of the algorithm. The classifier training cannot be delegated to the GPU due to the lack of memory resources. Once trained, the classifiers instances hold at the Weka side are the ones that are serialized and injected in the GPU memory as described in (a). OpenCL does not allow complex structure usage such as multidimensional arrays. So, we defined the serialization output of the object instances representing the needed Weka classifiers as simplified data-structures (1-dimensional arrays) filled with only the data members of the Weka objects that will be necessary in running the classification function defined in (b). Getting access to these data members required us to slightly modify the Weka open source code. There are as many classifiers as discrete times, and to store all of them, we work with 1-dimensional arrays grouping all same data members of all classifiers in the same array. This means we work with position indexes to access to data of a specific classifier. Notice that it is impossible to make sure member accesses from threads of a same warp are done in the same global memory segment, because threads are not necessary simultaneously at the same discrete time.

In order to perform global memory accesses in one memory transaction, we store simulated prices of each asset of the basket in a contiguous manner. Thus threads perform coalesced accesses for each to simulate their S^i_t before simulating S^{i+1}_t.

There are as many kernel launches as training instances (*nb_class*) per classifier. To overlap the overhead of launching a kernel, the loop at line 5 is reduced to *(nb_class/2)*, and at every turn we manage two kernels with separate command queues: while one kernel executes the other gets launched. This allows overlapping kernel executions with data transfers between the host and the device as highlighted in the best practice document [8]. The overlapping is only possible with page-locked allocations, and our data transfers are not large enough to benefit much from it. So overlap performance gain happens but is limited: for instance, an American basket option pricing takes 2073s against 2146s for a non-overlap implementation.

We use the OpenCL Mersenne Twister method implementation to generate inside the kernel uniform pseudo-random numbers (high quality and fast generation). We compute and transfer a global seed from host to global memory at every kernel launch. These seeds are fixed as `currentTimeMillis() * 1000 + nanoTime()`.

B. *Kernel adaptation*

Kernel parameter calibration is an essential step in our optimization. On mid-term, we aim at exploiting distributed GPUs in a Cloud and consequently coping with a wide range of GPUs. This parameterization can be achieved through several benchmarks to find the best kernel configuration. However, to perform best fit values at run-time and for a single GPU, time spent at start for benchmarks is not suitable in HPC. This strategy must also be avoided for many heterogeneous GPUs even if distributed.

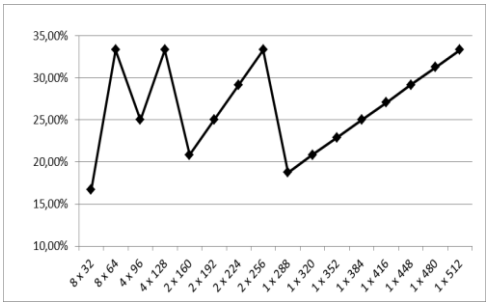
Because size of fast access memories is limited, our implementation requires many accesses in global memory. Furthermore many branch conditions encountered by threads depend on random asset prices leading to warp divergences, and we also use explicit barriers for computing the intermediate reductions, both of these introducing some wasted time. In our case to improve time execution, we need to hide latencies and keep hardware busy, which requires setting up kernel parameters in a way that will lead to the best multiprocessor occupancy when running the pricing algorithm. To perform a run-time kernel parameter calibration, we developed a dedicated Java class which imitates the NVIDIA occupancy calculator spreadsheet (this tool can be extended to AMD GPUs with occupancy formula given at [12]). With regards to the multiprocessor limitations (maximal warp number, shared memory, registers), the occupancy calculator computes number of active warps while taking into account program memory usage and block size. Statically just from the source code, we inform the calculator about registers and shared memory program usage, from which it computes at program start multiprocessor occupancies for all possible block sizes, starting from warp size to maximum block size allowed.

The occupancy calculator computes active block number per CU before deducting active warp number. FIGURE IV shows occupancy evolution over all possible block sizes and deducted active block numbers. Increasing block size will fall number of active blocks per CU. Shared memory usage input parameter changes with block size because it is used for block reductions. A given peak of occupancy is reached for every different active block number with maximal threads number. Given our program memory usage, the calculator deducts that the execution can benefit from a maximal occupancy (~33.33%) with blocks of 64, 128, 256 and 512 threads. Block size of 64 threads offers more active blocks per CU than others (8 against 4, 2 and 1). This occupancy does not take into account program the behavior, and we know from our algorithm behavior that maximizing the number of active blocks per CU could reduce waiting time between blocks (as

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

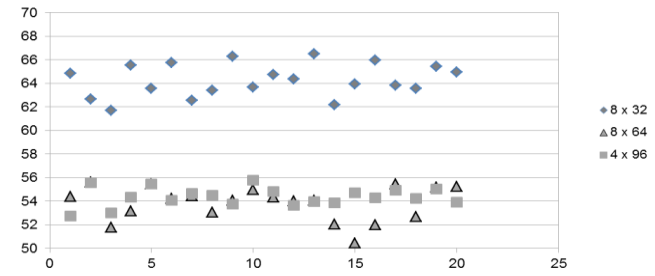
each block would be given a smaller simulations number to perform).

FIGURE IV. CU OCCUPANCIES OVER KERNEL CONFIGURATIONS (NUMBER OF ACTIVE BLOCKS PER CU X BLOCK SIZE)



We confirmed the relevance of occupancy when choosing block size by comparing the results given block sizes for 16.67%, 25% and 33.33% occupancy rate (FIGURE V). The NVIDIA Visual Profiler does not reveal increasing block number over active block number improve effective occupancy, and the total thread number has been set to `nbActiveBlockPerCU * blockSize * nbCU`. Such knowledge is taken into account to help decide at start time of the complete kernel parameters (block size and total thread number), given the output of the occupancy calculator.

FIGURE V. TOTAL CLASSIFICATION TIMES IN SECONDS OVER 20 PROGRAM LAUNCHES PER KERNEL CONFIGURATION. PROGRAM PARAMETERS ARE THE SAME THAN PREVIOUSLY



The GPU adaptation allows performing American basket option pricing in an acceptable time execution even with resources exhibiting moderate performance characteristics: the NVIDIA Quadro 600 or NVIDIA GTX 560M we are granted access to are not the high-end GPUs. Overall, this implementation benefits from a less expensive architecture than a distributed infrastructure like a Cloud with many nodes, featuring also lower energy consumption. However, GPU does not offer as much memory as a CPU, that lead to multiple small kernels launches and many data transfers between CPU and GPU in order to complete the pricing. In order to spread kernel launches among multiple GPUs, we will add in future work a new level of parallelism and adapt the parallelization strategy accordingly, as sketched briefly in the next section.

IV. TOWARDS COMBINING CLOUD/GPU TO RESOLVE LARGE SIZE PROBLEMS

We evaluate here the performance of our single GPU based approach over the traditional Cloud based approach. This latter was tested on ProActive PACAGRID, a computing Cloud operated by INRIA. ProActive API provides a job scheduler which facilitates the CPU resource allocation. We intend to employ it to extend our work over a heterogeneous GPU/CPU architecture. We defined an American basket option pricing featuring high dimension and high values regarding the classification parameters (legend of TABLE I).

TABLE I. Overall computational time with NVIDIA GTX 560M

Geometric average American call option
$S^i_{t^0} = 100$ $d = 40$ $K = 100$ $N = 50$ $T = 1$ $r = 3\%$ $\delta_i = 5\%$
$\sigma_i = 40\%$ $nb\_class = 5000$ $nb\_cont = 10^4$ $nbMC = 2*10^6$
AdaBoost/DecisionStump 150 boosting iterations

	64 cores at 2.3GHz from AMD Opteron 2356 series processors (1 Opteron provides 4 cores)	Tesla M2050 112 blocks x 64 threads
[phase 1] Total classification time	7 h 01 min 30 s	9 h 19 min 42 s
[phase 2] Final pricing time	0 h 53 min 12 s	0 h 00 min 27 s
Price (~10^-5)	0.70557 ± 0.00135	0.68976 ± 0.00147

Price for this option is reported (TABLE I) with 95% confidence interval (CI). Note that for such problems offering many adjustment parameters, CI can be biased by *nb_class, nb_cont, N, boosting iterations* number, and must be only considered to fix *nbMC*. These results reveal the GPU ability to resolve non-embarrassingly and large problem in same order of time than on a distributed architecture. More than low energy consumption and architecture cost, providing the same calculus performance level with fewer resources allows easier way to extend the architecture. An hybrid architecture seems to be a natural way to combine high parallelization of GPU device and large memory resources of Cloud, and this is why we plan to explore such hybrid combination in the near future.

To fully exploit any such heterogeneous architecture, we must focus on a best fit implementation and a dynamic kernel configuration for distributed GPUs. Technically, we can parallelize training instance computations among distributed CPUs, and keep our GPU implementation to perform Monte Carlo simulations (continuation values and final price). Nodes with advanced GPUs, offering many CUDA cores can be in charge of performing larger packets of training instances. Overall the strategy of node acquisition depends on many criteria, fixed by user or computed at runtime: energy consumption, performance level, delay of accessibility or time availability. Some of them need to be evaluated through preliminary test phases. We presented static split of our algorithm, but considering Cloud architecture with heterogeneous nodes, a dynamic split at runtime would be suitable to fully exploit Cloud resources according to user constraints. Longer term objective will be to generalize this pattern to the most popular algorithms in option pricing.

978-1-4673-4510-1/12/$31.00 ©2012 IEEE

V. Related work

Lokman [13] has proposed numerous mathematical works and parallelized implementations related to American option pricing, especially with the Longstaff and Schwartz regression method. However the main difficulty of our GPU implementation comes from the many random size simulations involved in the Picazo algorithm. Moreover he does not tackle a dynamic parameterization to cope with heterogeneous GPU devices.

Recent works intend to reduce warp divergence in GPU programs ([14] [15]). We proposed a new implementation to cope with termination divergence. As an alternative, we could have combined a basic implementation (same number of simulations assigned to every thread, FIGURE II 1) with a more generic strategy: in [14] a SIMT micro scheduler is in charge of providing new tasks to threads having exited a loop. However as required in proposed implementations, we are limited in the amount of fast access memory and it can be expensive to manage "task pools" for several small kernel launches. In [15] are introduced software optimizations. One of these targets divergent if-then-else branches in loops: at every iteration it groups same execution paths in a warp, delaying the others. As a complementary improvement it could be adapted to our kernel: in for loop of FIGURE II 2, our classify call depends on if threads reach or not maturity and it requires a lot of computation time for large classification parameters.

VI. Conclusion

Our work focusing on an effective but challenging HPC problem for market finance reveals we must adapt parallelization strategy to SIMT architecture when bringing algorithms designed in a parallel but asynchronous and distributed way to a GPU. We can benefit from advanced libraries at the cost of writing some specific solutions to interact with them, taking care of data types exchanged between both. Because we intend to exploit an hybrid Cloud/GPU architecture possibly featuring heterogeneous GPU devices, it is necessary to have a parametric solution to configure the kernel depending of each device type: we proposed a tool and a dynamic kernel calibration methodology applicable to different GPUs. This tool has been evaluated on the NVIDIA family and an extension to AMD GPUs has been suggested.

A natural way to extend our work is to consider architectures as heterogeneous as possible: distributed infrastructures composed of CPU/GPU nodes acquired from Clouds. This will require in a future work, an elaborated strategy for CPU/GPU nodes acquisition over specific criteria such as energy consumption, performance and acquisition cost, a new distribution/parallelization, algorithm and a possible evolution of our kernel configuration.

Acknowledgment

This work has received the financial support of the Conseil régional Provence-Alpes-Côte d'Azur.

References

[1] Mike Giles, Computational finance on GPUs, Intel Finance Forum, October 8th, 2009

[2] Cognizant Technology Solutions, How Cloud Computing Impacts Trade Finance, November 2011

[3] Tom Groenfeldt, Faster, faster… HPC in financial services, http://www.bankingtech.com/bankingtech/faster-faster-hpc-in-financial-services/20000207828.htm, June 2011

[4] NVIDIA Corporate, NVIDIA Tesla GPUs Used by J.P. Morgan Run Risk Calculations in Minutes, Not Hours, http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&prid=784689&releasejsp=release_157&xhtml=true

[5] J.A. Picazo. American Option Pricing: A Classification-Monte Carlo (CMC) Approach. Monte Carlo and Quasi-Monte Carlo Methods 2000: Proceedings of a Conference Held at Hong Kong Baptist University, Hong Kong SAR, China, November 27-December 1, 2000, 2002

[6] Viet Dung Doan, Grid computing for Monte Carlo based intensive calculations in financial derivative pricing applications, Phd thesis, University of Nice Sophia Antipolis, March 2010

[7] A. Ibanez and F. Zapatero. Monte Carlo Valuation of American Options through Computation of the Optimal Exercise Frontier. Journal of Financial and Quantitative Analysis, vol.39, no.2, 239-273, 2004

[8] OpenCL Best practises guide

[9] Khronos Group, www.khronos.org/opencl/

[10] Machine Learning Group at University of Waikato, www.cs.waikato.ac.nz/ml/weka

[11] JOCL team developer, www.jocl.org

[12] AMD Corporate, http://developer.amd.com/tools/AMDAPPProfiler/html/clkerneloccupancydescriptionpage.html

[13] Lokman A. Abbas-Turki, Parallel Computing for linear, nonlinear and linear inverse problems in finance, Phd thesis, University of Paris-Est, September 2012

[14] Steffen Frey,Guido Reina,Thomas Ertl, SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms, 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 399-406, 2012

[15] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In GPGPU-4, pages 3:1–3:8. ACM, 2011

978-1-4673-4510-1/12/$31.00 ©2012 IEEE