Fast American Basket Option Pricing on a multi-GPU Cluster

Michael Benguigui, Françoise Baude

American Basket Option Pricing on a multi GPU Cluster

Michaël Benguigui[‡], Françoise Baude[*]
INRIA Sophia-Antipolis Méditerranée[‡], CNRS I3S[*], University of Nice Sophia-Antipolis[*]
michael.benguigui@inria.fr, francoise.baude@unice.fr

Keywords: Distributed and parallel computing; Cluster; GPU; OpenCL; Machine learning; Mathematical finance; Option pricing

Abstract

This article presents a multi GPU adaptation of a specific Monte Carlo and classification based method for pricing American basket options, due to Picazo [1]. The first part relates how to combine fine and coarse grained parallelization to price American basket options. A dynamic strategy of kernel calibration is proposed. Doing so, our implementation on a reasonable size (18) GPU cluster achieves the pricing of a high dimensional (40) option in less than one hour against almost 8 as observed for runs we conducted in the past [2], using a 64-core cluster (composed of quad-core AMD Opteron 2356). In order to benefit from different GPU device types, we present a dynamic strategy to load balance GPU calculus which greatly improves the overall pricing time. An analysis of possible bottleneck effects demonstrates that there is a sequential bottleneck due to the training phase that relies upon the AdaBoost classification method, which prevents the implementation to be fully scalable, and so prevents to envision further decreasing pricing time down to handful of minutes. For this we propose to consider using Random Forests classification method: it is naturally dividable over a cluster, and available like AdaBoost as a black box from the popular Weka machine learning library. However our experimental tests will show that its use is costly.

1. INTRODUCTION: GPUS IN FINANCE

Many financial measures require huge resources to be computed in acceptable time. "Acceptable" is related to specific context: Value at Risk may be performed to forecast the maximum loss of a given portfolio at a two weeks horizon whereas computing hedging portfolios is often dedicated to intraday operations. The difficulty not necessarily depends on computation methods but on engaged financial instruments. For instance, a portfolio can be composed of several financial instruments and which can vary from a simple asset to option on several assets. In this paper, we focus on pricing one complex financial instrument: an American option, which for being realistic, is based upon a basket of up to 40 assets. The difficulty to price an American option is to predict an exercise frontier to consider all possible exercises times until the maturity date. Furthermore, model parameters such as discretization, number of simulations, complicate computation time. Our previous work [2] highlights the necessity to target a GPU rather than distributed CPUs to provide the same performance level. By this way we price complex American basket options, in the same order of time than a CPU cluster implementation [3] on a 64-core cluster (quad-core AMD Opteron 2356 with Gigabit Ethernet connections), which is around 8 hours. However a single GPU is limited for such complex problems. Targeting cluster of GPUs is the natural following step to benefit of both aggregated memory of their host CPUs, and high parallelism of SIMT architectures. Consequently, our newest goal has been to reach the symbolic 1 hour or less of computation time for solving such a complex problem, characterized by its non-embarrassingly parallel nature. To this aim, we have been obliged to thoroughly optimize each step of the parallel method as will be detailed.

The paper makes the following contributions. First we propose a two-level CPU/GPU parallelization of the Picazo pricing algorithm. Then we perform a dynamic load balancing strategy to exploit heterogeneous multi GPU clusters. Finally we show how to integrate Random Forests [4] in our pricing engine to make it better scale: we propose a distribution of the classifier training and a GPU based implementation of the classification.

We will describe in section 2 a multi GPU implementation to price such financial instruments through Picazo method. At a coarse-grained level, we will focus on the parallelism orchestration across the cluster nodes. Then we will explain our fast dynamic strategy to calibrate kernel parameters in parallel, and expose our load balancing solution for heterogeneous multi-GPU clusters. Finally at a fine-grained level, we will detail the SIMT oriented implementation. In section 3, we will expose our strategy to tackle the bottleneck effect of the sequential learning phase supported by a boosting (AdaBoost) or a Support Vector Machines (SVM using Sequential Minimal Optimization) method, replacing it by the naturally parallelizable Random Forests method. We are able to divide it over CPU nodes, each node training a small forest through the Weka libray [5]. Doing so, we obtain a fully parallel pricing algorithm. In both sections, tests will highlight advantages/disadvantages of each classification method.

2. A GPU CLUSTER BASED OPTION PRICING ENGINE

Here we describe a Java implementation of the selected pricing method due to Picazo. We use the JOCL [6] and OpenCL [7] libraries to exploit distributed GPUs. Through a dynamic strategy we recognize GPUs over nodes and adapt kernel parameters before load balancing main computation phases. Tests reveal bottleneck effect due to building phases of classifiers and necessity to parallelize them as exposed in section 3.

2.1. Picazo pricing algorithm

High dimensional American basket call/put option is a contract allowing the owner to buy/sell at a specified strike price *K*, a possibly high size (e.g. 40, as in the CAC 40 index) set of underlying assets S^i_t (numbered *i*) at any time *t* until a maturity date *T*. So a call option owner expects the basket of assets price on the market to raise over strike, as in this case and according to the option contract, the owner will have to spend less money to buy these assets, i.e. to exercise the option. There is no analytic solution to price this financial instrument but Monte Carlo (MC) 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 price trajectories of the basket of assets following geometric Brownian motion processes, $\Psi\ (f\ (S_t^{(s)}),\ t)$ as the option payoff, *f* as the arithmetic or geometric mean function, *r* as the risk free rate. European option price *V* at time zero can be estimated, through a number of MC simulations *nbMC*, as follows

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

As opposed to European contracts, American ones offer more flexibility for the exercise: it can be performed at any time until the maturity date, and this over all discrete times. This is reflected in the mathematical definition below

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

Here, the formula $\mathrm{E}[e^{-r(t_{m+1}-t_m)} V(S_{t_{m+1}}, t_{m+1}) \mid S_{t_m}]$ defines the continuation value at time t_m, noted *C* in Figure 1, i.e. the forecasted option price at t_{m+1}. The option owner will keep it, if its forecasted price is over the benefit of immediately exercising it, i.e. the payoff.

Picazo method exposes an efficient way to define continuation or exercise regions, separated by a frontier named exercise boundary, by combining a machine learning technique with MC methods. The algorithm is shown in Figure 1. We note *d* the basket size, δ_i and σ_i respectively the dividends and volatilities of the *i* = 1...*d* underlying assets, *N* the discrete time number.

The key pricing method strategy is to call a specific classifier per discrete time during the *nbMC* simulations of the final pricing phase [phase 2], to decide if current simulation must be stopped or not, i.e. if simulated prices reach or not an exercise region. To achieve this, we need during a previous phase [phase 1], to train each classifier [step 2] over *nb_class* training instances. Each training instance is composed of simulated underlying asset prices and a boolean, depending on if the option payoff is over or not an estimation of the continuation value. Each continuation value requires *nb_cont* MC simulations [step 1]. Consequently there are *nb_cont* MC simulations needed per training instance.

Algorithm Classification and Monte Carlo Algorithm

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] :** **each worker iterates over a subset of *nb_class* points**
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 $\Psi^{(s)}(S_{t_m}, t_m)$. **each worker simulates**
7: **if** $C^{(s)}(S_{t_m}, t_m) \leq \Psi^{(s)}(S_{t_m}, t_m)$ **then** ***nb_cont* trajectories on its GPU**
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** **each worker simulates a subset of *nbMC* trajectories on its GPU**
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.

Figure 1. Picazo pricing method and the two parallelization levels (in rectangles)

2.2. Distribution orchestration for coarse-grained parallelism

Our CPU/GPU parallel version of the Picazo pricing algorithm introduces two levels of parallelism as Figure 2 depicts. The first level follows a coarse-grained parallel master-slave approach. We use the Java ProActive library [8] which offers an abstraction of distribution management by introducing the concept of Active Object. By this way, during the detection phase described in **part I** of Figure 2, whose role is to dynamically detect what are the available computing resources, we deploy as many active objects as cluster nodes and discover the number of residing CPU cores and GPUs per node. In our pricing strategy, more than workers, we require a merger to gather intermediate results. Finally during this initialization phase illustrated in **part II**, we allocate the merger active object on the node with the fewer GPUs, and there will be as many workers active objects as GPUs, each responsible to handle the corresponding GPU kernel execution, which the second level fine-grained SIMT parallelism is. Running multiple workers to exploit GPUs on a single node will not significantly impact performance because workers jobs are GPU intensive.

Part III (as summarized on the corresponding part of the schema on the left of Figure 2) details the orchestration of the training instances computation for each classifier. To estimate a continuation value per training instance, a worker launches *nb_cont* MC simulations on its GPU. The merger recovers all the training instances from workers to train (sequentially) a

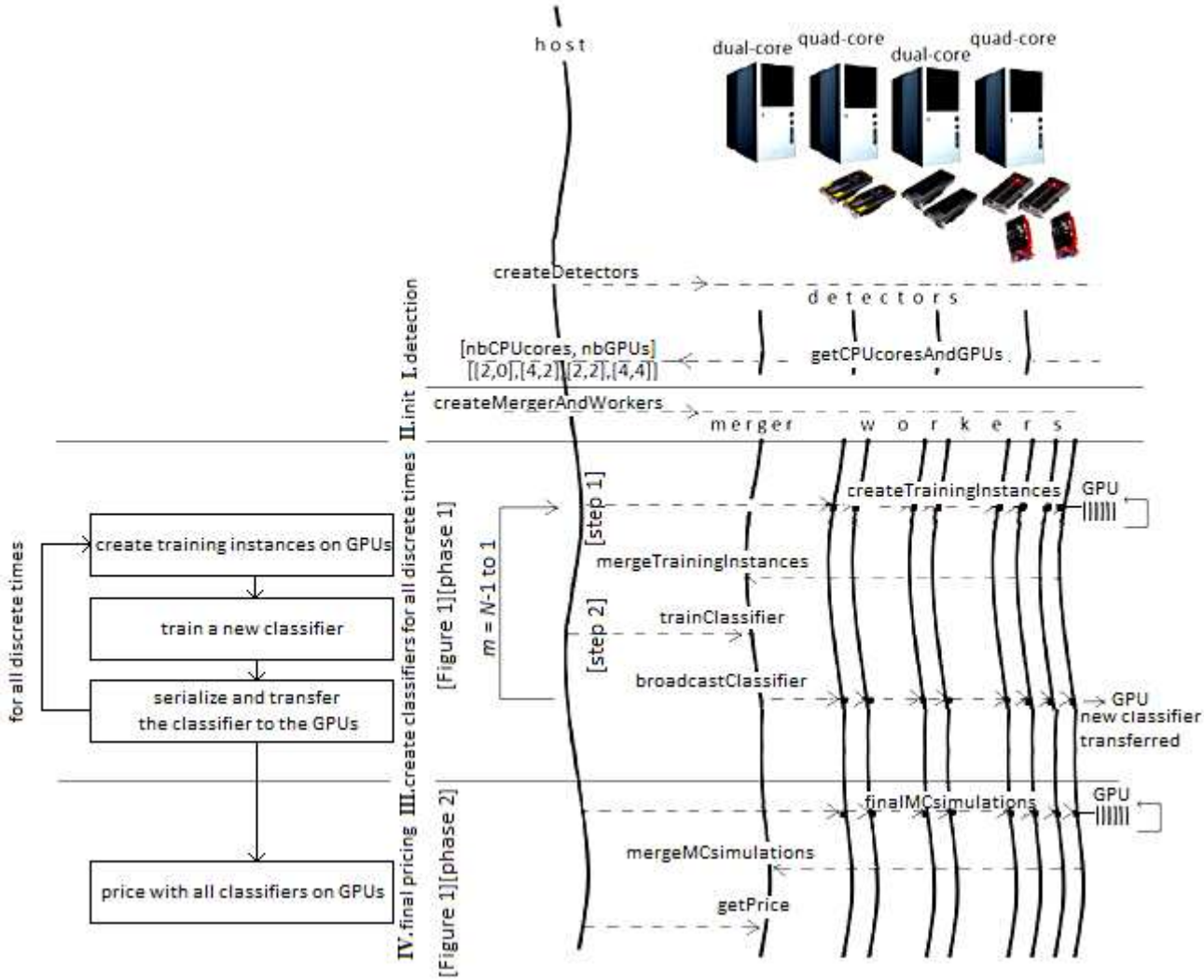

Figure 2. Parallelism orchestration of the Picazo pricing method

new classifier. Notice that this classifier will be used during the MC simulations of the final pricing phase, but also during the MC simulations of the continuation values. Therefore the merger broadcasts the new trained classifier to all workers, at the beginning of each discrete time loop iteration. Once all classifiers are trained (and have already been copied on each GPU by the loop of part III), each worker is distributed a subset of MC simulations to estimate the final price as **part IV** depicts.

2.3. Kernel parameters calibration and load balancing

2.3.1. Dynamic kernel parameters calibration

Targeting GPU programming implies to be ready to cope with a wide variety of GPUs. To ensure high multiprocessor occupancies for each worker, we must calibrate kernel parameters, i.e. work-group size and global size. For this, we provide a Java class which imitates the CUDA occupancy spreadsheet. Before starting the first step of the pricing algorithm, each worker, in charge of one GPU device, computes theoretical multiprocessor occupancies for all possible work-group sizes: from the warp size up to the maximal work-group size allowed, increased by warp size. As required in the spreadsheet, some device specifications are required: each worker detects shared memory amount per multiprocessor, maximal work-group size, generates the program compilation log to parse used registers. Different kernel configurations can describe same multiprocessor occupancies, for instance 4 work-groups of 32 threads against 2 of 64. In such case, our program will keep the one offering more work-groups, to reduce waiting time between them (as each work-group would be given a smaller simulations number to perform). As intermediate calculus to deduce the multiprocessor occupancy, the theoretical active work-group number by multiprocessor is estimated, and will be reused to fix the total threads number to: work-group size multiplied by number of active work-group per multiprocessor multiplied by number of multiprocessors on the device. This strategy allows a fast estimation of kernel parameters for each of the detected GPUs to ensure a high multiprocessor occupancy without launching any preliminary fake pricing calculations.

2.3.2. Load balancing strategy

We have to assign a performance indicator to each GPU, regarding the user pricing parameters. The random length of the trajectories, does not afford to only consider their durations as performance indicators. The idea is to measure the execution time of a "small" kernel, and divide it by the maximal number of time steps processed by a thread. Obviously the kernel is launched with the user parameters but processes short trajectories starting close to the maturity date. By this way, we only need to train one classifier before launching the kernel. There are as many performance indicators $perf_W$ estimated in parallel, as workers W attached to GPUs. Finally, the subset of the *nb_class* training instances and the subset of the *nbMC* simulations, processed by a given worker, are inversely proportional to the performance indicator as follows

$$nb\_class_W = \frac{1/perf_W}{\sum_{ALL\ WORKERS\ P} 1/perf_P} \times nb\_class$$

Figure 3 highlights the impact of our dynamic split strategy over a heterogeneous GPU-based cluster holding three different GPUs. On Grid'5000 [9], each cluster node can directly interact with other cluster nodes, i.e. without having to traverse a cluster front-end node. Thus, virtually all Grid'5000 nodes form a single heterogeneous cluster. Each node of the Grenoble Adonis cluster has 2 Intel Xeon E5520 and 2 NVIDIA Tesla S1070. The Lille Chirloute cluster includes 4 NVIDIA Tesla M2050 and each node has 2 Intel Xeon E5620. The Lyon Orion cluster holds a single NVIDIA Tesla M2075 and each node has 2 Intel Xeon E5-2630. These sites are connected with 10Gbit/s optical fibers. We launch a single worker on each site to exploit 3 different Tesla cards. The merger is executed on a single node from the Adonis cluster.

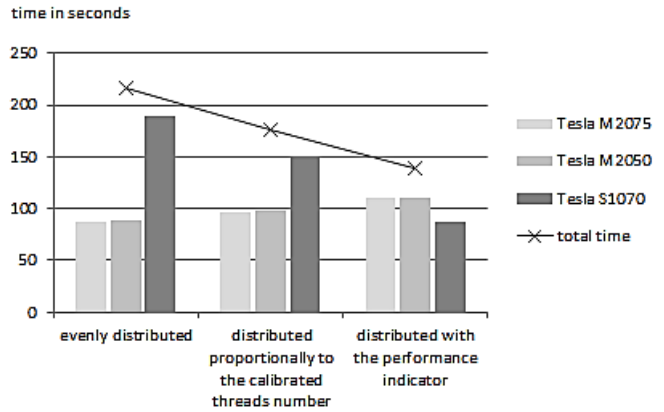

Figure 3. Total durations of training instances creations, and total pricing times, on a cluster of 3 GPUs, with different distribution strategies. AdaBoost classification, with 150 boosting iterations/decision stumps. Geometric average American call option, S^i_0=90, d=7, K=100, N=10, T=1, r=3%, δ_i=5%, σ_i=40%, *nb_class*=5000, *nb_cont*=10^5, *nbMC*=2x10^6

In order to highlight the benefits of our strategy, we decided to compare three methods to spread the *nb_class* training instances creations among GPUs. First we evenly distribute among the GPUs. Then we distribute proportionally to the calibrated threads number (cf. 2.3.1). Finally we use our strategy with the performance indicator. The Tesla S1070 slowdowns the pricing time as illustrate the dark grey bars in the two first strategies (1.5x – 2x slower than with the Tesla M series). The last strategy tackles the bottleneck effect due to the Tesla S1070 as depicts the decreasing solid line: using our load balancing method, we reduce respectively by 36% and 21% the overall pricing time with the first and the second strategy.

2.4. Fine-grained parallelism with OpenCL

Each worker computes a subset of *nb_class* training instances and requires for each to estimate a continuation value through *nb_cont* MC simulations, c.f. Figure 1 line 6. MC simulations are launched through an OpenCL kernel function. There are as many parallel simulations on the GPU as threads iterating to provide the *nb_cont* simulations. Difficulty of pricing American option is the random length of simulations: a classifier can predict the exercise region is reached at any time before the maturity date. Consequently we cannot forecast the required random variables number and we use the GPU based Random Number Generator MWC64X [10] to generate at runtime only required variables. At each discrete time of a single simulation, a thread generates as many uniform random variables as underlying assets, performs the Box Muller transformation to retrieve the Gaussian values, simulates the underlying assets prices, call the specific classifier, and finally computes the actualized payoff, adds it to a variable allocated in a register, and start a new simulation.

This random stopping time leads to some threads finishing earlier their simulations than others. A "warp", for NVIDIA architecture or "wavefront" for AMD, is the smallest quantity of threads that are issued with a SIMT instruction. Because threads of the same warp cannot perform at the same time different instructions, some of them will block at the main loop condition if they perform short simulations (as dictated by the classifier call). These unwanted synchronizations lead to low occupancy of the multiprocessor. That's why we cannot simply iterate over the same fixed number of simulations for all threads when computing the *nb_cont* simulations. Thus, each thread computes after *nbStepsBeforeReduction* time steps and through intermediate reductions (parallel sums), how many MC simulations have been achieved (see further details in [2]). This is repeated by each thread, until the total number of simulations of all threads reaches at least *nb_cont*.

We kept in mind all recommendations of the GPU device programming guide to avoid possible performance losses. In particular, (1) coalesced access allow threads to get asset prices from global memory in few instructions, (2) we employ constant cached memory to store read-only values such as volatilities or dividends, and (3) perform the intermediate

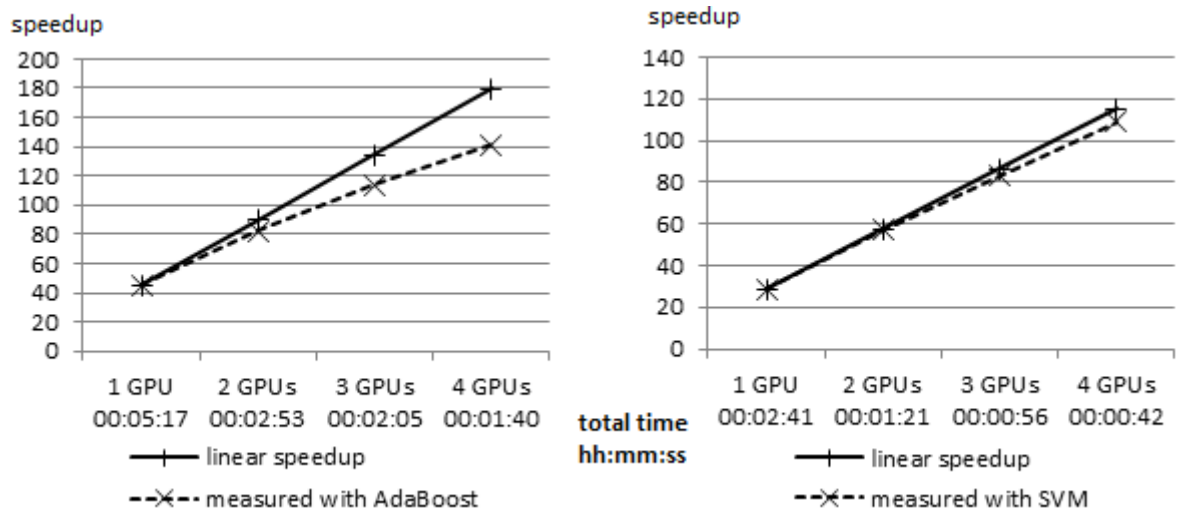

Figure 4. (Left) Speedups of the pricing algorithm using AdaBoost classification, with 150 boosting iterations/decision stumps. (Right) Speedups of the pricing algorithm using SVM SMO, with a linear kernel. Other pricing parameters are the same than in Figure 3.

parallel reductions in shared memory. Specific tests revealed that even a high number of reductions for summing do not impact global execution time.

Classifiers used during Monte Carlo simulations are previously created and trained on the CPU by the merger with the Weka library. Since OpenCL does not allow advanced library call, each worker needs to work with a serialized version of the Weka Classifier object obtained at kernel launches. The two possible classifiers from Weka we experimented with, AdaBoost and SVM, were slightly modified to retrieve all the private members of the Weka object and only cope with basic structures in OpenCL. Then all of them are transferred to the global memory to imitate the Weka classify call on the GPU. At the end, we can afford to imitate the original Weka behavior with basic structures, and store as many classifiers as discrete times in arrays. During a kernel execution, threads work with position indexes to access in parallel different classifiers to predict the stopping times.

2.5. Speedup experiments on a 7-asset American option

Figure 4 depicts the speedups of a 7-asset American option pricing, on the Chirloute cluster. Our implementation of the non-embarrassingly algorithm achieves a speedup of 140 using 4 GPUs, with AdaBoost classification method. Scalability with more GPUs will be discussed in 3.3. Training a linear SVM classifier takes less than 1 second and does not slowdown the total pricing time, as our almost linear curve illustrates (**right**). The counterpart of using a linear kernel with SMO is the underestimation of the option price (-15%), which can be corrected by considering a polynomial kernel, increasing the training duration. The total time of the AdaBoost classifiers trainings, varies from 8% (1 GPU) to 25% (4 GPUs) of the total pricing time: this bottleneck is highlighted on the **left** figure. We need to consider a fully scalable classification method to approach a linear speedup, without impacting price accuracy. AdaBoost and SVM are based upon iterative algorithms during the learning phase, unlike Random Forests whose training phase can be entirely split over the cluster. Our follow-up aim is thus to experiment using this alternative classification method.

3. RANDOM FORESTS INTEGRATION FOR PARALLEL CLASSIFIER TRAINING

We focus here on the integration of Random Forests in our pricing engine. Experimental tests will illustrate the scalability of our implementation, thanks to the parallelization of the training phase. However, this will come at the expense of a high increase of the creation of training instances time as executed by GPU devices.

3.1. Training Random Forests over CPU cluster

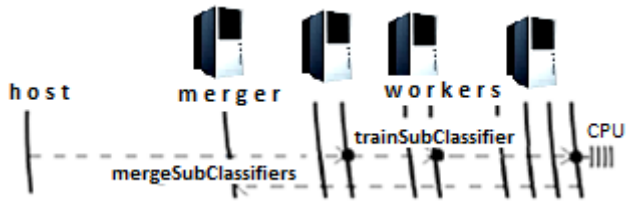

Figure 5. Parallelization of a random forest training. Each subClassifier/small forest is trained over the detected CPU cores through the Weka library

When distributing a random forest training, we decided to preserve the Weka behavior: the idea was to train in parallel small random forests with the same *buildClassifier()* call as it was for a single larger one. The Weka library was slightly modified so that the original random forest and the one

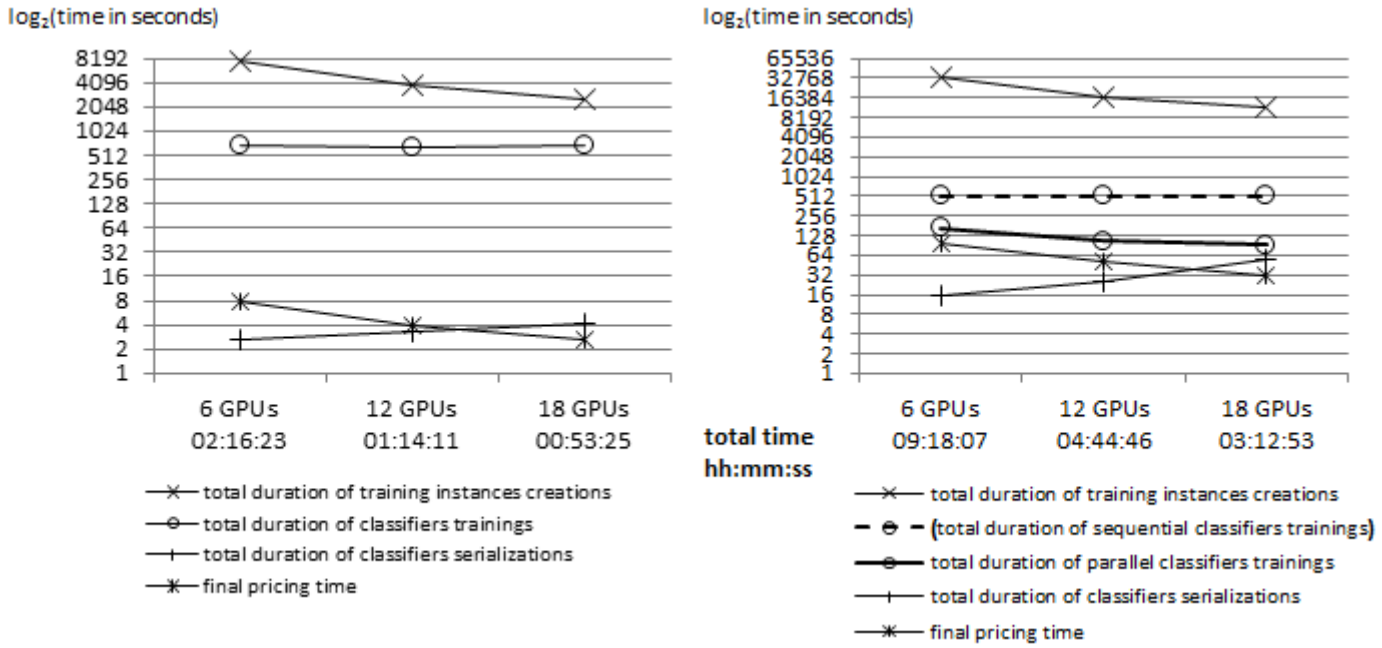

Figure 6. (Left) Comparison of algorithm phases execution times with AdaBoost classifiers over workers numbers. (Right) Comparison of algorithm phases execution times with random forests of 150 unlimited depth trees, over workers numbers. The pricing parameters are the same than previously and total times correspond to the situation where training of classifiers is distributed

obtained after merging all smaller forests built by workers, provide strictly identical classification measures. By this way, we can train in parallel subsets of a forest over cluster nodes (Figure 5). As complementary optimization, we decided to exploit the last Weka library version affording parallelization over CPU cores. For this, only one active object worker per node is in charge of a sub classifier to take advantage of all CPU cores for the training.

We set the Weka parallelization degree of each node with the number of detected CPU cores. A simple load balancing mechanism affords each worker *W* to build a specific subset $nbTrees_W$ of the total number $nbTrees_{CLASSIFIER}$ of a random forest, such as

$$nbTrees_W = \frac{nbCPUcores_W}{\sum_{ALL\ CPUs\ P} nbCPUcores_P} \times nbTrees_{CLASSIFIER}$$

For the following tests (Figure 6), we will disable this optimization, in order to highlight the benefit of the training distribution over cluster nodes. Once all workers have finished, the merger retrieves all sub classifiers, merges them and broadcasts the trained global random forest to all workers that will use them, as explained in the following subsection.

3.2. Parallel Random Forests classifications on GPU Units

As for AdaBoost or SVM, a random forest per discrete time must be serialized by the worker, and transferred to the GPU global memory, in order to predict the exercise boundary at this time, during the simulations, c.f. Figure 1 line 6 and 15. The difficulty comes from the storage of the trees that are indeed incomplete. Only an experimental solution is provided by the JOCL team, to transfer tree structures to the device, so we had to imagine one solution that fits our needs. To cope with sparse tree storage, we work with compressed arrays representation. Once workers are broadcasted the merged global random forest, they parse all trees, retrieve and queue node information in specific arrays for the compression. Indeed considering all trees, there is an array for split values, another one for attribute indexes. We store indexes of tree roots in a dedicated array. Finally, we work with a left children indexes array and a right children indexes array, to imitate tree parsing when classifying instances in OpenCL. As for AdaBoost and SVM, we queue all the classifier representations in the same specific arrays to be accessed for each discrete time, complicating indexes management.

3.3. Scalability experiments on a 40-asset American option

Figure 6 depicts execution times of parts III and IV (Figure 2) in case of high dimensional American option, on the homogeneous Adonis cluster. Parts I and II are not specified here due to their small execution times, and possibility to reuse the resulting active objects deployment for multiple program runs. With the Adonis classification (**left**), the option price is around 0.64108 ± 0.0015 (95%CI), which is in line with the reference price according to [3], and so validates the correctness of the program. We performed more tests to ensure results and execution times presented are representative of our pricing executions. Times of training instances creations and final pricing phases include calculus and broadcast/merge operations from/to the merger. We fall below 1 hour when performing tests over 18 GPUs (no high-end). All performed tests have revealed linear dependence of workers numbers with the computation part of each phase, but have also shown managing more workers complicates broadcast/merge operations and slowdowns these operations respective overall time. More annoying, because the merger sequentially trains each classifier through the Weka library and does not solicit workers, the implementation is not scalable: when increasing workers number, the training instances computation time decreases, and consequently tends to vanish in comparison to the constant (because sequential) time of the classifiers training (~650s).

Using Random Forests (**right**), the option price of a single run is around 0.63651 ± 0.0016 (95%CI) which is in line with the expected value. Working with such random forests parameters (as of 150 trees) provides the same order of confidence interval than AdaBoost tests. The training instances creations (~3h10min with 18 GPUs) require more time than with AdaBoost (~42min with 18 GPUs) due to the cost of forests classifications. Indeed, to classify an instance, a GPU thread will take more time to parse the 150 unlimited depth trees, rather than the 150 one-level decision trees of the AdaBoost classifier. Conversely, as describe the dotted and solid lines with circles, we take advantage of the distributed CPUs during the classifiers trainings, allowing the algorithm better scales.

4. RELATED WORK

Regarding the fine tuning for GPU configuration, Grauer and Cavazos present an auto-tuning implementation in [11] to produce the configuration that minimizes local memory accesses against registers and shared memory. Since they play with data partition sizes via changing the maximum occupancy, the strategy allows finest kernel parameters calibration for bandwidth-bound applications but is less generic than ours. Raphael Y. de Camargo [12] describes a load distribution algorithm for heterogeneous GPU cluster to reduce the total execution time of his neuronal network simulator. To estimate each quantity of data input assigned to each GPU, he formalizes the problem to a linear system of equations. Some variables in the system represent the execution time functions of each kernel on each GPU over input sizes. This requires each kernel to be executed a few times on each GPU, with different input sizes to get the interpolation function. This can spend a lot of time and become inconvenient, in case of several types of GPUs and compute-intensive kernels. On the contrary, our parallel and dynamic load balancing strategy, with small kernels launches, allows a fast comparison of the performance degrees of each GPU for a given kernel. Tse [13] proposes a dynamic scheduling strategy for Monte Carlo simulations, targeting multi-accelerator heterogeneous clusters. Each accelerator requests a MC distributor, a subset of the remaining MC simulations to perform; the distributor applies a distribution strategy through which subset size allocated to each requesting accelerator increases (either linearly or exponentially given the tested distribution strategy) at each time. The faster accelerator will logically process more simulations than the slower after a period of time. The non-embarrassingly parallel Picazo algorithm, involves multiple small kernel launches, for each discrete time sequentially processed, and is not suitable for a runtime scheduler. Thanks to our adequate initial load-balancing strategy, the amount of work given to each accelerator is precisely known at the beginning of the "for all discrete time" loop (Part III figure 2). Regarding the final pricing phase of an American option (Part IV figure 2) which is embarrassingly parallel, we also apply the method of 2.3.2 to decide the subset size of MC simulations each accelerator is allocated at once. In the experiments we run, this phase was quickly executed because of the chosen, still realistic, pricing parameters. Be these parameters much higher, then it could be worth experimenting the dynamic load distribution of [13], the same way they apply it for pricing an Asian option.

In [14] is presented CudaRF, a CUDA-based implementation of Random Forests. During the training phase, each thread constructs a tree of the forest. It could be used within our ProActive-based distributed training phase so that huge random forests could benefit of a dual-level of parallelism offered at both worker and GPU sides. However, having a GPU thread handles one single tree of the forest during the classification phase, is not suited to our algorithm. We cannot afford to exploit at a specific time the entire device for a single instance, as our implementation exploits SIMT architecture to call simultaneously possibly different classifiers, depending on the discrete time reached by each thread.

5. CONCLUSION

Our works propose a multi GPU based implementation of Picazo method to price high dimensional American options, allowing pricing time to fall below 1 hour on 18 GPUs, for a 40-asset option (c.f. 3.3). This outperforms the CPU cluster implementation, which spends almost 8 hours on a 64-core cluster. We reach a speedup ratio of 140 on 4 GPUs with a less complex American basket option (c.f. 2.5). To fully exploit the dual-level of parallelism of such architecture, we distribute the training instances computation over the cluster nodes and solicit the SIMT architecture of each detected device to parallelize all the Monte Carlo simulations of the

algorithm. Our fast parallel strategy to estimate kernel parameters of devices can be adapted to a wide range of GPUs to target any cluster. We presented a dynamic load balancing strategy reducing by 36% the parallel pricing time of a 7-asset option. The integration of Random Forests, tackles the sequential bottleneck effect due to the classifiers trainings by parallelizing them, but slowdown the training instances creations due the expensive classification. Obviously, a challenging alternative would be to come up with a faster parallel classification method with a scalable learning phase, such as [15]. Also working with more GPUs (100+) than in our experiments, would further decrease these computation operations but increase broadcast/merge operations, impacting the overall pricing time. Thus, to face this only remaining bottleneck effect, we could implement one of the broadcasting schemes detailed in [16] to parallelize the propagation of data between adjacent nodes. Furthermore, we could parallelize merge operations along a parallel tree reduction. Next step is to prove in a practical way that pricing a complex option can now be achieved within minutes; however this would require getting access to a GPU cluster hosting several hundreds of – probably heterogeneous -- accelerators, a rare resource type. Consequently, our work also militates in favor of research for much more efficient parallel classification methods.

It would be exiting to take advantage of high end CPUs (Xeon Phi) if available on the cluster, to perform part of the Monte Carlo simulations. By relying on OpenCL in our pricing engine, it already abstracts the hardware architecture. The only point to consider in order to take advantage of such hybrid hardware environment is to extend our dynamic calibration and load balancing strategy. A natural exploitation of our work is to evaluate a portfolio of such complex assets, which is an ongoing task.

Acknowledgment

This work has received the financial support of the Conseil régional Provence-Alpes-Côte d'Azur. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies.

References

[1] 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

[2] Michael Benguigui, Françoise Baude, Towards parallel and distributed computing on GPU for American basket option pricing, in the 2012 International Workshop on GPU Computing in Cloud in conjunction with 4th IEEE international conference on Cloud Computing Technology and Science, 2012

[3] 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
http://www-sop.inria.fr/oasis/personnel/Viet_Dung.Doan/thesis/

[4] L. Breiman, Random Forests, Statistics Department of California Berkeley, January 2001

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

[6] JOCL, http://www.jocl.org/

[7] Khronos Group, http://www.khronos.org/opencl/

[8] http://proactive.inria.fr/

[9] https://www.grid5000.fr/

[10] David Thomas, http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html

[11] Scott Grauer-Gray and John Cavazos, Optimizing and Auto-tuning Belief Propagation on the GPU, In 23rd International Workshop in Languages and Compilers for Parallel Computing (LCPC), 2010

[12] Raphael Y. de Camargo, A load distribution algorithm based on profiling for heterogeneous GPU clusters, Third Workshop on Applications for Multi-Core Architecture, 2012

[13] Anson H.T. Tse, David B. Thomas, K.H. Tsoi, Wayne Luk, Dynamic Scheduling Monte-Carlo Framework for Multi-Accelerator Heterogeneous Clusters, in Proceedings of IEEE Symposium on Field-Programmable Technology (FPT), 2010

[14] Håkan Grahn, Niklas Lavesson, Mikael Hellborg Lapajne, and Daniel Slat, "CudaRF": A CUDA-based Implementation of Random Forests, Proc. Ninth ACS/IEEE International Conference on Computer Systems and Applications, IEEE press

[15] Munther Abualkibash, Ahmed ElSayed, Ausif Mahmood, Highly Scalable, Parallel and Distributed AdaBoost Algorithm using Light Weight Threads and Web Services on a Network of Multi-Core Machines, International Journal of Distributed & Parallel Systems, Vol. 4 Issue 3, p29, May2013

[16] John Matienzo, Natalie Enright Jerger, Performance Analysis of Broadcasting Algorithms on the Intel Single-Chip Cloud Computer, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013