Valorisation d'options américaines et Value At Risk de portefeuille sur cluster de GPUs/CPUs hétérogène

Michaël Benguigui

UNIVERSITE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour l'obtention du grade de
Docteur en Sciences

de l'Université Nice-Sophia Antipolis
Mention : Informatique

présentée et soutenue par
Michaël BENGUIGUI

Valorisation d'options américaines et Value At Risk de portefeuille sur cluster de GPUs/CPUs hétérogène

Thèse dirigée par *Françoise BAUDE*

soutenue le *27 août 2015*

Jury

Rapporteurs

KRAJECKI Michaël	Université de Reims Champagne-Ardenne
COUTURIER Raphaël	Université de Bourgogne Franche-Comté
HAINS Gaétan	Huawei R&D

Examinateurs

BOSSY Mireille	INRIA
FILLATRE Lionel	UNS

Directrice de thèse

BAUDE Françoise	UNS

Valorisation d'options américaines et Value At Risk de portefeuille sur cluster de GPUs/CPUs hétérogène

Résumé

Le travail de recherche décrit dans cette thèse a pour objectif d'accélérer le temps de calcul pour valoriser des instruments financiers complexes, tels des options américaines sur panier de taille réaliste (par exemple de 40 sous-jacents), en tirant partie de la puissance de calcul parallèle qu'offrent les accélérateurs graphiques (Graphics Processing Units). Dans ce but, nous partons d'un travail précédent, qui avait distribué l'algorithme de valorisation de J.Picazo, basé sur des simulations de Monte Carlo et l'apprentissage automatique. Nous en proposons une adaptation pour GPU, nous permettant de diviser par 2 le temps de calcul de cette précédente version distribuée sur un cluster de 64 cœurs CPU, expérimentée pour valoriser une option américaine sur 40 actifs. Cependant, le pricing de cette option de taille réaliste nécessite quelques heures de calcul. Nous étendons donc ce premier résultat dans le but de cibler un cluster de calculateurs, hétérogènes, mixant GPUs et CPUs, via OpenCL. Ainsi, nous accélérons fortement le temps de valorisation, même si les entrainements des différentes méthodes de classification expérimentées (AdaBoost, SVM) sont centralisés et constituent donc un point de blocage. Pour y remédier, nous évaluons alors l'utilisation d'une méthode de classification distribuée, basée sur l'utilisation de forêts aléatoires, rendant ainsi notre approche extensible. La dernière partie réutilise ces deux contributions dans le cas de calcul de la *Value at Ris*k d'un portefeuille d'options, sur cluster hybride hétérogène.

Mots-Clés: parallélisme – distribution - GPGPU – OpenCL - cluster hybride hétérogène de GPUs/CPUs - mathématiques financières - calcul de risque - option américaine – Monte Carlo - apprentissage automatique

American option pricing and computation of the portfolio Value at risk on heterogeneous GPU-CPU cluster

Abstract

The research work described in this thesis aims at speeding up the pricing of complex financial instruments, like an American option on a realistic size basket of assets (e.g. 40) by leveraging the parallel processing power of Graphics Processing Units. To this aim, we start from a previous research work that distributed the pricing algorithm based on Monte Carlo simulation and machine learning proposed by J. Picazo. We propose an adaptation of this distributed algorithm to take advantage of a single GPU. This allows us to get performances using one single GPU comparable to those measured using a 64 cores cluster for pricing a 40-assets basket American option. Still, on this realistic-size option, the pricing requires a handful of hours. Then we extend this first contribution in order to tackle a cluster of heterogeneous devices, both GPUs and CPUs programmed in OpenCL, at once. Doing this, we are able to drastically accelerate the option pricing time, even if the various classification methods we experiment with (AdaBoost, SVM) constitute a performance

bottleneck. So, we consider instead an alternate, distributable approach, based upon Random Forests which allow our approach to become more scalable. The last part reuses these two contributions to tackle the *Value at Risk* evaluation of a complete portfolio of financial instruments, on a heterogeneous cluster of GPUs and CPUs.

Keywords: parallel computing – distributed computing – GPGPU – OpenCL - hybrid GPU-CPU cluster – financial mathematics - risk – American option – Monte Carlo – machine learning

Table des matières

Table des illustrations et des algorithmes

Remerciements

A ma compagne Sylvie pour son soutien continuel
A toute ma famille
A ma directrice de thèse Françoise Baude pour son encadrement et ses nombreux conseils
Un grand merci aux rapporteurs et membres du jury pour le temps accordé à l'évaluation de mon travail
A Bastien Sauvan et Fabrice Huet si souvent sollicités
A l'équipe de recherche SCALE ainsi qu'à l'entreprise ActiveEon
Aux membres de Grid'5000
Au Conseil régional Provence-Alpes-Côte d'Azur
A tous ceux qui me sont chers

Chapitre I. Introduction

1. Contexte

Un nombre grandissant de domaines scientifiques bénéficient du calcul haute performance (HPC), notamment en recourant aux simulations de Monte Carlo (MC) qui s'illustrent dans la résolution de problèmes complexes. En physique nucléaire, HPC et méthode de Monte Carlo sont combinés pour simuler les interactions de radiation. En finance de marché, le HPC devient incontournable dans l'arbitrage ou la couverture de risque. En 2009, 10% des supercalculateurs du top 500 étaient dédiés à la finance [1].

Les accords de Bale imposent aux établissements financiers, un niveau minimum de capitaux propres, pour faire face aux variations brusques du marché. Selon le directeur financier de Goldman Sachs de l'époque, les variations de marché constatées pendant la crise de 2007, avaient moins d'une chance sur 1 million de survenir. L'évaluation du risque de marché repose principalement sur l'estimation de la Value at Risk (VaR), pouvant s'avérer couteuse en calculs, si les instruments financiers sont nombreux et complexes. La Monte Carlo VaR, basée sur des simulations de MC est parfaitement adaptée aux options américaines de grande dimension.

Certaines optimisations, telles que la réécriture des algorithmes, ou la vectorisation du code pour exploiter l'architecture vectorielle d'un processeur, n'est dans certains cas pas suffisant pour réduire considérablement les temps d'exécution, d'où l'intérêt de cibler plusieurs unités de calcul en parallèle. Même si le principe du cloud computing permet de déléguer le calcul intensif à des établissements financiers plus importants[1], afin entre autre de bénéficier de nouvelles architecture de calcul [2], il n'en demeure pas moins que solliciter des clusters de CPU a un coût en matériel et en consommation électrique que certains qualifient d'exorbitant. C'est le cas d'Aon Benfield, un leader mondial des compagnies d'assurance, dont le cluster de CPUs dédié à la valorisation d'obligations a coûté 4M de dollars, ainsi que 1,2M de dollar par an d'électricité [3]. Comparativement, un moteur de calcul exploitant des GPUs (Graphics Processing Unit) couterait 144k dollars pour une consommation annuelle électrique de 31k dollars. La section Equity Derivatives de J.P. Morgan quant à elle effectue ses calculs de risque sur une infrastructure hybride constituée de CPUs et GPUs, améliorant ainsi de 40 fois le temps de calcul comparé à un cluster de CPUs, pour une consommation électrique équivalente [4]. Exploiter des GPUs et éventuellement en soutien des CPUs, devient

[1] Il est dans la culture du monde de la finance de détenir en privé les infrastructures informatiques, sur lesquelles transitent des données sensibles. Ainsi, le cloud computing se réduit au modèle de cloud privé, au sein de l'établissement financier ou de ses partenaires.

incontournable dans le calcul intensif en finance, et constitue donc l'approche que nous suivons dans cette thèse.

Nombreuses sont les mesures financières requérant d'importantes ressources pour être estimées en temps raisonnable. Ce temps diffère selon le contexte. La Value At Risk d'un portefeuille peut être calculée pour deux semaines alors que l'estimation d'un portefeuille de couverture est souvent utilisée pour des opérations journalières. La difficulté ne vient pas nécessairement des méthodes de calcul, mais des instruments mis en cause. Un portefeuille peut-être de taille conséquente, et composé de simples actions comme d'options américaines de grande dimension. La difficulté majeure dans la valorisation de l'option américaine est d'estimer sa frontière d'exercice. De plus, les paramètres du modèle comme la finesse de la discrétisation ou encore le nombre de simulations de MC, complexifient les calculs et rallongent leurs temps.

2. Problématique et contributions

La problématique d'exploiter un ensemble de nœuds en parallèle pour résoudre des problèmes complexes non trivialement parallélisables n'est pas nouvelle.

Dans le cadre qui nous intéresse ici, autant effectuer le pricing d'une option européenne qui nécessite simplement de lancer un nombre important de simulations de Monte Carlo indépendantes est aisé à paralléliser, autant le cas d'une option américaine est plus complexe et nécessite de sélectionner une méthode de pricing appropriée. Le travail de thèse de Viet Dung Doan, soutenu en 2010 [5], a permis d'identifier l'intérêt de la méthode de pricing d'option américaine proposée par J. Picazo, dans un objectif de parallélisation. Cette méthode, bien que procédant par itérations sur chaque pas de temps, permet d'introduire du parallélisme au sein de chaque itération. Son travail a ainsi démontré expérimentalement qu'il était possible d'utiliser un cluster de CPUs, selon un pattern maitre-esclave traditionnel, et ainsi réussir à valoriser en quelques heures une option américaine de grande taille (panier d'actifs de taille 40) sur 64 cœurs, une telle option pouvant potentiellement requérir plusieurs jours de calcul si valorisée sur un seul CPU!

Avec l'arrivée massive d'accélérateurs ou cartes graphiques sur tout ordinateur du marché, même ceux d'entrée de gamme, le potentiel de parallélisme est à la portée de toute application. Encore faut-il réussir à concevoir l'algorithme parallèle, puisque programmer un GPU nécessite de se plier à un modèle

parallèle particulier, celui-ci étant connu sous l'appellation SIMT (Single Instruction Multiple Thread). Ainsi, depuis quelques années, nombreux sont les travaux qui ont permis de complètement réadapter des algorithmes distribués ou parallèles à gros grain (MIMD ou MPMD -- Multiple Process Multiple Data) en algorithmes parallèles à grain fin (SIMT). Cette tendance à exploiter des GPUs pour effectuer des calculs est connue sous le terme GP-GPU "General Purpose Graphical Processing Unit".

Notre **première problématique** sera donc de proposer une nouvelle méthode de parallélisation de la méthode de Picazo nécessairement différente de celle que le travail de thèse de Viet Dung Doan avait proposé, afin de réussir à exploiter pleinement le potentiel de parallélisme offert par un GPU. Cette problématique est non triviale car même en se basant sur des simulations de MC indépendantes, la méthode elle-même ne génère pas naturellement la même quantité de travail pour chaque thread du GPU. Ceci nous amènera à une solution, publiée dans [6] où le temps de pricing d'une option américaine non triviale est encore réduit de moitié par rapport à l'utilisation d'un cluster de 64 cœurs [5].

A lui seul, un GPU peut donc potentiellement résoudre aussi rapidement un problème distribué sur un cluster réaliste, c.-à-d. de plusieurs dizaines de cœurs CPU. Néanmoins, et plus récemment dans un but de calculer toujours plus vite, plusieurs travaux s'attèlent à évaluer l'utilisation combinée de plusieurs GPUs, réunis en clusters, et pourquoi pas à profiter du parallélisme natif des multi-cœurs CPUs présents évidemment dans le cluster. Une autre motivation, et non des moindres, est de pallier à la limitation en mémoire d'un GPU, qui empêche alors de résoudre des problèmes plus gros. Mais, cibler un cluster de nœuds foncièrement hétérogènes puisque soit CPU soit GPU, doit naturellement avoir comme ambition de pallier à leur hétérogénéité en termes de puissance de calcul et mémoire disponible.

Notre **seconde problématique** sera donc de voir comment ré introduire un niveau de parallélisme de plus gros grain de type MIMD dans le même esprit qu'en [5], mais en orchestrant plusieurs CPUs et GPUs exploitant eux-mêmes un parallélisme à grain fin hétérogène, et offrant de ce fait une mémoire agrégée conséquente. Le but étant aussi de franchir le seuil symbolique de l'heure dans le temps nécessaire pour valoriser une option américaine complexe avec la méthode de Picazo. Une des difficultés majeures étant de limiter l'impact du goulot d'étranglement séquentiel de cette méthode. Ceci nous amènera à une solution publiée dans [7], et davantage élaborée dans cette thèse en considérant l'hétérogénéité du cluster. Cette solution nous permet d'atteindre

un temps de calcul proche de l'heure avec seulement un cluster hybride de 18 nœuds, grâce entre autre à une stratégie de répartition des tâches appropriée.

Comme évoqué plus haut dans cette introduction, une option est rarement seule, mais fait partie d'un portefeuille d'instruments financiers. D'une part la valorisation du portefeuille est nécessaire, mais plus généralement, l'institution gérant un tel portefeuille doit assurer une couverture de risque et donc régulièrement évaluer la VaR associée. Se pose donc la **troisième problématique** abordée dans notre travail consistant à effectuer la valorisation unique ou multiple (répétée) d'un portefeuille sur un cluster hybride. Et plus particulièrement, comment répartir sur le cluster les valorisations des différentes options, européennes ou américaines, présentes dans le portefeuille.

En bref, notre travail démontre comment profiter des sources hybrides de parallélisme (MIMD, SIMT, sur clusters de CPUs et GPUs) afin d'accélérer théoriquement sans limites, le pricing d'option Américaine d'aussi grande dimension que voulu. La clé est avant tout de sélectionner la méthode de pricing adéquate. Comme en fera état le Chapitre III « état de l'art », celle proposée par J. Picazo (et qui fut la base de la thèse de Viet Dung Doan dont la nôtre est dans la continuité) présente toutes les potentialités requises pour être parallélisée, y compris dans le cas d'une option de grande dimension. La méthode de Picazo [8] repose sur les simulations de Monte Carlo, et l'apprentissage automatique (*machine learning*) qui couvre de nombreux domaines en science : traitement d'image, simulation de particules,... Dans les prochains chapitres, on explicitera son adaptation sur GPU nécessitant une approche de programmation SIMT, et aussi comment y ajouter un niveau de parallélisme supplémentaire de type MIMD, afin d'exploiter un cluster d'accélérateurs et d'unités de calcul plus traditionnelles tels des CPUs multi-cœurs.

3. Organisation du manuscrit

Le Chapitre II reprend quelques fondamentaux nécessaires à la compréhension des chapitres qui le suivent: méthode de MC, option américaine, machine learning, paradigmes MIMD et SIMT.

Le Chapitre III dresse un panorama des méthodes de pricing d'option, et plus particulièrement d'option Américaine. Un focus est fait sur les approches parallèles, et en particulier celles plus récentes sollicitant des GPUs.

En Chapitre IV nous présentons une version GPU de l'algorithme de Picazo, tout en permettant son extension pour un cluster de GPUs. Nous

expliquons les choix d'implémentation que nous soutiendrons par de nombreux tests de performance.

Nous détaillons dans le Chapitre V une implémentation pour cluster hybride et hétérogène, de l'algorithme de pricing d'option américaine, ainsi que notre stratégie de *load balancing*, que nous mettrons en évidence aux travers de tests comparatifs.

Le Chapitre VI étend le problème au calcul de la MC VaR. Nous proposons une série d'optimisations pour réduire le temps de calcul de ce problème complexe si bien connu en finance.

Nous profitons du Chapitre VII pour détailler l'architecture de notre application et certaines spécificités plus techniques, avant de conclure en Chapitre VIII.

Chapitre II. Fondamentaux

1. Aspects mathématiques

1.1. La méthode de Monte Carlo dans notre cadre

Comme nous le détaillerons dans le chapitre suivant, le pricing d'option américaine sur panier d'actions n'offre pas de solution analytique, et certaines méthodes numériques comme celles des différences finies, ne sont pas adaptées pour obtenir en temps acceptable un résultat précis. La méthode de Monte Carlo permet d'y remédier par une approche simplifiée.

Pour le cas simple d'une option européenne, le principe est de générer n trajectoires de rendements (payoffs) afin d'en déduire sa moyenne empirique, proche de son espérance pour n grand. En effet, la Loi forte des Grands Nombres stipule [9]

> *Théorème 1* *Loi forte des Grands Nombres. Soit $(X_i)_{i\in\mathbb{N}}$ des variables aléatoires indépendantes identiquement distribuées et intégrables. Alors presque sûrement et dans L^1*
>
> $$\widetilde{X_n} = \frac{1}{n}\sum_{i=1}^{n} X_i \xrightarrow{n\to+\infty} \mathbb{E}\,X$$

Le biais de l'approximation dépend du nombre de tirages. Pour calculer l'intervalle de confiance, le théorème de la Limite Centrale stipule

> *Théorème 2* *Limite Centrale. Soit (X_n) une suite de variables aléatoires indépendantes identiquement distribuées de carrés intégrables, d'espérance commune μ et de variance commune σ^2. Alors*
>
> $$Z = \frac{\widetilde{X_n} - \mu}{\sqrt{\frac{\sigma^2}{n}}} \to N(0,1)$$

De plus, les tables de la loi normale nous permettent de trouver un unique z pour un α donné tel que

$$\mathbb{P}\{|Z| < z\} = 1 - \alpha$$

Calculer l'intervalle de confiance revient alors à chercher t tel que

$$\begin{aligned}\mathbb{P}\left\{\mathbb{E}X \in \left[\widetilde{X_n} - t; \widetilde{X_n} + t\right]\right\} &= \mathbb{P}\left\{\left|\widetilde{X_n} - \mu\right| < t\right\} \\ &= \mathbb{P}\left\{\frac{|\widetilde{X_n} - \mu|}{\sqrt{\sigma^2/n}} < \frac{t}{\sqrt{\sigma^2/n}}\right\} \\ &= \mathbb{P}\left\{|Z| < \frac{t}{\sqrt{\sigma^2/n}}\right\}\end{aligned}$$

$$=1-\alpha$$

En posant $z=\frac{t}{\sqrt{\sigma^2/n}}$, on obtient $t=z\sqrt{\sigma^2/n}$.

De par le caractère non-linéaire du payoff de l'option américaine, la VaR d'un portefeuille comportant ce type d'instrument est estimée aussi par simulations de MC. La VaR représente la perte maximale x d'un portefeuille à un horizon donné t, selon un seuil de confiance α. Ainsi, une VaR à 95% sur 1 an d'1M d'euros, signifie qu'il y a 95% de chances que la perte maximale du portefeuille dans 1 an soit d'1M. On calcule la MC VaR ainsi

$$\mathbb{P}(L_t>x)=\frac{1}{nb\_VaR}\sum_{i=1}^{nb\_VaR}\mathbb{1}_{L_t^i>x}=1-\alpha$$

En notant V_t^i une valeur de portefeuille simulée en t, on calcule une perte entre t et t_0 ainsi

$$L_t^i=V_t^i-V_{t_0}$$

La méthode de Monte Carlo requiert un générateur de nombres pseudo-aléatoires (MWC, LCG, XORShift,..) contrairement à la méthode de quasi-Monte Carlo qui utilise une suite à discrépance faible (Halton, Sobol,.). Cette dernière présente l'avantage de réduire la variance et le biais du prix de l'option dans certains cas [10]. Cependant, l'estimation d'une option de dimension élevée par la méthode de quasi-Monte Carlo, ne permet pas nécessairement de converger plus rapidement, comme les travaux de Bratley et al. [11] le soulignent pour une option de dimension $d>12$. Finalement, la principale difficulté liée à la méthode de Monte Carlo, réside dans son temps de calcul nécessaire pour produire un intervalle de confiance réduit. Ce qui constitue l'enjeu même de nos travaux nous orientant vers une infrastructure parallèle et distribuée. Enfin, pour générer des variables aléatoires normalement distribuées à partir de variables aléatoires uniformément distribuées, notées

$$u_1,u_2\rightarrow U(0,1)$$
$$z_1,z_2\rightarrow N(0,1)$$

nous employons la transformation de Box-Muller opérant ainsi

$$\begin{cases}z_1=\sqrt{-2\ln(u_1)}\cos(2\pi u_2)\\ z_2=\sqrt{-2\ln(u_1)}\sin(2\pi u_2)\end{cases}$$

1.2. Définition de l'option américaine sur panier

Une option américaine sur panier d'actions, plus communément appelée call/put, est un contrat permettant d'acheter/vendre à un prix d'exercice K un ensemble d'actions, à n'importe quel instant jusqu'à sa maturité T. On note les prix en t des $i = 1 \dots d$ actifs du panier S_t^i. Dans le cas d'un put américain, en notant f la moyenne arithmétique

$$f(S) = \frac{1}{d}\sum_{i=1}^{d} S^i$$

ou géométrique

$$f(S) = \left(\prod_{i=1}^{d} S^i\right)^{\frac{1}{d}}$$

son détenteur qui l'exerce à l'instant t bénéficie d'un payoff Ψ

$$\Psi = \max\big(0, K - f(S_t)\big)$$

Soit $S_t^{(S)}$ les trajectoires indépendantes du panier d'actifs, suivant un mouvement brownien géométrique, sur lequel le lemme d'Ito s'applique

$$W_t \to N(0, \mathrm{t})$$
$$S_t = S_0 e^{\left[\left(\rho - \frac{\sigma^2}{2}\right)t + \sigma W_t\right]}$$

On note ρ le dividende et σ la volatilité. Le prix V en t_0,d'une option européenne, estimé via $nbMC$ simulations de MC vaut

$$V\big(S_{t_0}, t_0\big) \approx \frac{1}{nbMC}\sum_{s=1}^{nbMC} e^{-rt}\Psi\left(f\left(S_t^{(s)}\right), t \in [0, T]\right)$$

On note r le taux sans risque. Contrairement à l'option européenne qui ne peut s'exercer qu'à maturité T, l'option américaine offre plus de flexibilité. La formulation de son prix reflète donc l'opportunité d'exercice à tout instant

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

L'expression $\mathbb{E}\big[e^{-r(t_{m+1}-t_m)}V\big(S_{t_{m+1}}, t_{m+1}\big) \mid S_{t_m}\big]$ notée aussi C, définit la valeur de continuation de l'option en t_m, c.à.d. son prix en t_{m+1}. A chaque instant, le

prix de l'option correspond au maximum, entre son payoff et sa valeur de continuation. En effet, le détenteur de l'option ne l'exercera pas si sa valeur future est supérieure au bénéfice qu'il peut en tirer en l'exerçant immédiatement.

2. Apprentissage automatique

Nous présentons ici, les différentes méthodes d'apprentissage automatique intégrées dans nos travaux: AdaBoost (Adaptive Boosting), SVM (machines à support de vecteurs) et forêts aléatoires. Une fonction de classification, dans notre contexte, renvoie une valeur binaire, selon que la valeur du panier d'actifs nous permette ou non d'exercer le contrat.

L'**Algorithme 1** décrit l'entrainement d'un classificateur AdaBoost [12]. L'entrainement consiste à calculer un classificateur « fort » comme une combinaison linéaire de classificateurs « faibles », qui dans notre cas sont des arbres de décision binaire à un niveau. A chaque itération, l'algorithme accorde plus de poids aux instances d'entrainement mal prédites, et pondère chaque classificateur faible de manière à minimiser la somme des erreurs sur chaque instance. On note d la dimension d'un point à classifier, correspondant dans notre contexte au nombre d'actifs du panier de l'option.

Algorithme 1 Entrainement d'AdaBoost.

Entrée : dimension d'un point à classifier d
Entrée : ensemble d'entrainement $(x_1, y_1), .., (x_m, y_m)$ $tq\ x_i \in \mathbb{R}^d, y_i = \pm 1$
Entrée : poids initiaux $w_1^0, \dots, w_m^0 = \frac{1}{m}$
Entrée : classificateurs faibles $h_1, .., h_n : x \to \pm 1$
Sortie : classificateur fort $H(x) \to \pm 1$

1 : Pour j **de** 1 **à** n
2 : [Choisir $\boldsymbol{h_j(x)}$**] :**
3 : Trouver $h_j(x)$ qui minimise $\epsilon_j = \sum_i w_i^j e^{-y_i h_j(x_i)}$
4 : Fixer $\alpha_j = \frac{1}{2} ln\left(\frac{1-\epsilon_j}{\epsilon_j}\right)$
5 : [Intégrer $\boldsymbol{h_j(x)}$ **au classificateur final] :**
6 : $F_j(x) = F_{j-1}(x) + \alpha_j h_j(x)$
7 : [Mise à jour des poids] :
8 : Pour i **de** 1 **à** m
9 : $w_i^{j+1} = w_i^j e^{-y_i \alpha_j h_j(x_i)}$ pour tout i
10 : Normaliser w_i^{j+1} tel que $\sum_i w_i^{j+1} = 1$
11 : Fin pour

12 : Fin pour
13 : Retourner $H(x) = \langle sign\big(F_n(x)\big)\rangle\, 1$

Les SVM sont un problème d'optimisation quadratique. Le but est de calculer l'équation de l'hyperplan, faisant office de marge maximale parmi les valeurs d'entrainement de classe différente (**Figure 1**).

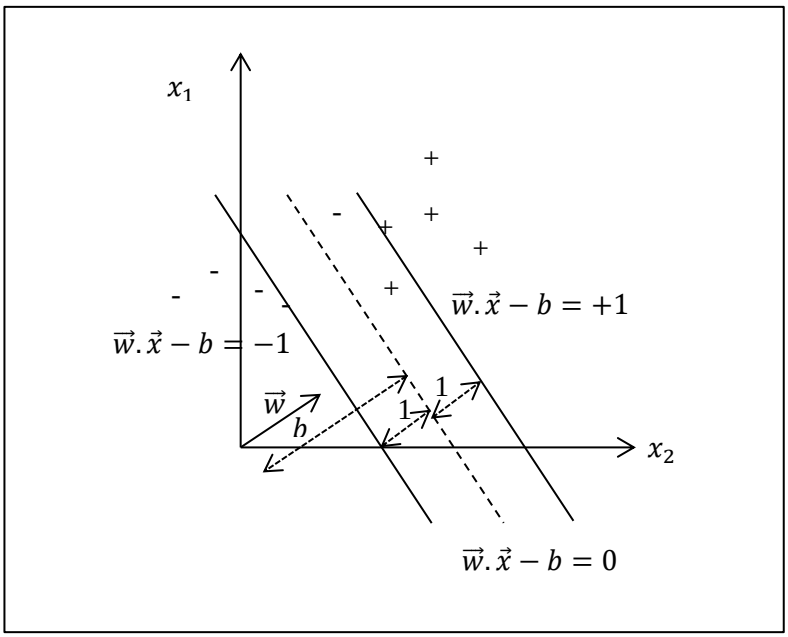

Figure 1 Exemple de SVM linéaire dans un plan à deux dimensions, représentant dans notre cas un panier à 2 actifs.

L'équation d'un SVM linéaire en notant $\vec{w}$ le vecteur normal à l'hyperplan est

$$\vec{w}.\vec{x} - b = 0$$

La marge séparant les points les plus proches de l'hyperplan est notée

$$m = \frac{1}{\|\vec{w}\|}$$

Maximiser cette marge revient à résoudre le problème d'optimisation sur les N points

$$\begin{cases} \min\limits_{\vec{w},b} \frac{1}{2}\|\vec{w}\|^2 \\ y_i(\vec{w}.\vec{x_i} - b) \geq 1\ , \forall i \end{cases}$$

Le Lagrangien permet d'obtenir la formulation duale du problème

$$\begin{cases} \max\limits_{\alpha} \sum\limits_{i=1}^{N} \alpha_i - \frac{1}{2} \sum\limits_{i=1}^{N} \sum\limits_{j=1}^{N} y_i y_j K(\overrightarrow{x_i}, \overrightarrow{x_j}) \alpha_i \alpha_j \\ 0 \leq \alpha_i \leq C, \forall i \\ \sum\limits_{i=1}^{N} y_i \alpha_i = 0 \end{cases}$$

α_i sont les multiplicateurs de Lagrange. Dans le cas de SVM non linéaire, K peut définir une fonction polynomiale, gaussienne, C contrôle la tolérance aux erreurs de classification, affectant directement l'orientation de l'hyperplan et la largeur de la marge. Nous renvoyons le lecteur à l'article de John C. Platt [13] présentant l'algorithme SMO (Sequential Minimal Optimization) pour calculer l'équation de l'hyperplan, en résolvant ce problème d'optimisation.

Les forêts aléatoires [14] combinent l'algorithme de boosting (bootstrap aggregating) avec le concept de sous-espaces aléatoires. L'idée est de construire un ensemble d'arbres décisionnels, qui voteront la classe la plus probable d'un point à classifier. La construction d'une forêt aléatoire suit l'**Algorithme 2**.

Algorithme 2 Entrainement d'une forêt aléatoire.

Entrée : nombre de dimensions à considérer pour chaque noeud p
Entrée : ensemble d'entrainement $(x_1, y_1), .., (x_m, y_m)$ $tq\ x_i \in \mathbb{R}^d, y_i = \pm 1$
Entrée : nombre d'arbres de la forêt à construire B
Entrée : hauteur d'un arbre h
Entrée : taille de l'échantillon bootstrap N
Sortie : forêt aléatoire H

1 : $H = \emptyset$
2 : **Pour** b **de** 1 **à** B
3 : Construire un échantillon bootstrap E_b en tirant aléatoirement N points parmi l'ensemble d'entrainement
4 : **Pour** chaque nœud **de** l'arbre T_b **jusqu'à** atteindre h
5 : Choisir aléatoirement un sous-ensemble de p attributs parmi d
6 : Choisir l'attribut parmi les p, qui découpe le plus équitablement l'échantillon en cours
7 : **Fin pour**
8 : $H = H \cup T_b$
9 : **Fin pour**
10 : **Retourner** H

La classification à l'aide d'un arbre de la forêt, s'effectue en le parcourant de la racine vers les feuilles : l'instance à classifier est comparée à la valeur de chaque nœud, selon l'attribut sélectionné dans l'algorithme de construction. La classification par la forêt résulte de la classification majoritaire de ses arbres.

3. Programmation répartie

La programmation répartie s'impose dans la résolution de problèmes couteux en ressource mémoire et en temps de calcul, via la sollicitation de plusieurs ressources physiques et en faisant ensuite en sorte de les faire travailler en parallèle. Par ailleurs, la répartition d'une application ne doit idéalement ni entraver sa lisibilité, ni son extensibilité. Plusieurs solutions de programmation répartie existent, mais la contrainte posée aux développements logiciels effectués dans notre travail impose d'utiliser la librairie ProActive développée antérieurement au sein de l'équipe.

La librairie ProActive basée sur une utilisation transparente de Java RMI (Remote Method Invocation), nous permet de manipuler des objets java sur différents nœuds, appelés objets actifs, en simplifiant le code de communication inter-objets. Plus encore, ProActive intègre un gestionnaire de ressources de calcul, nous permettant de supporter notre stratégie de gestion dynamique des ressources : étant donné une liste plus ou moins explicite de ressources physiques à acquérir (par exemple, au travers d'une réservation de nœuds de la plateforme Grid'5000), ProActive démarre sur chacune d'elles un nœud ProActive (point de contact distant). Il est alors aisé d'instancier des objets actifs sur ces différents nœuds, afin d'exploiter les ressources présentes, CPU comme GPU. L'équivalent d'une telle instanciation, en MPI-2 par exemple, se réaliserait grâce à la directive `MPI_Comm_spawn`.

De plus, ProActive permet le regroupement de plusieurs objets actifs du même type, en un groupe d'objets lui-même du type de celui des objets contenus. Par exemple, la création d'un groupe d'instances `workers` sur un ensemble de nœuds `workersNodes` du cluster, s'effectue ainsi

```
MonteCarloOptionWorker workers =
      (MonteCarloOptionWorker)PASPMD.newSPMDGroupInParallel(
            workerClassName,
            params,
            workersNodes);
```

Il est ensuite possible de travailler avec l'ensemble des objets distants comme avec un seul. Par exemple,

```
workers.doWork();
```

aura pour effet d'ajouter (par défaut en fin) l'appel de méthode `doWork()` dans la file d'appels de méthode en attente d'exécution sur chaque objet actif. A noter que dans la version standard de ProActive, un objet actif n'exécute qu'une seule de ces méthodes en attente à la fois: il est donc muni d'un unique thread d'exécution de ces méthodes.

La librairie ProActive permet une exécution asynchrone des méthodes demandées par d'autres objets actifs (y compris par lui-même), ou plus généralement par une thread. C'est-à-dire qu'un objet actif A effectuant un appel de méthode sur un objet actif B, n'attendra pas le résultat: une fois son appel de méthode déposé dans la file de B via le réseau, il pourra poursuivre son exécution sans avoir à se soucier du moment où la méthode sera effectivement exécutée, jusqu'à l'instant où il aura besoin de son éventuel résultat. Le développeur peut évidemment contrôler cette synchronisation via différents types de barrières, explicites ou implicites. La **Figure 2** illustre ce procédé d'exécution asynchrone, dans un exemple mêlant 3 threads, une thread principale issue de la méthode main d'une application, et deux threads associées à deux objets actifs (`merger`, `worker`) qui ont été préalablement instanciés par cette méthode main. Noter que tous les appels de méthode indiqués sur la figure sont donc des appels des méthodes effectués à distance.

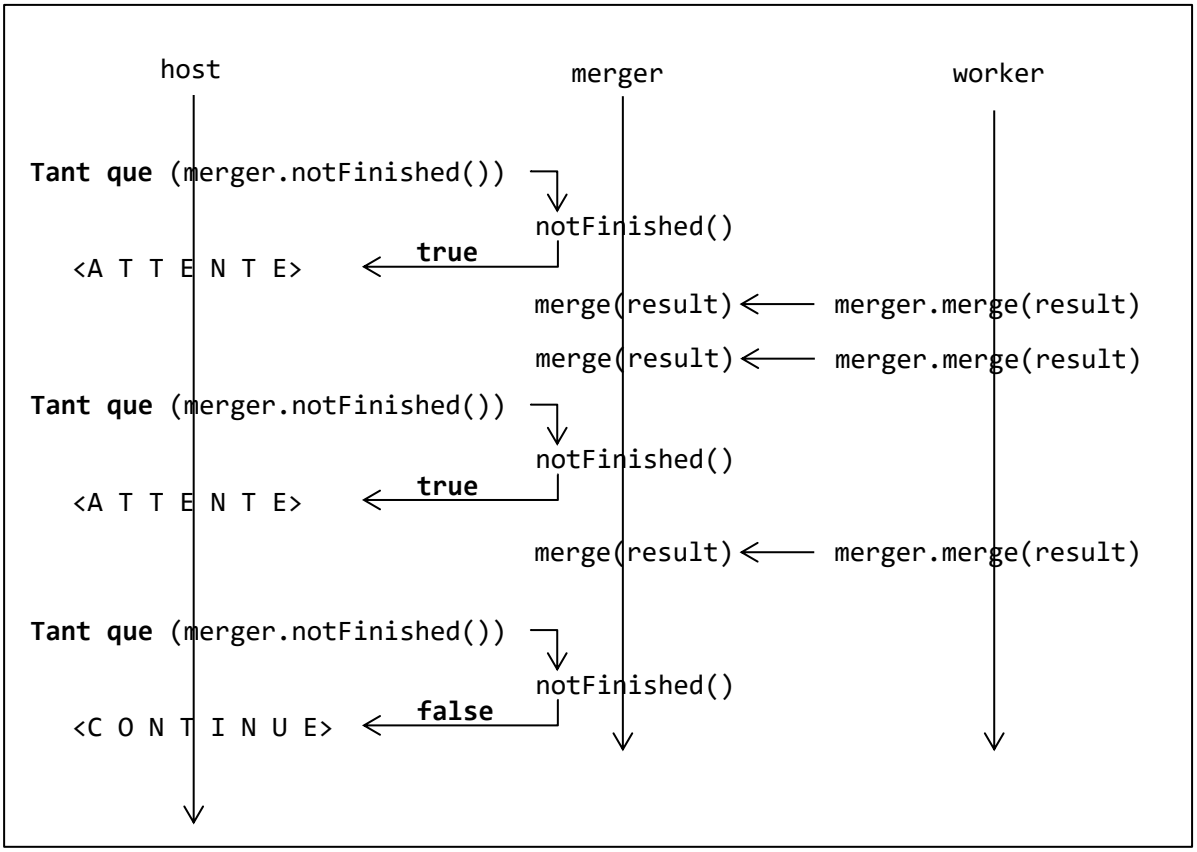

Figure 2 Fils d'exécution d'une thread principale et de deux objets actifs, dans un exemple d'agrégation de 3 résultat.

4. Programmation GPU

4.1. Architecture GPU

Programmer efficacement sur carte graphique nécessite une connaissance approfondie de son architecture [15]. Un GPU est composé de multiprocesseurs dotés de plusieurs cœurs (par exemple, une NVIDIA GeForce GTX 580 offre 512 cœurs répartis sur 16 multiprocesseurs). Selon la carte, un multiprocesseur détient un ou plusieurs *schedulers* de *warp*, orchestrant des *warp*s sur ses cœurs. On parle de warp pour les cartes NVIDIA, wavefront pour les cartes AMD. Un warp (de taille 32 threads pour les cartes NVIDIA) constitue le plus petit ensemble indissociable de threads opéré par les schedulers de warp [16]. Ainsi, un branchement conditionnel exécuté par un warp, sera exécuté séquentiellement, c.-à-d. embranchement par embranchement (phénomène connu sous le terme de warp divergence ou divergence de contrôle), impliquant un « gaspillage » de la puissance de calcul puisque les threads non concernées

par l'embranchement en cours d'exécution sont passives. Le nombre de cycles d'horloge requis pour son exécution varie d'une architecture à une autre. Sur une Tesla, l'exécution d'une instruction par l'ensemble d'un warp s'effectue en 4 cycles d'horloges, alors que l'architecture Kepler permet au warp d'enchainer 2 instructions par cycle d'horloge.

Une carte graphique est constituée de mémoire hors chipset (off-chip) et sur chipset (on-chip) comme l'illustre la **Figure 3**. La mémoire dite globale est off-chip et est accessible par l'ensemble des threads du kernel. On appelle kernel la fonction appelée par le CPU et exécutée sur GPU. Des variables peuvent être allouées en mémoire globale et non cachée (__global), ou en mémoire globale et cachée (__constant) les rendant accessibles qu'en lecture seule. En mémoire on-chip, on distingue la mémoire locale dite aussi partagée (__local) accessible par les threads d'un workgroup (cf 4.2 pour sa définition), et la mémoire privée à chaque thread stockée en registres. Notons que si le nombre de registres requis lors de l'exécution d'un kernel dépasse celui résidant sur la carte, la mémoire globale s'y substitue (register spilling), dégradant ainsi les temps d'accès aux données. La mémoire on-chip présente une faible latence permettant un débit théorique plus important que la mémoire off-chip (par exemple sur architecture NVIDIA Fermi, le débit en mémoire locale agrégée peut dépasser 1TByte/s contre 177GByte/s en mémoire globale).

4.2. OpenCL

OpenCL est un langage de programmation pour cartes GPU. Son ambition est d'être portable. Au contraire d'un langage tel CUDA qui ne peut s'exécuter que sur du matériel NVIDIA.

Lors de l'exécution d'un kernel défini en OpenCL, deux paramètres sont requis : le nombre total de threads, et le nombre de threads par workgroup, regroupement logique de threads. Un multiprocesseur peut gérer un ou plusieurs workgroups, chacun constitué de un ou plusieurs warps, afin de maximiser le nombre de threads actives sur chaque multiprocesseur, tout en respectant l'indissociabilité d'un warp.

Par exemple, on souhaite effectuer un branchement conditionnel (`if else`) au sein d'un workgroup de 64 threads, constitué sciemment de deux warps de 32 threads ; puis on veut s'assurer qu'il n'y ait plus d'accès par ses threads à une zone de mémoire locale, avant de poursuivre leurs exécutions. Pour éviter une divergence de contrôle, la première moitié des threads (donc dans le premier warp) réalisera la branche `if`, la seconde moitié la branche `else,` et une barrière de synchronisation est rajoutée avant que les threads du workgroup puissent exécuter la suite. La structure du code OpenCL correspondant est donc :

```
// get_global_id(0) <=> thread id dans le workgroup
if (get_global_id(0) < warpSize) {
// faire …
} else {
// faire …
}
// synchronisation des threads du workgroup
barrier (CLK_LOCAL_MEM_FENCE)
// suite du code
```

L'ensemble des threads d'un kernel se partagent le même espace en mémoire globale, et les threads d'un même workgroup se partagent le même espace en mémoire locale. Chaque thread accède à sa mémoire privée allouée en registres. La coalescence des accès en mémoire globale par un warp (ou même par un demi-warp) permet d'assurer un débit optimal en lecture/écriture, tout comme l'accès en mémoire locale. Nous renvoyons le lecteur en [17] et [18] pour plus de détails suivant les différentes architectures de cartes.

Les directives d'OpenCL reposent sur la mise en queue d'appels dans des files de commande. Par exemple `clEnqueueWriteBuffer`, `clEnqueueReadBuffer` permettent au CPU d'enfiler respectivement un appel d'écriture dans la mémoire globale, et un de lecture depuis la mémoire globale. Ces appels peuvent êtres bloquants ou non pour le thread appelant, selon un paramètre booléen.

Contrairement à CUDA, OpenCL permet d'exploiter les cartes graphiques des principaux fabricants (NVIDIA, AMD) mais aussi les CPUs Intel [19]. Par analogie aux GPUs, OpenCL considère un cœur CPU comme un multiprocesseur. Ainsi, un cœur CPU aura à charge 1 ou plusieurs workgroups. Le compilateur d'Intel peut scalariser les structures vectorielles déjà présentes dans un kernel, pour les vectoriser à nouveau (vectorisation implicite), en fonction de l'architecture vectorielle d'un cœur CPU (registres XMM,..).

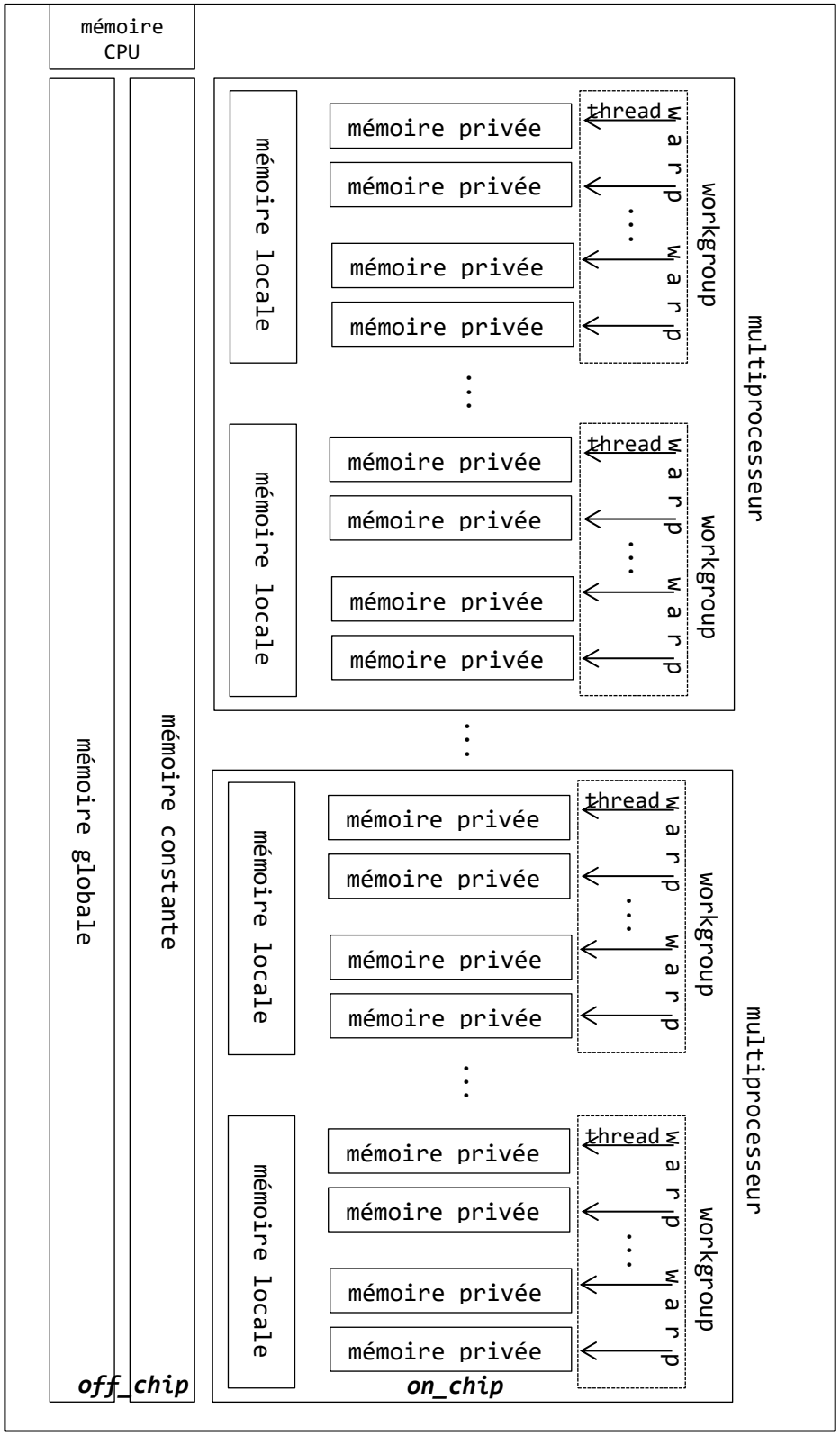

Figure 3 Découpage logique de la mémoire et des threads d'un GPU, sans scindement des warp.

Chapitre III. Etat de l'art

1. Introduction

La nécessité de diversifier les risques d'un portefeuille recourt aux produits optionnels. L'option américaine en fait partie. La difficulté de valoriser l'option américaine sur panier réside à la fois dans l'estimation de la frontière d'exercice, et la dimension de l'option, c.-à-d. la taille du panier. L'absence de solution analytique pour ce type de contrat suggère une approche par solutions quasi-analytiques ou par méthodes numériques.

En section 2, nous présentons par catégorie, les principaux travaux théoriques relatifs à la valorisation d'option américaine, avant d'énoncer en section 3, ceux abordant leurs parallélisations. L'intérêt de ce chapitre étant d'acquérir une connaissance générale des principaux travaux dans le domaine, et ainsi en section 4, de nous positionner selon certains critères jugés pertinents (scalabilité – passage à l'échelle, consommation mémoire,..).

2. Rapide panorama des méthodes de valorisation d'option américaine

2.1. Méthodes analytiques/quasi-analytiques

Nous ne pouvons commencer ce chapitre sans évoquer les travaux de Black, Scholes (B&S) [20] et Merton [21] établissant un modèle à deux actifs, risqué et non risqué (typiquement une action et une obligation). De leur modèle, les auteurs décrivent le prix de l'option européenne en t, selon l'équation différentielle partielle

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

avec S le prix de l'actif, $V(S,t)$ le prix de l'option, σ sa volatilité, r le taux d'intérêt sans risque. Geske et al [22] décomposent l'option américaine comme une série d'options européennes, et effectue une extrapolation de Richardson en 3 et 4 points (méthode itérative) pour l'approximation numérique. L'auteur conclut que sa méthode peut s'étendre pour évaluer des options de dimension 2. Les travaux de MacMillan [23], Barone-Adesi et al [24] présentent une approximation quadratique pour calculer la valeur critique du sous-jacent pour

lequel l'option est à la monnaie[2]. La convergence s'avère faible, en particulier pour des options de longue maturité. Joon [25], Jacka [26], Carr et al [27] formulent le prix d'un put américain comme la somme d'un put européen et d'une prime d'exercice anticipée. Broadie dans [28] estime les bornes inférieures et supérieures de l'option américaine représentée sous forme d'intégrale. Dans [29] l'auteur approche la frontière d'exercice par série de fonctions exponentielles. Contrairement à Geske, les auteurs dans [30] préfèrent une extrapolation de Richardson en 2 points et effectuent une approximation tangentielle de la première opportunité d'exercice. L'algorithme de Carr [31] connu sous le nom de *randomization*, est basé sur l'évaluation américaine à maturité tirée aléatoirement (suivant la loi d'Erlang). Zhu dans [32] propose une solution analytique sous forme d'une série de Taylor infinie.

2.2. Méthodes par arbre

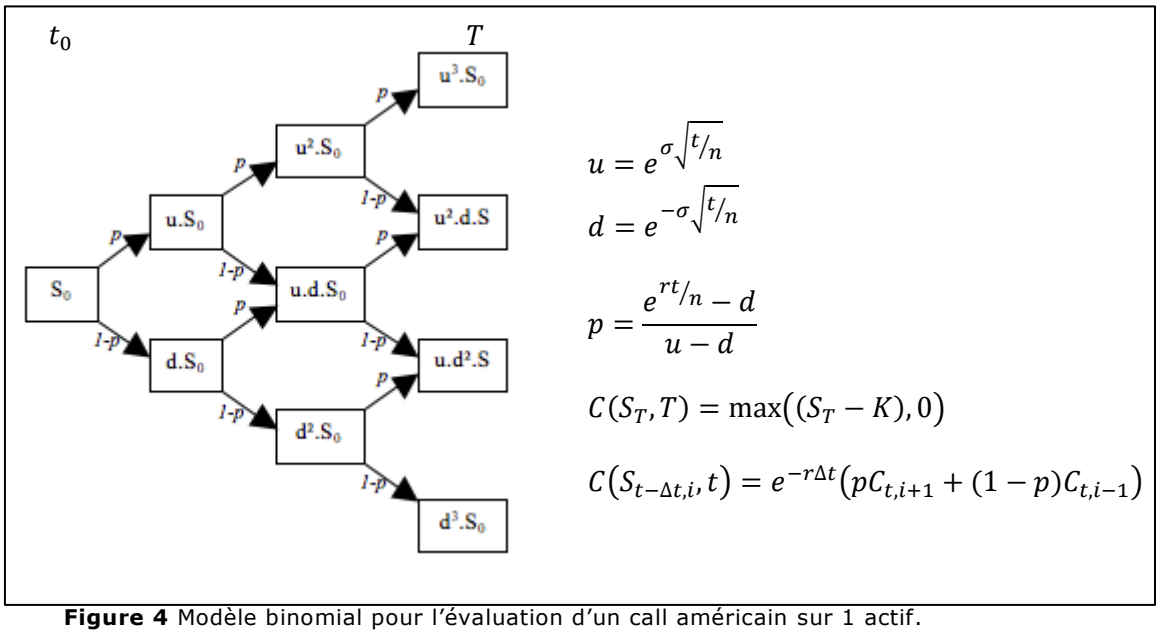

Figure 4 Modèle binomial pour l'évaluation d'un call américain sur 1 actif.

Le modèle binomial, aussi nommé modèle CRR, fut initié par Cox, Ross, Rubinstein [33] et Rendleman [34]. La première étape simule l'arbre des prix de l'actif. Ainsi, à chaque nœud i de l'arbre (**Figure 4**), le prix de l'actif croit ou décroit sur chaque Δt (cf u et d sur la figure, respectivement up et down).

[2] Une option est dite à la monnaie, si le prix du sous-jacent équivaut au srtke. Un call (respectivement un put) est dans la monnaie, si le prix du sous-jacent est strictement supérieur au strike (respectivement inférieur). Un call (respectivement un put) est hors la monnaie, si le prix du sous-jacent est strictement inférieur au strike (respectivement supérieur).

Ensuite, l'arbre des prix de l'option est construit des feuilles vers la racine. Pour valoriser un call américain, les feuilles correspondent aux payoffs à maturité $(S_T - K)$, et les valeurs des noeuds aux valeurs de l'option à chaque instant. La convergence du modèle binomial dans le cas de l'option américaine est démontrée dans [35]. Parkinson [36] et Boyle [37] initient le modèle trinomial (ajoutant une branche sans variation de prix), présentant l'avantage d'une convergence rapide à défaut d'une consommation en mémoire plus conséquente. Boyle [38] et Rubinstein [39] étendent le modèle binomial à des options sur panier. En particulier, Rubinstein combine 2 arbres binomiaux pour former une pyramide binomiale et ainsi valoriser une option sur 2 actifs. Breen [40] effectue une extrapolation de Richardson sur un arbre binomial standard pour réduire le nombre d'états. Broadie et al [28] améliorent le modèle binomial en deux points : ils introduisent la formule de B&S dans l'arbre des prix de l'option en $t - \Delta t$ et effectuent une extrapolation Richardson, améliorant ainsi la précision et la convergence du prix. Les travaux de Figlewski et al [41] améliorent considérablement la précision et le temps de calcul d'un arbre en augmentant sa densité dans les ramifications où l'option est dans la monnaie. L'auteur souligne que son approche s'adapte aisément aux options de grande dimension, ainsi qu'aux autres produits dérivés. Nous renvoyons aux travaux de Heston et al [42] sur l'étude de la précision du modèle CRR.

2.3. Méthodes itératives

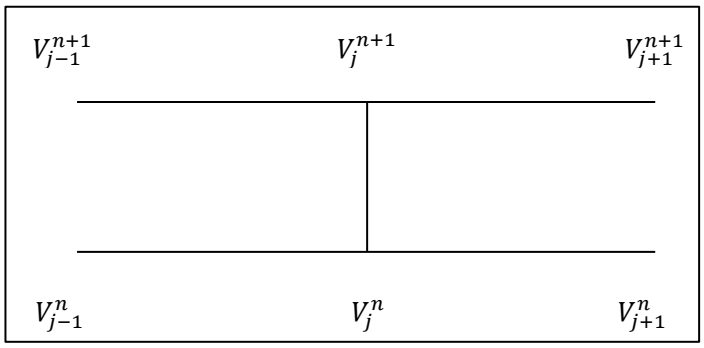

Figure 5 Maillage de différences finies et schéma de Crank-Nicolson.

Parmi les méthodes itératives répandues pour valoriser une option américaine, figure celle des différences finies initiée par Brennan et al [43]. Le principe est d'approximer une équation différentielle en temps continu, par sa forme discrétisée, selon le temps et le prix du sous-jacent. Ainsi, l'équation différentielle discrétisée se résout itérativement pour converger vers le prix de l'option. Les travaux de Jaillet et al [44] portent sur la convergence de cette

méthode. Le schéma de discrétisation adopté par Courtadon [45] et connu sous le nom de schéma de Crank Nickolson (**Figure 5**) est le suivant. En notant

$$V_j^n = V(j\Delta S, n\Delta t)$$

le schéma décrit la relation

$$\frac{V_j^{n+1} - V_j^n}{\Delta t} = \frac{1}{2}\left(\frac{V_{j+1}^{n+1} - 2V_j^{n+1} + V_{j-1}^{n+1}}{\Delta S^2} + \frac{V_{j+1}^n - 2V_j^n + V_{j-1}^n}{\Delta S^2}\right)$$

Ce schéma réduit l'erreur sur le temps et améliore la précision de l'approche de Brennan. Wilmott et al [46] utilise la méthode de relaxation (variante de la méthode de Gaus-Seidel) pour aborder le problème du pricing d'option sous forme d'un problème de complémentarité linéaire (LCP) tandis que Forsyth et al [47] appliquent la méthode des pénalités. Ikonen et al [48] fractionnent les pas de temps du problème posé sous sa forme LCP, et découplent ainsi le système linéaire de l'équation de B&S de sa contrainte d'exercice. Ito et al [49] utilisent le multiplicateur de Lagrange. Mayo [50], Oosterlee et al [51] Tangman et al [52] affinent le schéma en considérant un développement de 4eme ordre. D'autres travaux traitent de méta heuristique itératives, directement inspirées du domaine du vivant : réseaux neuronaux [53], programmation génétique [54] [55], algorithme des fourmis [56] [57], essaim particulaire [58].

2.4. Méthodes basées sur des simulations de Monte Carlo

Les travaux de Boyle [59] ont initié l'usage des simulations de Monte Carlo pour valoriser l'option européenne et ceux de Bossaert [60] abordent le problème d'exercice optimal. Le principe des méthodes recourant aux simulations de Monte Carlo dans le cas de l'option européenne, est de simuler un nombre conséquent de trajectoires de l'actif, et de calculer la moyenne actualisée des payoffs résultants. L'option américaine nécessite la prise en compte de la frontière d'exercice. L'intérêt de la méthode de Monte Carlo pour le pricing des options, réside dans la complexité qui ne dépend non pas de la dimension du problème mais du nombre de trajectoires simulées.

Pour valoriser un call américain sur un actif, Tilley [61] procède par bundle de prix : il simule nb_MC trajectoires de l'actif sous-jacent, les trie par ordre croissant des prix, calcule le payoff de chaque, et les partitionne en ensembles de P trajectoires. L'auteur estime ensuite les valeurs de continuation C de chaque trajectoire k ainsi

$$C(k,t) = e^{-r\Delta t}\frac{1}{P}\sum_{\forall j \; dans \; la \; même \; partition \; de \; k} V(j,t+1)$$

V est initialisée au payoff à maturité, puis est calculée de manière rétrograde ainsi. On compare les valeurs de continuation de chaque trajectoire avec son payoff, fixant ainsi un indicateur $x(k,t) = \{0;1\}$ (0 indiquant d'exercer, 1 sinon). On cherche parmi k l'indice de prix critique de l'action, c.-à-d. le début de la première séquence de 1 dont la longueur dépasse toutes celles de 0. On fixe ensuite $x(k,t) = 0$ pour $\forall k_i < k$, et $x(k,t) = 1$ sinon. $V(k,t)$ vaut ainsi $C(k,t)$ pour $x(k,t) = 0$, et le payoff sinon. Lorsque ce procédé est effectué pour $\forall t$, Tilley fixe une nouvelle variable $z(k,t) = 1$ si $x(k,t) = 1$ et si $x(k,s) = 0$ pour $\forall s < t$, 0 sinon. Enfin le prix actuel de l'option vaut

$$V = \frac{1}{nb\_MC}\sum_{\forall k}\sum_{\forall t} e^{-rt} z(k,t)\Psi(k,t)$$

Nous renvoyons le lecteur aux travaux de Grant et al [62] proposant aussi une approche rétrograde recourant aux simulations de MC. Barraquand et al [63] quant à eux, partitionnent l'espace de prix de l'actif pour réduire la dimension de l'option à 1. Ainsi, il est possible de pricer l'option réduite, par une approche rétrograde standard. Cependant pour certains cas d'option multidimensionnelle, la réduction n'assure pas un temps d'exercice optimal.

Un autre procédé combinant simulations et régressions, fut instauré par Carriere [64]. Tsitsiklis et al [65], et Longstaff et al [66], suivent cette approche. En particulier, Longstaff et al simulent dans un premier temps les trajectoires de l'actif jusqu'à maturité. Pour t de $T-1$ à 1, les auteurs calculent les cashflows en $t+1$ actualisés à t. Ce procédé est rétrograde car on sait que le cashflow de l'option en T équivaut au payoff. Ensuite, les auteurs estiment la fonction d'espérance conditionnelle des cashflows (Y) en t sachant les prix des actifs (X) en t, par régression quadratique

$$\mathbb{E}[Y|X] = a + bX + cX^2$$

Cette fonction s'applique alors aux prix des actifs en t pour en déduire les valeurs de continuation en t, et il devient possible de comparer ces dernières aux payoffs en t, pour fixer les cashflows en t, soit au payoff en t, soit à 0. On procèdera ainsi pour $t-1$, ainsi de suite jusqu'à t_1. Le prix de l'option s'obtient finalement en calculant la moyenne de l'ensemble des cashflows actualisés en t_0.

La méthode d'Ibáñez et al [67] repose sur l'estimation des valeurs critiques de l'actif à chaque temps discret, pour connaitre les temps d'arrêt des simulations de l'étape finale. Ces valeurs critiques sont estimées de manière

rétrograde sur les temps discrets, et chacune émane de la régression de « sous valeurs critiques ». Chaque sous valeur critique S^*, est approchée itérativement selon

$$C\left(S_{t_m}^{(1)}, t_m\right) = \Psi(S_{t_m}^{(2)}, t_m)$$
$$\left|S_{t_m}^{(n+1)} - S_{t_m}^{(n)}\right| \leq \varepsilon$$

C dénotant la valeur de continuation estimée par simulations de MC. $S_{T-1}^{(1)}$ est fixée au strike, et $S_{t_m}^{(1)}$ à $S_{t_{m+1}}^{*}$ pour $t_m < T - 1$.

Broadie et al [68] simulent aléatoirement un arbre, et calculent deux valeurs biaisées du prix de l'option convergeant asymptotiquement vers le prix non biaisé. L'inconvénient de cette approche est sa complexité exponentielle par rapport au nombre d'opportunités d'exercices. Les auteurs [69] adoptent ensuite un maillage stochastique (**Figure** 6) à nombre de points constant entre chaque temps, dont la complexité est liée linéairement aux opportunités d'exercice.

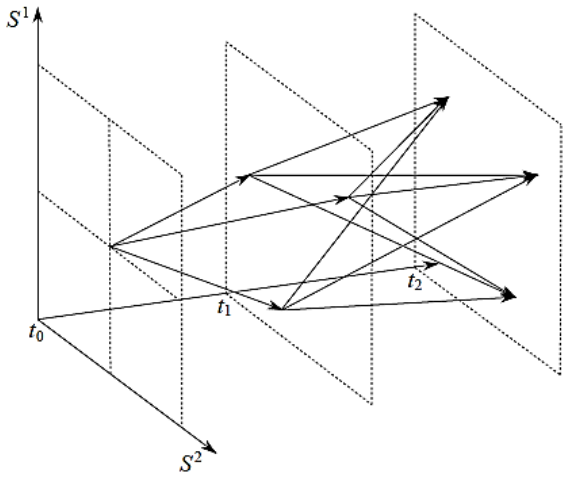

Figure 6 Exemple de maillage stochastique à 3 trajectoires par Δt, pour $d = 2$ et sur 2 Δt

Dans [70], Bally et al appliquent la méthode de quantification pour valoriser l'option américaine sur panier. Les variables aléatoires inhérentes aux processus de prix des actifs, considérés comme des chaines markoviennes, progressent selon les coefficients d'une matrice de transition, eux même estimés par simulations de MC.

3. Parallélisme des méthodes numériques

Dans cette partie, nous présentons brièvement les principaux travaux de parallélisation des méthodes numériques pour valoriser l'option américaine.

3.1. Méthodes par arbre

Tout l'enjeu de la parallélisation de cette approche réside dans le découpage et la répartition des calculs sur les ressources, pour limiter l'effet de goulot inhérent aux calculs d'espérance via les arbres. Typiquement, l'interdépendance des calculs entre 2 temps successifs, c.-à-d. 2 niveaux de l'arbre, nécessite d'alterner l'exécution de blocs de calcul judicieusement définis.

Gerbessiotis [71] propose une implémentation distribuée de l'arbre binomial, et valorise ainsi le put américain à 1 dimension sur un cluster de 16 CPUs. Bien que son implémentation ne conserve pas le même degré de parallélisme lors du calcul rétrograde de l'arbre, le speedup atteint dépasse 11. Podlozhnyuk [72] implémente une version GPU du modèle binomial, maximisant l'utilisation de la mémoire partagée entre threads d'un même bloc, au détriment de la mémoire globale, mais engendre des calculs redondants par les blocs de threads. Zubair et al [73] dans ses travaux, proposent une version non distribuée et une version multi-CPU du modèle binomial, réduisant les latences par optimisation de l'utilisation de la mémoire cache. La version non distribuée surpasse l'algorithme de [71], pour une option suffisamment complexe. Bien que la version distribuée ne permette pas de solliciter l'ensemble des processeurs durant toute l'exécution, son exécution sur 8 CPUs permet de dépasser un speedup de 7. Jauvion et al [74] implémentent en CUDA un modèle trinomial, qui étend la version de Podlozhnyuk, et attribue à chaque bloc de threads le pricing d'une option, leur permettant ainsi de saturer chaque multiprocesseur lors de la valorisation d'un portefeuille d'options. Dans [75], Peng et al implémentent une version multi-CPU du modèle binomial, pour simuler le mouvement brownien du processus stochastique de l'actif, et résoudre l'équation du prix de l'option exprimée sous la forme d'équation différentielle stochastique rétrograde (BSDE). Les auteurs limitent le nombre d'échange de données entre deux processeurs voisins, en introduisant un paramètre L, correspondant au nombre de niveaux de nœuds à calculer par processeur à chaque itération. Les tests sur cluster de 16 CPUs, permettent d'atteindre un speedup dépassant 10, pour une complexité élevée. Les travaux de Zhang et al [76] qui valorisent l'option américaine à une dimension, en tenant compte des frais du *market maker*, procèdent à un load balancing

dynamique des calculs durant l'exécution, pour dépasser un speedup de 5 avec 8 processeurs.

3.2. Méthodes itératives

L'auteur en [77] distribue la méthode de sur-relaxation pour évaluer l'option américaine exprimée dans sa forme LCP. Les tests sont effectués sur des options pour des paniers de 1 à 4 actifs, dont la dimension est préalablement réduite. Bien que la construction du maillage soit non distribuée et limite la dimension de l'option, l'exécution sur un cluster de CPU atteint un speedup super linéaire, et la valorisation d'une option américaine sur 4 actifs prend un peu plus de 30secondes, sans tenir compte de l'étape de construction. Dans [78], Dang et al implémentent en CUDA la méthode des pénalités pour aborder le problème sous sa forme LCP. Dans leur implémentation, ils répartissent les calculs matriciels tout en limitant les communications inter-threads, et proposent une méthode de sélection dynamique de taille des pas de temps, qui améliore leur précision du prix de l'option. Les auteurs atteignent ainsi un speedup dépassant 15 sur 1 GPU, lors de la valorisation d'une option américaine sur 3 actifs. Khodja et al [79] comparent les versions parallélisées pour cluster de CPUs et GPUs, de la méthode de Richardson et de relaxation par bloc. En particulier, les auteurs mettent en évidence par leurs tests, l'adaptabilité de la méthode de Richardson aux architectures SIMT, et de la méthode de relaxation par bloc aux CPUs.

Parmi les travaux notables sur la parallélisation d'algorithmes évolutionnaires et leurs applications dans notre domaine, Sharma et al [58] proposent une parallélisation sur GPU de la méthode d'optimisation par essaims particulaires (PSO). Ainsi, dans le cas du pricing d'option, une particule se déplace selon un prix (celui de l'actif) et un temps (temps d'exercice de l'option), et tient à jour une position optimale locale et globale. Chaque particule évolue aléatoirement vers une nouvelle position, en comparant les données de position optimale locale et celles futures : payoffs et temps d'exercice. A chaque déplacement des particules, la position optimale globale est fixée comme la meilleure des optimales locales. L'intérêt d'un tel modèle est de pouvoir intégrer aisément une volatilité dynamique, lors du calcul des vélocités des particules. Dans son implémentation, l'auteur attribue pour chaque thread GPU la simulation d'une particule : mise à jour de la position et de la vélocité, et effectue ses réductions en mémoire locale. Le speedup ainsi atteint pour le pricing d'option américaine sur 1 GPU dépasse 40.

3.3. Méthodes basées sur des simulations de Monte Carlo

Pour valoriser des options sur 5 actifs, Toke et al [80] parallélisent sur cluster de CPU l'algorithme de Ibáñez et al [67] (détaillé en 2.4) dans sa première étape lors des estimations indépendantes des prix critiques du sous-jacent, et dans l'étape finale des simulations de Monte Carlo. Doan [5] étend ces travaux en étudiant la sensibilité du prix de l'option aux paramètres du modèle, sur une implémentation multi-CPU.

Doan dans ses travaux [5] parallélise sur cluster de CPU la méthode de Picazo. Par une approche maitre-esclave, il estime les instances d'entrainement des classificateurs (AdaBoost ou SVM) ainsi que les simulations de MC de l'étape finale. Une option américaine sur 40 actifs est ainsi valorisée en moins de 9 heures sur un cluster de 64 cœurs CPU. Nous y reviendrons plus en détail en Chapitre IV.3.

L'un des principaux freins à la parallélisation de la méthode de Longstaff est l'étape de régression, faisant l'objet des travaux de Choudhury et al [81], Abbas-Turki et al [82]. Plus tard, ces derniers [83] implémentent une version GPU de l'algorithme que nous décrirons en Chapitre IV.6. Peng et al [84] optimisent la méthode de Longstaff pour la valorisation sur GPU d'options américaines de grande dimension, et obtiennent un speedup de 29 pour un panier de 7 actifs.

Dai et al [85] parallélise sur GPU la méthode du thêta pour résoudre l'équation différentielle stochastique rétrograde de l'option. En outre, l'équation en temps continu est discrétisée et les calculs d'espérance s'effectuent en chaque point par simulations de MC, leur permettant d'atteindre un speedup supérieur à 230.

Pagès et al [86] implémentent sur un GPU la méthode de quantification et atteignent un speedup de 200.

Zhang et al [87] valorisent l'option bermudienne sur panier, en estimant les bornes inférieures et supérieures par simulations de MC et la borne supérieure de son prix, et la méthode de Longstaff pour la borne inférieure. Les auteurs proposent une version multi-CPU et améliorent considérablement leurs temps de calcul (quelques minutes à quelques secondes).

Les travaux de Wan et al [88] portent sur la parallélisation de la méthode de maillage stochastique, et exposent un speedup quasi optimal sur 8 CPUs, pour des options de dimension 5. Le calcul du maillage est distribué, et deux approches sont proposées pour le calcul rétrograde des valeurs de continuation, en partitionnant les données pour réduire les couts de communication. La méthode proposée par Jain et al [89], combine maillage stochastique et méthode des moindres carrés. Une première étape consiste à simuler la grille des prix du panier. Les auteurs simulent le maillage des prix des actifs du panier, et calculent les payoffs associés à maturité. Lors de l'étape

dite de *bundling*, les auteurs procèdent aux regroupements des prix à chaque temps discret (par clustering par exemple), formant ainsi des bundles de taille variable. Une régression est ensuite calculée au sein de chaque bundle, et un calcul d'espérance conditionnelle sur l'ensemble des régressions, permet d'obtenir la valeur de continuation pour ce temps discret. Les travaux de Leitao Rodríguez et al [90] améliorent leur précédente version. En effet, l'emploi de l'algorithme de clustering des *k-means* pour construire les bundles reste trop couteux en temps de calcul et en mémoire, dans le cas d'options de grandes dimensions. De plus, les tailles variées des bundles ne permettent pas d'assurer pour certains, un nombre de points suffisants pour une régression précise. Pour y remédier, les auteurs trient les prix selon certains critères et les partitionnent uniformément ; moins de bundles sont nécessaires pour l'estimation des valeurs de continuation, et l'équilibrage des tâches est facilité. Leur parallélisation sur GPU permet de valoriser une option bermudienne sur 50 actions en 27 secondes. Yonghong et al [91] proposent une implémentation du maillage stochastique sur l'architecture CPU multi-cœur Intel nouvelle génération (Many Integrated Core), atteignant ainsi un speedup de 28.

3.4. Méthodes par transformée de Fourier

Surkov [92] résout numériquement de manière rétrograde par transformées de Fourier discrètes, l'équation intégro-différentielle de l'option américaine, et propose une implémentation GPU. L'auteur sollicite les librairies FFTW et CUFFT pour les calculs de transformée de Fourier sur CPU et GPU respectivement. Il améliore ainsi de 7 fois et demie le temps de calcul sur CPU avec son implémentation GPU, pour la valorisation d'une option américaine particulière (double-trigger stop-loss option). Zhang et al [93] implémentent en CUDA, une technique basée sur la représentation du prix de l'option européenne et bermudienne, en série de Fourier de cosinus.

4. Bilan et positionnement

Nous concluons ce chapitre en dressant un tableau comparatif selon certains critères, des principales méthodes parallélisées de valorisation d'option américaine. Nous retenons comme critères comparatifs : la facilité de compréhension, la scalabilité, la dépendance entre la consommation en mémoire et la dimension de l'option, la dépendance entre la complexité de calcul et la dimension de l'option, la convergence.

	Méthodes par arbre	Méthodes itératives	Méthodes basées sur des simulations de Monte Carlo	Méthodes par transformée de Fourier
Compréhension	++	+	+	-
Scalabilité	-	+	++	+
Consommation mémoire	-	+	++	+
Complexité calcul	-	+	+	+
Convergence	+	+	-	++

Figure 7 Tableau dressant les atouts et contraintes des différentes méthodes.

La table en **Figure 7** ne permet pas de conclure sur la supériorité d'une méthode de valorisation d'option américaine pour l'ensemble des critères, mais plutôt de les départager sur ces différents points.

La méthode par arbre tire son intérêt de son intuitivité, en décrivant de manière simple les trajectoires du sous-jacent et les calculs d'espérance conditionnelle. Les calculs sur les arbres de prix restent peu adaptés au parallélisme, bien que de nombreux travaux proposent des découpages de données pour exploiter au mieux le parallélisme d'une architecture multi-cœurs sollicitée. De par sa structure, la complexité croit exponentiellement en fonction du nombre d'états, la rendant inadaptée au pricing d'option sur panier, sauf pour de petites dimensions où des arbres binomiaux multidimensionnels sont employés. L'emploi d'arbres trinomiaux améliore la convergence du prix de l'option.

Les méthodes itératives dans un large spectre (extrapolation de Richardson, algorithmes évolutionnaires,..), bien que moins intuitives que la méthode par arbre, offrent une vision matricielle des processus de prix à chaque itération du modèle, jusqu'à sa convergence. Cette vision matricielle, aussi partagée par les méthodes numériques basée sur la transformée de Fourier, est

parfaitement adaptée aux architectures GPU, et permet une convergence étroitement liée à la taille de l'espace discrétisé.

Les méthodes basées sur les simulations de Monte Carlo exposent clairement les processus de prix des actifs, facilitant la compréhension des algorithmes. Celles-ci se prêtent parfaitement à la valorisation d'options de grande dimension, du fait que la convergence vers le prix réel de l'option est indépendante de la taille du panier (Chapitre II.1.1). De plus, la dimension de l'option influe principalement sur la simulation du panier. Le nombre de simulations de MC conditionne la précision de l'estimation, et peut s'avérer couteux si l'option est complexe. Différents procédés de réductions de variance permettent de diminuer ce nombre requis. Mais tout l'intérêt de nos travaux est de tacler le problème de complexité de calcul de MC, non d'un point de vue mathématique mais technique : en exploitant dynamiquement et au mieux une architecture distribuée et parallèle. Parmi ces méthodes existantes, l'algorithme de Picazo présente l'avantage majeur de simuler la frontière d'exercice indépendamment des simulations finales. Ainsi, cette même frontière d'exercice peut être réutilisée lors de multiples valorisations du même instrument ; ce qui est le cas lors de la MC VaR.

Chapitre IV. Valorisation d'option américaine sur GPU

1. Introduction

Nous présentons ici notre implémentation GPU de la méthode de valorisation d'option américaine sur panier de Jorge Picazo. Celle-ci repose à la fois sur la méthode de Monte Carlo pour estimer la valeur espérée du prix de l'option, et sur l'apprentissage automatique pour estimer sa frontière d'exercice. Nous synthétisons à nouveau l'intérêt de cette méthode par rapport aux autres, avant de la détailler. Nous exposons ensuite sa parallélisation en tenant compte du paradigme SIMT (Single Instruction Multiple Threads) des GPUs. La suite du chapitre regroupe les détails techniques de notre implémentation : choix des langages et des librairies, structures et organisation des données pour réduire les temps de calcul. Une stratégie dynamique d'estimation des paramètres de kernel y est présentée pour exploiter pleinement les différentes cartes du marché. Enfin, des tests validant notre implémentation y sont présentés, ainsi que d'autres analysant ses performances selon les paramètres de calcul.

2. L'algorithme de Picazo

La méthode de Picazo basée sur celle de Monte Carlo, présente de nombreux avantages. Pour les options américaines de dimension élevée, il n'existe pas de solution analytique, et les méthodes numériques de résolution d'équations aux dérivées partielles (EDP), comme les méthodes des différences finies ou le modèle CRR, sont plus complexes à mettre en œuvre. De plus, la valorisation par simulations de Monte Carlo permet de contrôler la convergence du prix de l'option quelle que soit sa dimension, en jouant uniquement sur le nombre de simulations. La simplicité et la flexibilité de cette méthode permet de valoriser tout type d'instrument dont le rendement suit un processus stochastique, quelle que soit sa distribution (normale, log normale,..). Enfin et surtout, la simulation de ces processus stochastiques est aisément parallélisable, permettant ainsi de réduire considérablement son temps de calcul.

La spécificité de la méthode de Picazo par rapport à d'autres également basées sur l'usage de simulations de MC ([66] [67]), est d'estimer la frontière d'exercice *indépendamment* des prix de départ du panier. Le prix de l'option peut donc être réévalué tout au long de sa vie, en conservant la même frontière d'exercice. Cela permet de s'affranchir de l'étape la plus de coûteuse de l'algorithme si plusieurs estimations de prix sont nécessaires (cas de la Monte Carlo VaR par exemple).

Algorithme 3 Méthode de Picazo.

Entrée: $S_{t_0}^i, d, r, \delta_i, \sigma_i, N, T$
Entrée: nombre de points de classification nb_class
Entrée: nombre de trajectoires par valeur de continuation nb_cont
Entrée: nombre de trajectoires finales nb_MC
Sortie: le prix de l'option

[phase 1]:
Pour m **de** $N-1$ **à** 1
  Générer nb_class points de $\left\{S_{t_m}^{i,(s)} : i = 1, \dots, d; s = 1, \dots, nb\_class\right\}$
  [étape 1]:
  Pour s **de** 1 **à** nb_class
    Calculer $C\left(S_{t_m}^{i,(s)}, t_m\right) = E\left[e^{-r(t_{m+1}-t_m)} V\left(S_{t_{m+1}}^{i,(s)}, t_{m+1}\right) | S_{t_m}^{i,(s)}\right]$ avec nb_cont trajectoires et calculer $\psi\left(S_{t_m}^{i,(s)}, t_m\right)$
    Si $C\left(S_{t_m}^{i,(s)}, t_m\right) \leq \psi\left(S_{t_m}^{i,(s)}, t_m\right)$ alors
      Signe = 1
    Sinon
      Signe = -1
    Fin Si
  Fin Pour
  [étape 2]:
  Entrainer une fonction de classification sur $\left\{\left(S_{t_m}^{i,(s)}, signe\right) : s = 1, \dots, nb\_class\right\}$ pour définir la frontière d'exercice en t_m, c.-à-d. H_{t_m}
Fin Pour
[phase 2]:
Générer nb_MC trajectoires $\left\{S_{t_m}^{i,(s)} : i = 1, \dots, d; m = 1, \dots, N; s = 1, \dots, nb\_MC\right\}$ En utilisant la frontière d'exercice estimée en tout t, on estime le prix de l'option
Retourner le prix de l'option

L'algorithme de Picazo y est ici détaillé (**Algorithme 3**). Notons d la taille du panier, c.à.d. la dimension de l'option, S_t^i, δ_i et σ_i respectivement les prix, dividendes et volatilités des $i = 1..d$ actifs risqués sous-jacents, T la maturité, N le nombre de temps discrets, r le taux sans risque. Le principe de cette méthode est d'appeler une fonction de classification à chaque temps discret pour chacune des nb_MC trajectoires de la phase finale **[phase 2]**, pour décider de leurs temps d'arrêt, c.à.d. si les prix simulés ont atteint la frontière d'exercice ou non. Pour cela, chaque fonction de classification est préalablement entrainée **[phase 1] [étape 2]** sur nb_class variables d'entrainement. Chaque variable d'entrainement est composée des prix simulés du panier et d'une valeur binaire, résultant de la comparaison du payoff et

d'une valeur de continuation C. L'estimation de chaque valeur de continuation requiert nb_cont simulations de MC **[étape 1]**. En conséquence, nb_cont simulations sont requises par variable d'entrainement. Comme pour les trajectoires de la phase finale, la simulation des valeurs de continuation requiert les appels aux fonctions de classification pour prédire leurs temps d'arrêt. Cependant, ces fonctions ne seront estimées en totalité qu'en fin de **[phase 1]**, d'où la nécessité de calculer en premier la fonction de classification du dernier pas de temps avant maturité t_{N-1}. Ainsi, les trajectoires simulées des valeurs de continuation atteignent immédiatement t_N, sans transiter par un pas de temps intermédiaire qui ferait appel à une fonction de classification. La fonction de classification en t_{N-1} est ainsi entrainée. Lors de l'estimation de la fonction pour t_{N-2}, les trajectoires des valeurs de continuation atteignent t_{N-1} avant t_N, faisant appel à la fonction précédemment entrainée. La seconde fonction est ainsi entrainée, et ainsi de suite jusqu'à t_1.

3. Adaptation pour un seul GPU

Interrogeons-nous sur quelles sections de l'algorithme déléguer au GPU. La boucle principale de la **[phase 1]** ligne 1 sur les temps discrets ne peut pas être parallélisée, due à l'interdépendance des entrainements des fonctions de classification. En effet, la fonction à t_{i+1} est sollicitée lors de l'estimation des variables d'entrainement de la fonction à t_i. Les travaux précédents [5] de Viet Dung Doan proposaient une adaptation multi-CPU selon une stratégie maitre-esclave. Les estimations des variables d'entrainement de chaque fonction de classification étant indépendantes, celles-ci étaient distribuées par une thread « maître » aux threads « esclaves », chacune à charge d'un CPU du cluster. Afin d'exploiter au mieux un cluster hétérogène, les esclaves travaillaient de manière asynchrone et requêtaient sans cesse le master pour l'attribution de nouvelles variables d'entrainement à estimer, jusqu'à atteindre globalement nb_class. Bien que l'architecture d'une carte graphique offre un niveau de parallélisme élevé (+1000 cœurs pour l'architecture NVIDIA Kepler ou l'architecture AMD Tahiti), ses ressources couvrent difficilement celles requises par chaque thread du kernel, si une partie des variables d'entrainement lui était distribuée. D'où l'intérêt de solliciter les nombreux cœurs dont dispose un GPU pour simuler les nb_cont trajectoires d'une valeur de continuation. L'intérêt est double, car en spécialisant notre kernel pour les simulations de MC, nous sollicitons ce même kernel lors de la **[phase 2]**, pour estimer le prix de l'option.

Algorithme 4 Méthode de Picazo parallélisée pour un GPU.

Entrée: $S_{t_0}^i, d, r, \delta_i, \sigma_i, N, T$
Entrée: nombre de points de classification nb_class
Entrée: nombre de trajectoires par valeur de continuation nb_cont
Entrée: nombre de trajectoires finales nb_MC
Sortie: le prix de l'option

1 : Déclarer dans la mémoire globale du GPU $d, r, \delta_i, \sigma_i, T, nb\_cont$
[phase 1]:
2 :Pour m **de** $N-1$ **à** 1
[étape 1]:
3 : Générer $S_{t_m}^{i,(1)}$
4 : **Pour** s **de** 1 **à** nb_class
5 : Transférer vers la mémoire globale du GPU $S_{t_m}^{i,(s)}$
6 : Exécuter le kernel estimant $C\left(S_{t_m}^{i,(s)}, t_m\right)$
7 : Générer sur CPU $S_{t_m}^{i,(s+1)}$ en parallèle de l'exécution du kernel
8 : Transférer vers la mémoire du CPU le résultat d'exécution du kernel
9 : Calculer $\psi\left(S_{t_m}^{i,(s)}, t_m\right)$ et $\left(S_{t_m}^{i,(s)}, signe\right)$
10 : **Fin Pour**
[étape 2]:
11 : Entrainer une fonction de classification H_{t_m}
12 :Fin Pour
[phase 2]:
13 : Transférer vers la mémoire globale du GPU $S_{t_0}^i, nb\_MC$
14 : Exécuter le kernel estimant le prix de l'option
15 :Retourner le prix de l'option

En conséquence, l'**Algorithme 4** présente une version, certes simplifiée, de l'implémentation de l'**Algorithme 3** sur un GPU. Toutes les données indépendantes de l'itération sur les temps discrets (ligne 2) et sur les valeurs de continuation (ligne 4) sont initialement transférées sur GPU (mémoire globale cachée et non cachée) : taux sans risques, dividendes, volatilités, matrice de Choleski pour la corrélation des processus de prix,.. A contrario, certaines données sont transférées sur GPU à chaque estimation de C (ligne 4) : prix du panier, graine du générateur pseudo aléatoire,.. Chaque graine transférée permet d'initialiser le générateur, générant ainsi pour chaque thread du GPU et à chaque exécution du kernel, une séquence distincte. Notons que pour exécuter un kernel en OpencCL, la méthode `clEnqueueNDRangeKernel(…)` permet d'ajouter la commande dans une queue FIFO, le CPU reprenant la main après. Nous profitons donc de l'exécution éventuelle du kernel pour simuler pendant ce temps sur CPU les processus de

prix du panier d'actifs avant de les transférer en mémoire globale du GPU pour la prochaine itération. Une fois sur GPU, ces prix du panier d'actifs servent de point de départ pour les simulations de MC de C. A la fin de chaque estimation de C, sa valeur est comparée au payoff calculé à partir des prix du panier pour enregistrer une instance d'entrainement.

Notre principale difficulté pour adapter cet algorithme à une architecture SIMT, provient des longueurs aléatoires des nb_cont trajectoires simulées au sein de chaque exécution de kernel. Dans la suite, on mesure la longueur d'une trajectoire en nombre de pas de temps. Naïvement (**Figure 8 - 1)**, nous pourrions attribuer à chaque thread un même nombre de trajectoires à simuler. Cependant, certains threads simulant de courtes trajectoires, seraient susceptibles de quitter la boucle itérant sur le nombre de trajectoires, avant d'autres threads du même warp : la divergence de contrôle à ce niveau dégraderait le taux d'occupation des multiprocesseurs.

La solution apportée d'un point de vue algorithmique est présentée en **Figure 8 - 2**. Les threads d'un même workgroup (regroupant des warp) simulent de manière synchrone `nbTimeStepsBeforeReduction` pas de temps, avant de calculer via une réduction intermédiaire (selon un opérateur de somme), le nombre de trajectoires simulées en son sein. Si après `nbTimeStepsBeforeReduction` pas de temps simulés, certains threads n'ont pas terminé leurs trajectoires (temps d'arrêt ou maturité non atteints), ceux-ci les poursuivront à la prochaine itération ; une thread redémarre donc une nouvelle trajectoire des prix du panier, à partir de S_{t_m} (**Algorithme 4** ligne 3) indépendamment des `nbTimeStepsBeforeReduction` pas de temps. On estime dynamiquement `nbTimeStepsBeforeReduction` au sein de chaque workgroup. Ceci est fait préalablement à la boucle en faisant simuler à chaque thread exactement une trajectoire et comptabilisant les temps d'arrêt; puis, en effectuant une réduction (selon l'opérateur maximum) des temps d'arrêts des threads composant le workgroup, pour en déduire le temps d'arrêt maximum. Le maximum, car en effet, lors de l'exécution du kernel, les threads simulent les browniens géométriques à partir des mêmes prix $S_{t_m}^i$; Et si ces prix évoluent de manière assez semblable, et qu'ils sont encore éloignés de la frontière d'exercice, il n'est pas nécessaire d'effectuer trop souvent des réductions intermédiaires (**Figure 8 - 2**), c.-à-d. fixer un `nbTimeStepsBeforeReduction` petit. Ces prix évoluant de manière assez semblable seulement si des mouvements browniens sont de faible volatilité. Cette hypothèse est donc clé pour justifier l'intérêt de fixer le même `nbTimeStepsBeforeReduction`, quelle que soit la simulation en cours au sein d'une itération.

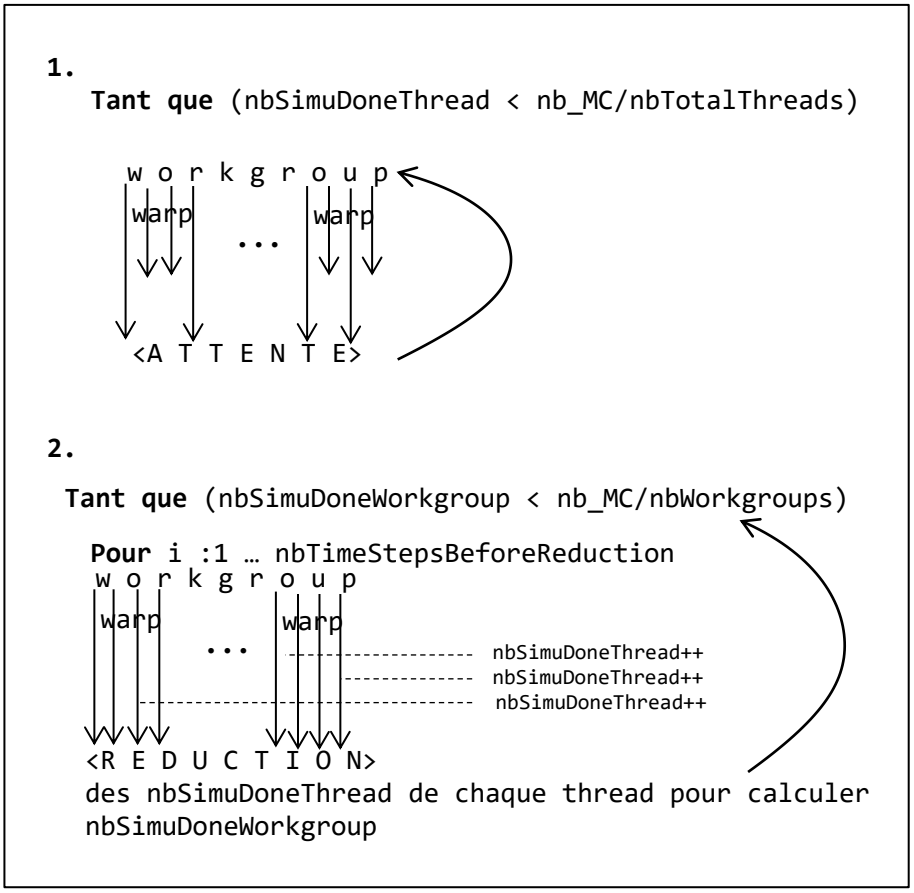

Figure 8 Comportement schématique des warps au sein d'un workgroup: 1. sans réductions intermédiaires, 2. avec réductions intermédiaires.

Plus précisément, chaque thread procède ainsi entre t et $t+1$: génération pseudo-aléatoire uniforme, transformation de Box-Muller pour en déduire les gaussiennes, corrélation des gaussiennes via la matrice de Choleski, simulation des mouvements browniens géométriques des actifs du panier en $t+1$, calcul de la moyenne arithmétique ou géométrique du prix des actifs, prédiction si la région d'exercice est atteinte via un appel de la fonction de classification spécifique à $t+1$, si tel est le cas, enregistrement du payoff actualisé et incrémentation du compteur `nbSimuDoneThread` comme le schématise la **Figure 8 - 2**. A chaque fin d'itération sur `nbTimeStepsBeforeReduction`, on calcule `nbSimuDoneWorkgroup` par réduction (somme) des `nbSimuDoneThread` des threads de chaque workgroup.

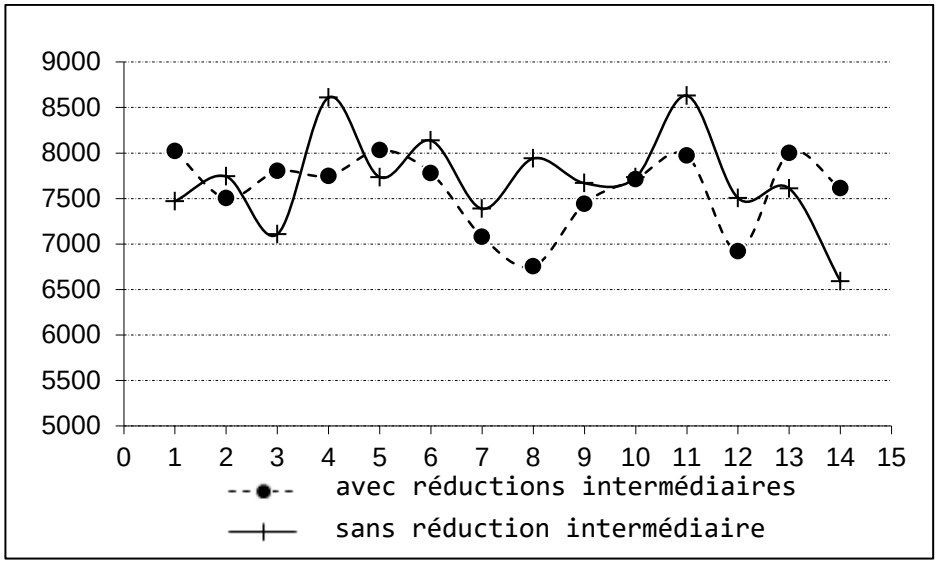

Figure 9 Temps de classification selon deux approches, en secondes pour 14 expérimentations.

Nous effectuons nos tests pour $N=100$ avec des paramètres de classification élevés, afin de comparer notre méthode proposée **Figure 8 – 2** à une approche naïve **Figure 8 – 1**. En effet, N petit ne permet pas de constater une variation suffisante des temps d'arrêts de trajectoire d'une thread à l'autre, dans la solution naïve. De plus, notre problème implique un temps de résolution variant d'une exécution à l'autre. Comme la figure l'illustre, notre stratégie lisse quelque peu le nombre de pas de temps des trajectoires simulées sur le workgroup, grâce aux réductions intermédiaires imposées dans chacun d'eux. Ainsi comme l'illustre la **Figure 9**, notre solution réussit dans la pratique, à réduire les durées globales de création des instances d'entrainement.

Néanmoins, notre seconde solution ne permet pas d'obtenir un même nombre de nb_cont simulations pour chaque valeur de continuation, et introduit en conséquence un biais dans le prix de l'option. En effet, nous avons mené de nombreuses expériences où nous avons joué très artificiellement sur le nombre de trajectoires exécutées pour chaque valeur de continuation (par exemple en doublant nb_cont), nous conduisant à suspecter qu'une frontière d'exercice entrainée avec des instances dont le nombre de simulations varie d'une valeur de continuation à une autre, introduit un biais dans le prix de l'option. Pour corriger ce biais, nous proposons de combiner les deux méthodes pour simuler exactement nb_cont trajectoires, comme résumé en **Figure 10**. Ainsi, une dernière boucle sur le nombre de trajectoires restant permet de simuler exactement les nb_cont ou nb_MC trajectoires désirées selon les phases de l'algorithme.

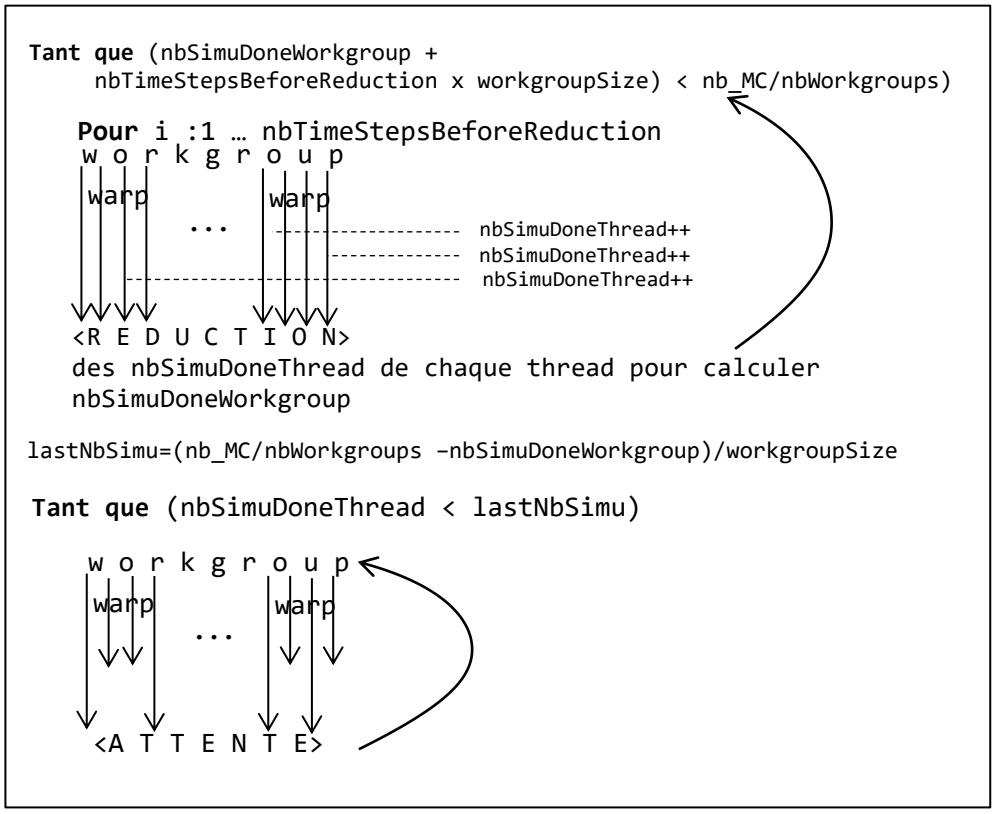

Figure 10 Comportement schématique des warps au sein d'un workgroup en combinant les méthodes 1 et 2 de la **Figure 8**.

4. Implémentation et Optimisations

4.1. Technologies utilisées

Notre application cible différents types de carte graphique, d'où la nécessité d'utiliser OpenCL [94] De plus, nous avons comme objectif (Chapitre V) d'étendre notre application pour exploiter un cluster de GPUs éventuellement hétérogène, via l'usage supplémentaire de la librairie Java ProActive. Nous utilisons donc la librairie JOCL [95], un wrapper Java d'OpenCL, regroupant l'ensemble des directives OpenCL. La librairie d'apprentissage Weka [96] nous permet de travailler avec différents algorithmes d'apprentissage automatique.

4.2. Implémentation OpenCL

Nous renvoyons le lecteur au Chapitre VII.3 pour l'implémentation détaillée d'un kernel, mais cette section présente les principes qui ont été appliqués.

La librairie Weka nous permet côté CPU, d'entrainer des fonctions de classification de type AdaBoost (Adaptive Boosting) ou SVM (Support Vector Machine), à l'aide d'instances d'entrainement (fonction `buildClassifier()` de Weka), côté GPU, de prédire les temps d'arrêt de chaque trajectoire simulée en utilisant idéalement directement la fonction `classify()` de Weka. En effet, il aurait été naturel de solliciter cette même librairie pour l'entrainement (construction du classificateur) comme pour la classification. Cependant, OpenCL n'autorise pas d'appel aux libraires avancées depuis le GPU. Nous décidons donc de fournir une méthode reproduisant le comportement de `classify()` de Weka, pour rendre possible la classification par une thread sur le GPU. La solution développée consiste à (a) effectuer une copie depuis le CPU de chaque fonction de classification entrainée (on rappelle qu'au temps t_m, toutes les fonctions de classification entrainées jusque-là vont être nécessaires, celles-ci correspondant aux temps t_{m+1} jusqu'à t_{N-1}) vers la mémoire globale de la carte graphique, nécessitant l'utilisation de structures adéquates pour la représenter ;(b) implémenter une nouvelle fonction OpenCL `classify()` adaptée à cette nouvelle structure et permettant son appel en OpenCL. Plus précisément, une fois entrainée par l'appel à `buildClassifier()` de la librairie, cette fonction est sérialisée en structures de données compatibles OpenCL. Les structures telles que les tableaux multidimensionnels, les instances, et autres sont prohibées, au profit de tableaux unidimensionnels. Par ailleurs, quelques modifications de la librairie Weka originale ont été effectuées, comme l'accès à certains membres de classe rendus public, collectés pour la sérialisation des classificateurs. Les attributs des fonctions de classification, sont groupés et indexés dans des tables allouées en mémoire globale, afin que chaque thread GPU y accède ensuite indépendamment.

Nous intégrons les optimisations complémentaires suivantes. Les threads accèdent en parallèle et surtout de manière coalescente aux prix des actifs du panier à chaque pas de temps t, afin de réduire le coût de transaction en mémoire (**Figure 11**).

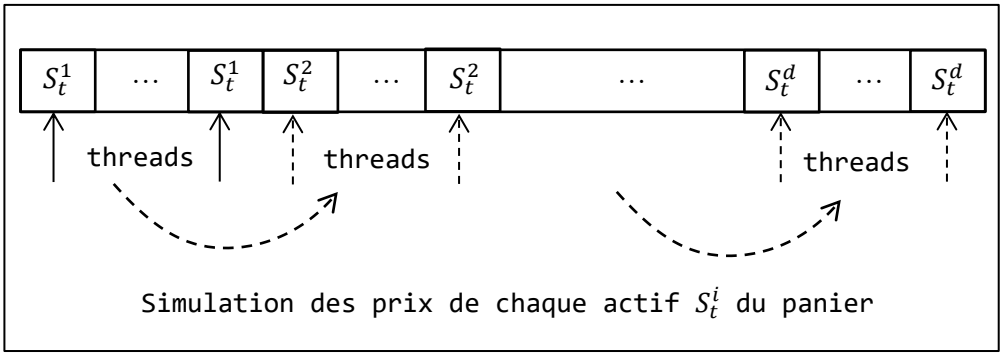

Figure 11 Accès coalescent en mémoire globale pour simuler les prix en t des actifs du panier. Les flèches en pointillé illustrent l'accès séquentiel aux différents actifs, par l'ensemble des threads.

Nous suivons les recommandations du guide de programmation NVIDIA [8] afin de tenter de profiter d'un'effet d'overlap: pour ce faire, nous réduisons de moitié (soit *nb_class/2*) les itérations de la boucle d'exécution des kernels, pour gérer l'entrelacement exécutions/transferts mémoire de deux estimations successives de variables d'entrainement au sein de chaque itération ; et allouons certaines données (par exemple, prix simulés des actifs sous-jacents et graines du générateur de nombres pseudo-aléatoires) en mémoire virtuelle non paginable de la mémoire globale (pinned memory). Cependant, la quantité de mémoire transférée entre CPU et GPU n'est pas suffisamment importante pour tirer un réel profit de l'overlap, et nous ne conserverons pas cette optimisation pour nos tests.

Toutes les réductions intermédiaires sont effectuées en parallèle en mémoire partagée, et les tests effectués mettent en avant leurs coûts négligeables en temps de calcul, même pour leur nombre important. La taille des workgroups étant fixée dynamiquement, nous décidons de ne pas dérouler les boucles de réduction (loop unrollings).

La mémoire globale cachée est sollicitée pour les données en lecture seule (volatilités, dividendes,..).

4.3. Calibration dynamique d'un kernel

Il est important d'assurer un taux élevé d'occupation du GPU, d'autant que de nombreuses raisons peuvent pénaliser les performances : coût d'accès à la mémoire off-chip non cachée, phénomène de warp divergence, utilisation des barrières de synchronisation... Maximiser ce taux d'occupation revient à maximiser le nombre de warps actifs du multiprocesseur (c.-à-d. dont les threads exécutent une instruction), ce taux étant le rapport entre le nombre de warps actifs et le nombre maximal de warps supportés par le multiprocesseur.

Le taux d'occupation dépend grandement du choix des paramètres de kernel. Une calibration optimale permet de réduire le temps d'exécution du kernel, et contribue donc à améliorer le temps d'exécution de notre application. D'autant plus lorsque celle-ci est destinée à s'exécuter sur un cluster hétérogène de GPUs, où chaque type de GPU aura des paramètres de kernel différents. Il nous faut donc les fixer dynamiquement, autrement dit, automatiquement, en fonction des caractéristiques de chaque carte. Une possibilité permettant de déterminer les paramètres optimaux serait d'exécuter l'application sur la carte cible avec différents paramètres. Mais cela s'avèrerait trop coûteux en temps et nous proposons donc une autre approche plus générique et portable pour déterminer les paramètres de chaque kernel dynamiquement.

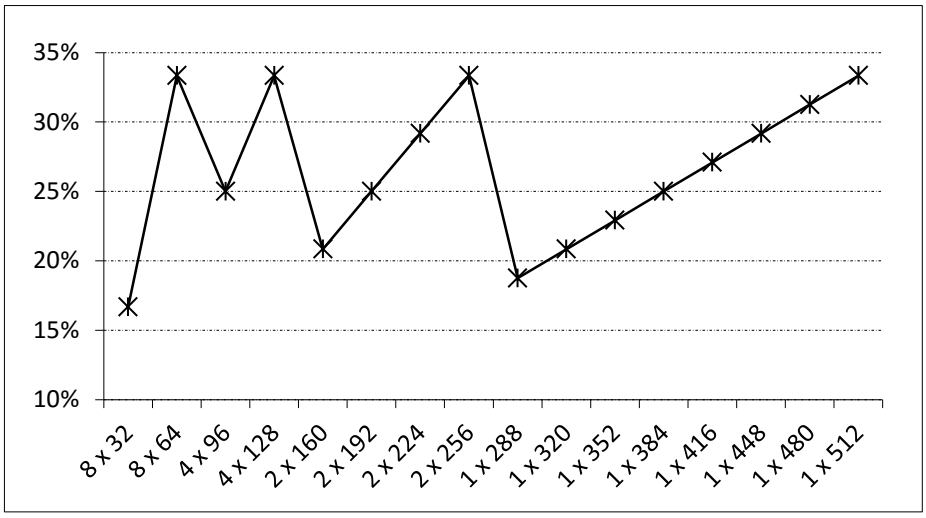

Figure 12 Taux d'occupation d'un multiprocesseur suivant les couples ($nombre\ de\ workgroup\ actifs\ par\ CU\ x\ taille\ d'un\ workgroup$). NVIDIA Quadro 600. Call américain de moyenne arithmétique. $S_{t_0}^i$=100, d= 5, K = 100, N= 50, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 50, nb_cont = 10^4, nb_MC= 10^6, SVM/linear PolyKernel.

Pour un kernel donné, la feuille Excel nommée « CUDA occupancy spreadsheet », permet de calculer le nombre théorique de workgroups actifs sur un multiprocesseur, pour en déduire le nombre de warp actifs. Cette feuille requiert en entrée : la « compute capability » de la carte (cc), la taille souhaitée des workgroups, le nombre de registres utilisés par thread et la quantité de mémoire partagée par workgroup. La feuille en déduit le taux d'occupation d'un multiprocesseur de la carte, en tenant compte de ses principales limitations : le nombre maximal de warps résidants, sa quantité de mémoire partagée et son nombre de registres. La **Figure 12** décrit pour notre kernel (celui basé sur SVM) l'évolution du taux d'occupation en ordonnées, selon, le nombre

de workgroups actifs par multiprocesseur (CU c.-à-d. Compute Unit) et leurs tailles, en abscisses. Un pic d'occupation est atteint pour chaque configuration de kernel offrant un nombre de workgroups actifs différent pour une taille maximale. Parmi les occupations estimées de notre programme, celle maximale atteignant 33.33% correspond à des workgroups de tailles : 64, 128, 256 et 512 respectivement. Cependant les workgroups de 64 threads permettent de maximiser leurs nombre (8 contre 4, 1 et 1 respectivement). L'occupation ne tient pas compte des particularités de notre application, mais nous savons que maximiser le nombre de workgroups actifs permet de réduire le nombre de simulations de MC parallélisées à chacun d'eux, et ainsi lisser leurs temps d'exécution.

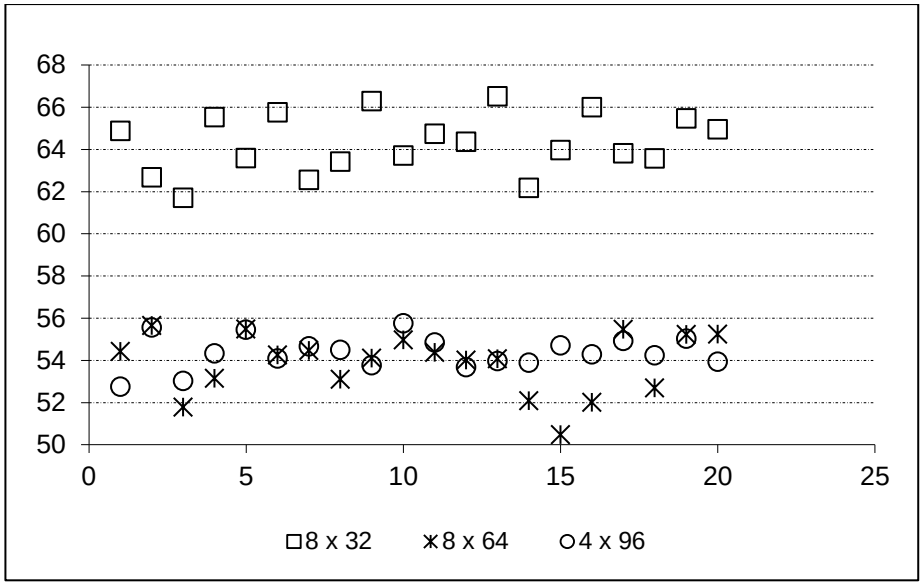

Figure 13 Temps cumulés en secondes, des estimations des variables d'entrainement, selon 20 exécutions pour chaque configuration. Les paramètres de pricing sont les même que ceux de la **Figure 12**.

Nous mettons en évidence dans la **Figure 13**, le lien existant entre taux d'occupation et temps d'exécution, en comparant les temps d'exécution selon trois configurations de kernel offrant des taux d'occupation de 16.67%, 25% et 33.33% (correspondant respectivement aux configurations 8x32, 4x96, 8x64). Le profiler NVIDIA ne révèle pas d'amélioration du taux d'occupation en travaillant avec plus de workgroups que le nombre actif.

Nous proposons donc un programme décrit par l'**Algorithme 5** qui détermine dynamiquement le plus grand nombre de workgroups actifs par CU et leur taille, en maximisant l'occupation théorique du CU, le tout sans exécution

préalable du kernel. Le programme fait appel aux fonctions Java que nous avons programmées `getNbActiveWgCU(…)` et `getOccupancyCU(…)` qui imitent la feuille Excel « CUDA occupancy spreadsheet ». Ces fonctions peuvent s'étendre aux cartes AMD selon la formule proposée en [97]. Durant son exécution, notre programme estime les taux d'occupation d'un multiprocesseur pour toutes les tailles autorisées de workgroup : de la taille d'un warp à la taille maximale d'un workgroup (multiple de la taille d'un warp). Chaque thread accède à la même quantité de mémoire partagée, lors des réductions parallèles. Par conséquent, la quantité de mémoire partagée d'un workgroup est proportionnelle à la taille d'un workgroup, et à la quantité de mémoire partagée par thread. Cette dernière `smemTH` est spécifique à chaque kernel et donc type de classificateur appelé, et n'est pas détectable durant l'exécution : nous devons donc la renseigner. Tandis que le nombre de registres requis par thread `regTH` est déterminé dynamiquement, en analysant le log de la compilation du kernel. Nous exécutons cet algorithme en préambule à l'exécution du pricing proprement dit, pour déduire les paramètres de kernel : le nombre total de threads du kernel et la taille des workgroups, c.-à-d. respectivement `nbActiveWgCU × nbCU × wgSize` et `wgSize`.

Algorithme 5 Calcul du nombre de `workgroups` actifs par CU et de leur taille, qui maximisent le taux d'occupation des CUs.

Entrée: *cc* de la carte
Entrée: *smemCU* quantité de mémoire partagée par multiprocesseur en bytes
Entrée: *smemTH* quantité de mémoire partagée utilisée par thread en byte
Entrée: *regTH* nombre de registres utilisés par thread
Sortie: {nombre de workgroups par CU, taille du workgroup}

```
1 .:warpSize=getWarpSize(cc)
2 .:wgMaxSize=getWorkgroupMaxSize(cc)
3 .:wgSizeTmp=0
4 .:maxOccupancyCU=0
5 .:Faire
6 .:  wgSizeTmp+=warpSize
7 .:  smemWgTmp=wgSizeTmp×smemTH
8 .:  nbActiveWgCUtmp=
      getNbActiveWgCU(cc,regTH,smemCU,smemWgTmp,wgSizeTmp)
9 .:  occupancyCU=getOccupancyCU (cc,nbActiveWgCUtmp,wgSizeTmp)
10 :  Si (occupancyCU>maxOccupancyCU)
11 :        wgSize=wgSizeTmp
12 :        nbActiveWgCU=nbActiveWgCUtmp
13 :        maxOccupancyCU=occupancyCU
14 :  Fin si
15 :Tant que (wgSizeTmp ≤wgMaxSize)
```

```
16 :Si (maxOccupancy==0)
17 :  Erreur «mémoire partagée ou registres insuffisants»
18 :Fin si
19 :Retourner {nbActiveWgCU, wgSize}
```

5. Tests et validation

Nous opposons dans ces tests, l'implémentation multi-CPU [5] à notre version mono-GPU. La version multi-CPU (pour laquelle nous disposons du code) est ainsi exécutée sur PACAGRID, un cloud privé de calcul opéré par l'INRIA. Nous valorisons une option américaine de dimension élevée, avec des paramètres de calcul complexes notamment de par le nombre d'actifs du panier (cf. légende **Figure 14**).

	64 cœurs à 2.3GHz provenant de AMD Opteron 2356 (1 Opteron = 4 cœurs)	Tesla M2075 112 workgroups × 64 threads
[phase 1] Estimation de la frontière d'exercice	7 h 01 min 30 s	3 h 36 min 30 s
[phase 2] Simulation de MC finales	0 h 53 min 12 s	0 h 00 min 28 s
Prix (~10^{-5})	0.70557 ± 0.00135	0.70664 ± 0.00135

Figure 14 Comparaison des temps d'exécution global des deux versions parallèles de l'algorithme de Picazo. Call américain de moyenne géométrique, $S_{t_0}^i$=100, d= 40, K = 100, N= 50, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 5000, nb_cont = 10^4, nb_MC= 2x10^6, AdaBoost/150 arbres binaires de décision.

Les prix sont reportés avec un intervalle de confiance de 95%. Notons que de nombreux paramètres de pricing peuvent biaiser cet intervalle : nb_class, nb_cont, N, paramètres de classification, et doivent aussi être considérés avec nb_MC.

Les tests mettent en exergue la capacité d'un GPU à résoudre ce problème difficilement parallélisable, en moins de temps qu'une version multi-CPU. La **[phase 2]** de l'**Algorithme 3** dédiée aux simulations de MC pour l'estimation du prix de l'option, convient parfaitement au parallélisme

important qu'offre le GPU (112 × 64 threads), comparé aux 64 cœurs CPU répartis du cluster (53 min avec le cluster CPU contre 28s avec le GPU).

Pour terminer, nous confrontons notre implémentation à celle proposée par Leitao Rodriguez et al [90], travaux à ce jour acceptés et non encore publiés, adoptant du maillage stochastique et une technique dite de *bundling* de prix sur lesquels sont appliquées des régressions. En effet, leurs résultats expérimentaux utilisant un GPU sont ceux qui à ce jour nous semblent être les plus remarquables d'autant qu'ils ciblent comme nous des paniers d'actifs de grande dimension. Entre autre, les auteurs valorisent sur GPU (NVIDIA Tesla K40m) un put bermudien sur 30 actifs, à 10 opportunités d'exercice, et obtiennent un prix de 0.93 ($\sim 10^{-2}$) en 11s. Il s'agit du cas le plus complexe qui est traité dans ce papier. Pour nous positionner en terme de performance, nous avons donc valorisé un put américain sur 10 pas temps pour correspondre au put bermudien (notre application actuellement ne nous permettant pas d'augmenter la discrétisation tout en fixant le nombre d'opportunités d'exercices, notre calcul pour le cas bermudien est donc moins précis et aussi moins couteux en temps) et obtenons un prix de 0.91 ($\sim 10^{-2}$) en 22s ; le tout sur un GPU d'ancienne génération (NVIDIA M2050), n'ayant pas accès à la même carte graphique récente.

Au-delà de l'économie faite tant sur l'infrastructure que sur la consommation électrique, fournir une telle puissance de calcul avec peu de ressources permet naturellement d'agréger d'autres ressources pour notre moteur de calcul, afin de réduire encore le temps de valorisation de l'option (ou afin d'adresser des options encore plus complexes). Travailler sur un cluster de GPUs/CPUs devient nécessaire pour bénéficier à la fois, du parallélisme important dont dispose l'architecture SIMT de chaque carte graphique, et d'un espace plus important de mémoire agrégée. C'est donc naturellement cette piste que nous suivrons dans le prochain chapitre.

6. Comparaison avec d'autres implémentations sur GPU

Comme nous l'avions déjà mentionné dans le Chapitre III, le pricing de l'option américaine est un problème majeur en finance, faisant l'objet de nombreux travaux pour réduire son temps de calcul. L'émergence de nouvelles architectures orientées HPC (GPU, FPGA), permet d'aborder ce problème d'un point de vue technique, en réadaptant les algorithmes de pricing parallèle pour ces unités de calcul. Nous revenons ici plus en détails sur certains de ces travaux, sélectionnés car représentatifs de tels efforts.

La valorisation de l'option américaine par la méthode des moindres carrés et sa parallélisation sur GPU en est un parfait exemple. Les auteurs en [98] proposent une version pour GPU. Durant la phase de régression, chaque thread pour un t donné, charge en mémoire les prix de l'actif et les payoffs, uniquement de ceux dans la monnaie (c.-à-d. $\Psi > 0$), la matrice des prix n'étant jamais chargée totalement.

Abbas-Turki et al. [83] proposent une implémentation GPU de l'algorithme de Longstaff et Schwarz pour valoriser une option américaine. Une des difficultés rencontrée lors la parallélisation, est l'étape dite de « régression » de l'algorithme, et plus précisément le calcul d'inverse de matrice, pour en déduire la matrice des coefficients de régression. Pour ce calcul et dans leur cas, l'utilisation d'un GPU n'est pas adaptée, sollicitant plutôt le CPU. Ce qui occasionne un surcout dû au transfert des données entre CPU et GPU, mais est compensé par la délégation d'autres calculs sur GPU. Le temps de calcul de l'étape de régression décroit presque linéairement avec le nombre de trajectoires simulées par machine. Toujours pour cette étape, les auteurs proposent d'exploiter un cluster de GPUs pour diminuer le nombre de trajectoires par machine et donc son temps de calcul. La première étape qui simule des browniens jusqu'à maturité, et l'étape finale de simulation des trajectoires, se parallélisent aisément sur GPU. Les auteurs valorisent ainsi sur GPU, une option américaine sur 4 actifs en un peu plus de 7 secondes.

Dans [99] Zhang et al présentent une version hybride CPU-GPU et parallèle du modèle binomial pour valoriser l'option américaine. Le modèle binomial permet de simuler toutes les trajectoires possibles d'un actif sous-jacent, connaissant son prix initial S_0, en considérant uniquement un facteur de diminution et d'augmentation de prix. La répartition des tâches par processeur (p) est représentée en **Figure 15**. L'algorithme exécute séquentiellement des blocs de calculs, depuis les feuilles de l'arbre vers sa racine. L'ensemble des blocs de calcul est distribué équitablement sur l'ensemble des processeurs. Chaque sous bloc (triangle) est composé d'une partie de calculs [phase 1] exécutée en premier par chaque processeur, et d'une autre partie [phase 2] exécutée après, qui dépend d'autres résultats. L'algorithme ajuste dynamiquement l'assignation des sous blocs aux processeurs, car le degré de parallélisme diminue en progressant vers la racine du modèle. L'exécution de l'algorithme sur un CPU double cœur permet dans certains cas, d'obtenir un speedup super linéaire contre une version optimisée pour CPU. Chaque processeur sauvegarde ses résultats intermédiaires dans un buffer, dont la taille dépend du nombre maximal d'étapes de calculs de chaque bloc (L).

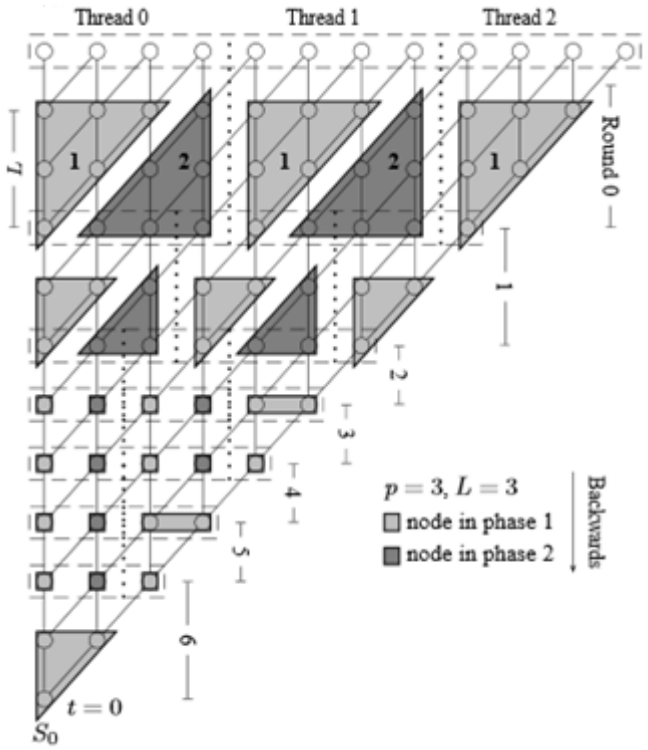

Figure 15 [99] Distribution de l'algorithme basé sur le modèle binomial.

Ganesan et al [100] utilisent aussi le modèle binomial pour valoriser une option américaine sur un GPU, et cassent la dépendance entre les niveaux successifs de l'arbre, lors du calcul des prix pour chaque nœud. Ils établissent la relation entre deux prix de l'option à différents instants, notant p_u (resp.p_d) la probabilité que le prix de l'actif augmente (resp. diminue)

$$F_{t-2\Delta t}(i) = e^{-r\Delta t}\left(p_u F_{t-\Delta t}(i) + (1-p_u)F_{t-\Delta t}(i+1)\right)$$
$$F_{t-2\Delta t}(i) = e^{-2r\Delta t}\left({p_u}^2 F_t(i) + 2p_u p_d F_t(i+1) + {p_d}^2 F_t(i+2)\right)$$

En raisonnant similairement en $t - 3\Delta t$

$$F_{t-3\Delta t}(i) = e^{-3r\Delta t}(c_0 F_t(i) + c_1 F_t(i+1) + c_2 F_t(i+2) + c_3 F_t(i+3))$$

Plus généralement

$$F_{t-N\Delta t}(i) = e^{-Nr\Delta t}\sum_{j=0}^{N} c_j F_t(i+j)$$

Les auteurs obtiennent ainsi la forme itérative de calcul de ses coefficients

$$c'(i) = p_u c(i) + p_d c(i+1)$$

ayant pour critère d'arrêt $c'(i) = c(i)$. L'arbre de calcul est ainsi découpé en p bandes de largeur T/p comme l'illustre la **Figure 16**, p correspondant au nombre de partitions, chacune attribuée à un multiprocesseur. Ceci permet de calculer en parallèle les dépendances successives entre les limites des partitions, et ainsi propager les prix aux limites par bonds de $T/p\ \Delta t$ jusqu'au temps initial. Cette stratégie permettrait en théorie d'améliorer de 15 fois les performances d'une implémentation parallèle comparable sur 1000 pas de temps.

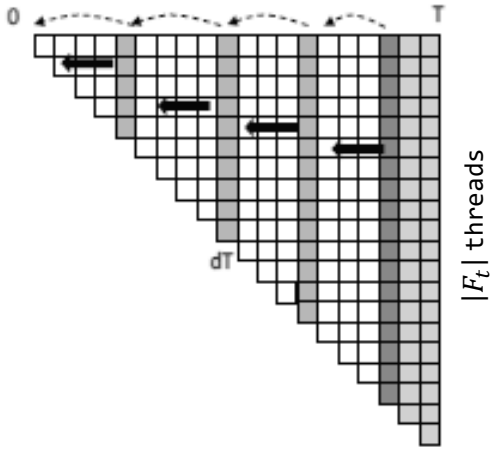

Figure 16 [100] La partie la plus à droite calcule les prix de l'option, pendant que les autres partitions calculent les dépendances entre les différents prix.

L'une des principales difficultés lors de l'adaptation d'un algorithme sur GPU, est de respecter le paradigme SIMT de son architecture. Des récents travaux traitent des problèmes de « warp divergence » des applications GPU [101] [102] dont nous aurions pu nous inspirer. Ceux-ci ne ciblent pas spécifiquement la valorisation d'option bien évidemment.

Nous aurions effectivement pu penser à jumeler une implémentation naïve (**Figure 7 - 1**) avec une stratégie plus générique, telle celle de [101] où un micro scheduler distribue de nouvelles tâches aux threads terminant plus tôt. Cependant, l'adaptation GPU de l'algorithme de Picazo, déjà couteuse en mémoire, ne permet pas de réserver plus de mémoire à accès rapide pour gérer des pools de tâches. De plus cette stratégie est peu adaptée lors de l'exécution de multiples kernels simulant peu de trajectoires, ce qui est notre cas lors de l'estimation des instances d'entrainement.

Thread 1	Thread 2	Thread 3	Instr. Count
(1,T)	-	(3,T)	100
-	(2,N)	-	100
-	(5,T)	(6,T)	100
(4,N)	-	-	100
(7,T)	-	-	100
-	(8,N)	(9,N)	100
Total			600

(a) Original execution.

Thread 1	Thread 2	Thread 3	Chosen Br. Dir.	Instr. Count
(1,T)	-	(3,T)	T	100
(4,N)	(2,N)	-	N	100
(7,T)	(5,T)	(6,T)	T	100
-	(8,N)	(9,N)	N	100
Total				400

(b) Iteration delaying with majority-vote strategy.

Figure 17 [102] Cas d'un warp de 3 threads itérant trois fois sur un « if ». Chaque instruction requiert 100 instructions FMA (fused multiply-add). Pour un couple (x,y) donné, x définit le numéro d'itération, y vaut T si l'instruction du « if » est exécutée et N sinon.

En [102] les auteurs proposent des solutions de limitation de la warp divergence qui se situent à plus haut niveau. L'une d'elles porte sur les instructions conditionnelles au sein de boucles : pour chaque itération, seules les exécutions empruntant le même chemin sont traitées par un warp, retardant les autres **Figure 17**. Comme amélioration à nos travaux, nous pourrions l'intégrer lorsque nous testons si une trajectoire atteint la maturité, effectuant la classification le cas échéant.

7. Conclusion

Nos travaux présentés dans ce chapitre ont révélé la nécessité de reconsidérer l'algorithme parallèle d'un programme ciblant un cluster de CPUs, lors de son adaptation pour profiter de la source de parallélisme provenant d'un seul GPU.

Lors du travail d'implémentation lorsque CPU et GPU collaborent à la résolution du problème, il s'est avéré possible de travailler avec certaines librairies avancées côté CPU (uniquement) ; néanmoins, pour en tirer profit côté GPU, il faut reproduire/imiter les fonctions voulues. Ce qui nécessite

principalement de choisir des structures de données simples (comme des tableaux unidimensionnels par exemple) afin de transférer les données en mémoire entre CPU et GPU.

Par ailleurs, nous avons proposé une stratégie dynamique de calibration des paramètres de kernel, pour maximiser le taux d'occupation théorique, et ce, quel que soit le type de la carte graphique. Nous l'avons évaluée sur des cartes NVIDIA, mais une adaptation pour les cartes AMD est facilement faisable. Notre solution générique devrait s'appliquer aisément à d'autres problèmes.

L'objectif étant de réduire encore plus le temps de valorisation d'une option américaine, nous voulons tirer parti non pas d'un GPU, mais de l'ensemble des unités de calcul que peut offrir un cluster (CPUs et GPUs). Ainsi, dans le prochain chapitre, nous devrons reconsidérer l'implémentation de l'algorithme de Picazo pour introduire un second niveau de parallélisme et proposer une stratégie d'équilibrage des tâches, sur les différents nœuds de calcul du cluster.

Chapitre V. Valorisation d'option américaine sur cluster hétérogène de GPUs/CPUs

1. Introduction

Nous présentons ici notre adaptation de l'algorithme de Picazo pour cluster hétérogène de CPUs et GPUs. Dans un premier temps, nous décrivons comment cohabitent les deux niveaux de parallélisme : l'orchestration de la distribution des calculs entre les nœuds du cluster, et le parallélisme des calculs au sein de chaque CPU et GPU des nœuds. Nous constaterons que l'entrainement centralisé des classificateurs (Boosting, SVM) entrave la scalabilité de l'application, et ne permet pas de réduire davantage le temps d'exécution de l'application. Nous expliquons comment y remédier par les forêts aléatoires ; parfaitement adaptées à notre environnement distribué, et disponibles via la libraire Java Weka.

Pour exploiter un cluster hétérogène, nous exécutons simultanément sur l'ensemble des unités de calcul, notre algorithme de calibration de kernel précédemment détaillé, qui s'étend dorénavant aux CPUs. Nous étudions différentes stratégies d'équilibrage des phases de calculs pour cluster hétérogène, et leurs impacts sur le temps global d'exécution. Dans les différentes parties, nous mettons en évidence les avantages/inconvénients de nos choix par des tests.

2. Adaptation multi-CPU-GPU de l'algorithme de Picazo

L'adaptation multi-CPU-GPU se situe à deux niveaux. L'ensemble des points à classifier est distribué sur plusieurs nœuds CPU. Charge à chaque nœud d'exploiter lui-même son GPU (ou CPU multi-cœur), pour profiter du parallélisme et simuler les trajectoires nécessaires à l'obtention de chaque point. Nous détaillons deux entrainements possibles de classificateur, centralisé et distribué.

2.1. Entrainements centralisés et séquentiels des classificateurs

Notre adaptation de l'algorithme de Picazo introduit donc deux niveaux de parallélisme comme l'illustre la **Figure 18**. Le premier niveau de parallélisme suit une stratégie maitre-esclave, via le concept d'objets actifs fourni par la libraire ProActive. Ainsi, durant la phase dite de détection décrite en **[partie I]**, l'application alloue dynamiquement un objet actif, nommé *detector*, sur chaque

nœud, qui détecte les ressources de son CPU : nombre de cœurs CPU du nœud et le nombre de GPUs associés.

Les objets actifs maitre-esclaves (`merger` et `worker`) sont déployés ensuite dynamiquement **[partie II]**. Le `merger`, dont le rôle est de collecter les résultats des calculs intermédiaires, est alloué sur le nœud disposant le moins de GPUs. A contrario, les `workers` exploitent les GPUs/CPUs du cluster pour leur sous-traiter les exécutions des phases de calcul intensif (kernels), qui constitue notre second niveau de parallélisme. Ainsi, nous instancions un objet actif `worker` par GPU/CPU, en excluant le CPU du `merger`. Plusieurs `workers` peuvent cohabiter sur le même nœud lorsque plusieurs GPUs y résident, ce qui n'impacte pas les performances, car les tâches sont GPU-intensives.

La **[partie III]**, résumée à gauche de la **Figure 18**, détaille l'orchestration du calcul des instances d'entrainement de chaque classificateur. Pour chacun d'eux, l'application procède ainsi: en **[étape 1]** chaque `worker` estime une partie des instances d'entrainement, en faisant exécuter sur le GPU/CPU qu'il contrôle les nb_cont simulations de MC de chaque valeur de continuation et en récupère les résultats; en [**étape 2]** le `merger` qui a collecté l'ensemble des instances d'entrainement de chacun des `workers`, entraine séquentiellement un nouveau classificateur par l'appel `buildClassifier()` de la librairie Weka. Le classificateur est ensuite broadcasté sur l'ensemble des `workers`, pour être utilisé par chaque kernel lors des estimations des valeurs de continuation du prochain classificateur. Nous avons donc un entrainement de chaque classificateur, qui est exécuté séquentiellement sur le `merger` à chaque temps discret de la [phase 1] de l'algorithme de Picazo.

Avant de débuter la **[partie IV]**, tous les classificateurs ayant été transférés aux `workers` (c.-à-d. alloués dans la mémoire du GPU/CPU associé), ces derniers peuvent ainsi exécuter chacun une partie des simulations de MC finales. Enfin, le `merger` collecte les résultats intermédiaires pour déduire le prix de l'option, la variance, et l'intervalle de confiance.

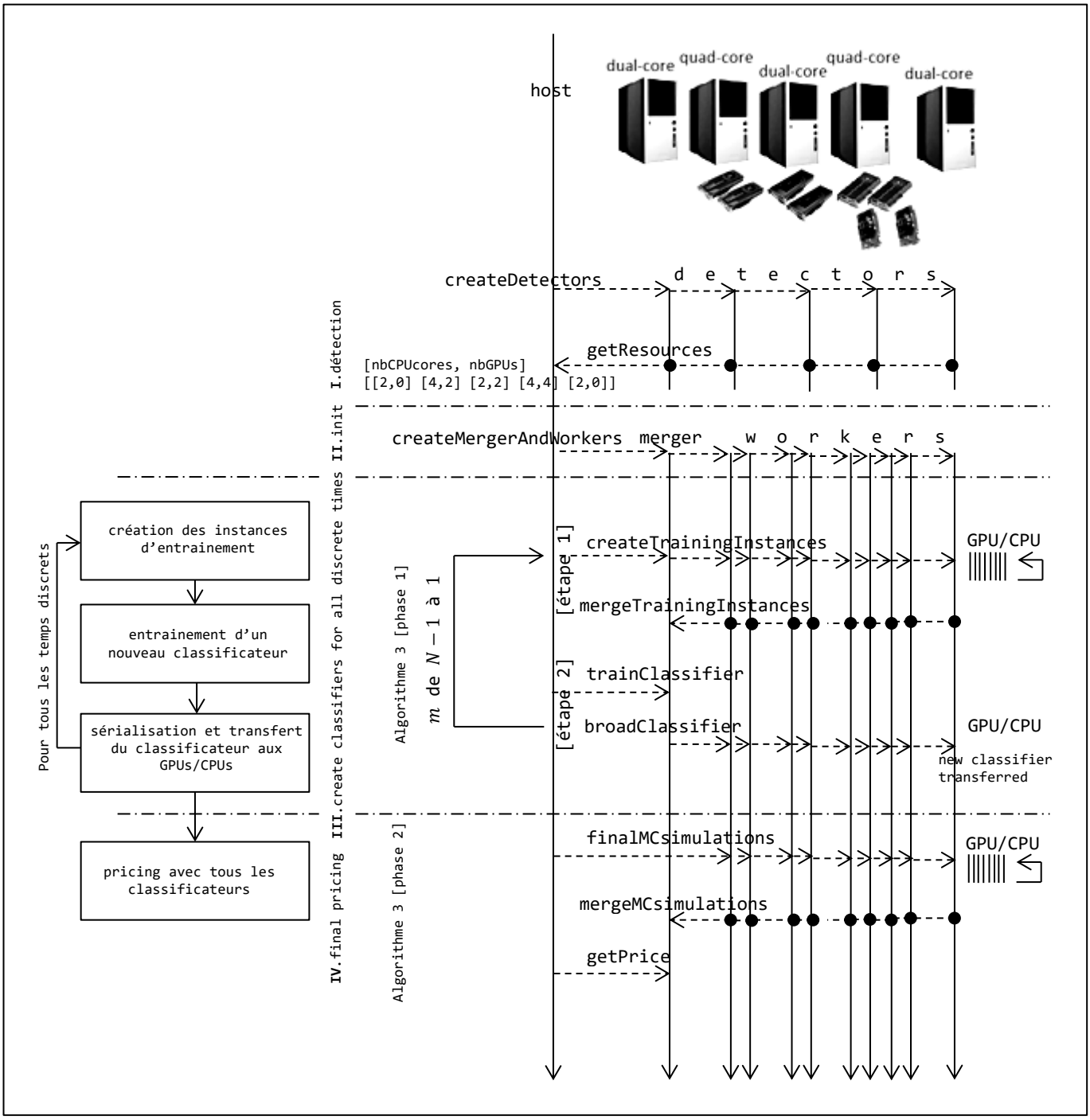

Figure 18 Adaptation multi-CPU-GPU de l'algorithme de Picazo : (Gauche) vue d'ensemble, (Droite) vue détaillée.

2.2. Entrainements parallélisés des classificateurs

Pour espérer paralléliser sans limite l'algorithme de Picazo, nous distribuons également les phases d'entrainement encore séquentielles, et de ce fait, changeons de méthode de classification. En adoptant les forêts aléatoires, nous distribuons la phase de construction sur plusieurs nœuds.

Dans un premier temps, nous distribuons la [partie III] [étape 2] de la **Figure 18**, puis dans un second temps, nous représentons sous OpenCL les arbres binaires non complets d'une forêt.

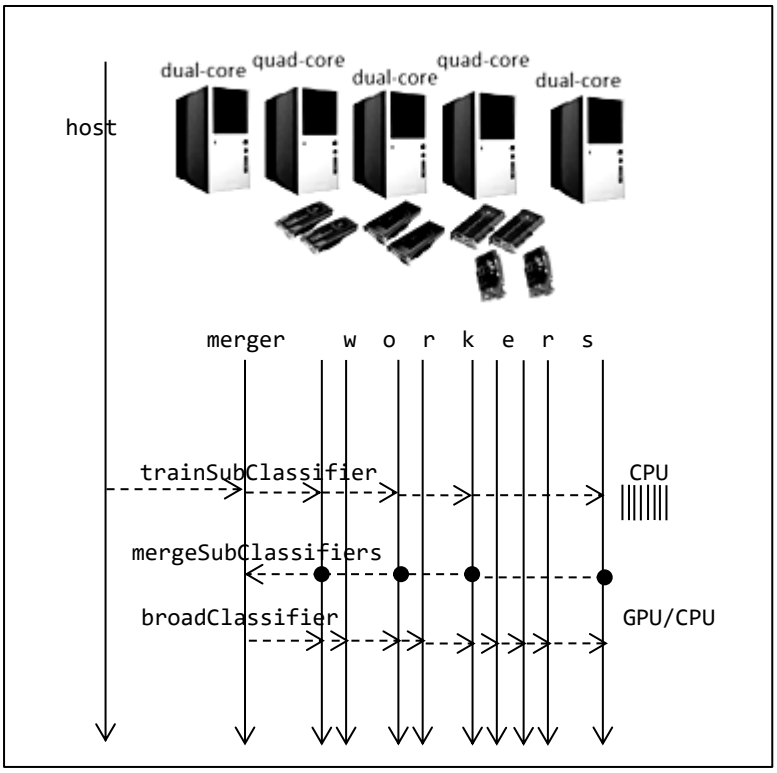

Figure 19 Distribution de l'entrainement d'une forêt aléatoire sur les cœurs CPU du cluster. Cette étape remplace la [partie III] [étape 2] de la **Figure 18.**

Nous décidons de préserver le comportement de la librairie Weka : nous entrainons en parallèle sur chaque *worker* des sous-forêts, par l'appel à `buildClassifier()`, comme nous l'aurions fait pour une unique forêt sur le `merger`. La librairie Weka est légèrement modifiée afin d'obtenir sur le `merger`, une forêt résultant de la fusion des sous-forêts, parfaitement identique à la forêt initiale, c.-à-d. produisant les mêmes résultats de classification. Ceci nous permet donc d'entrainer en parallèle des sous-forêts sur chaque nœud du cluster, et manipuler la forêt résultant de leur agrégation comme l'originale

(**Figure 19**). Comme optimisation complémentaire, nous exploitons les dernières versions de Weka permettant de fixer le degré de parallélisation de l'appel à `buildClassifier()` pour chaque `worker` : nous fixons ce paramètre au nombre total de cœurs du CPU sur lequel s'exécute un `worker` n'entrainant qu'une sous-forêt à la fois.

Nous fixons le degré de parallélisation de l'entrainement de chaque sous-forêt au nombre de cœurs du CPU. Cependant, les CPUs étant hétérogènes, nous équilibrons aussi la quantité de sous-forêts à entrainer, en attribuant à chaque `worker` w, un sous-ensemble $nbTrees_w$ du nombre total $nbTrees_{classifier}$ d'arbres d'une forêt aléatoire, tel que

$$nbTrees_w = \frac{nbCPUcores_w}{\sum_{all\ CPUs\ P} nbCPUcores_P} \times nbTrees_{classifier}$$

Dans les tests expérimentaux sollicitant les forêts aléatoires, nous désactivons cette dernière optimisation pour mettre davantage en évidence le bénéfice de distribuer l'entrainement de classificateur: en effet même sans cette optimisation, on verra que l'usage des forêts aléatoires permet à notre algorithme de passer à l'échelle.

Au même titre qu'un classificateur entrainé avec AdaBoost ou SVM, une forêt aléatoire est sérialisée par chaque `worker` puis transférée vers la mémoire de son unité de calcul supportant OpenCL. Les forêts aléatoires sont constituées d'arbres non complets, dont la représentation en sous tableaux est couteuse en mémoire pour des arbres de grande taille. De plus, seule une solution expérimentale est proposée en JOCL pour travailler avec une structure arborescente en OpenCL, transférée depuis le CPU. A notre connaissance, la littérature ne nous fournissait donc pas de solution adéquate pour travailler en OpenCL sur une structure arborescente. Nous avons donc défini notre propre représentation d'arbres, nous permettant même de les compresser avant de les transférer en mémoire GPU et qu'ils y soient parcourus par le kernel. Une fois la forêt aléatoire transmise par le `merger`, chaque `worker` parcourt les informations des nœuds de tous les arbres de la forêt et les enregistre dans des tableaux spécifiques pour chaque type d'information (indices d'attributs, splitvalue des noeuds, valeurs de distribution). Par exemple, l'ensemble des splitvalues des nœuds de l'ensemble des arbres d'une forêt, résident en contigu dans un tableau dédié. Pour dissocier les informations d'un arbre à l'autre, nous enregistrons dans un tableau spécifique les indices de position des racines de chaque arbre. Lors de la classification en OpenCL, pour imiter le parcours de chaque arbre de la forêt, nous parcourons un tableau d'indices des positions de fils gauche et de fils droit. Enfin, lors de l'exécution d'un kernel, les threads

sollicitent l'ensemble des classificateurs déjà entrainés. Ceux-ci sont donc enregistrés dans le même tableau par type de donnée : par exemple `attributesClassifiers` comprend l'ensemble des *attributesClassifier* de tous les classificateurs, nous contraignant à utiliser un indice de position supplémentaire pour sélectionner les attributs du classificateur correspondant au pas de temps voulu.

2.3. Tests sur cluster homogène de GPUs

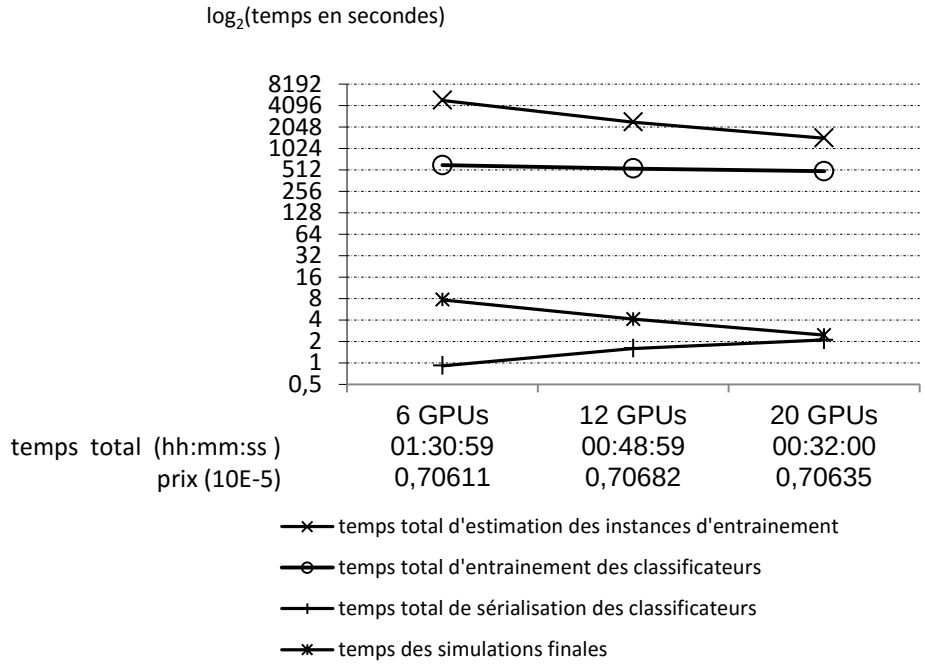

Figure 20 Comparaison des temps d'exécution des différentes étapes de pricing selon le nombre de *workers* avec AdaBoost, 150 arbres binaires de décision. Les paramètres de pricing sont les mêmes qu'en **Figure 14**.

La **Figure 20** illustre les temps d'exécution de [partie III] et [partie IV] (**Figure 18**) sur un cluster homogène (Adonis de Grid'5000), pour une option américaine de grande dimension (panier de 40 actifs). [partie I] et [partie II] ne sont pas spécifiées sur cette figure, de par leurs temps d'exécution faible et donc négligeable, et la possibilité de réutiliser les objets actifs déployés pour plusieurs exécutions successives. Les temps de création des instances d'entrainement et de la phase finale de pricing incluent les opérations de broadcast/merge depuis/vers le `merger`.

Nous franchissons le seuil d'une heure en exploitant un cluster de 12 GPUs (de type Tesla S1070). Nos tests révèlent une dépendance linéaire entre le nombre de `workers` et les temps de calcul de chaque phase, mais démontrent qu'augmenter le nombre de `workers` complexifie les opérations de brodcast/merge. De plus, la réduction des temps de calcul de [phase III] et [phase IV] en augmentant le nombre de `workers`, fait tendre le temps total de pricing vers le temps (forcément constant quel que soit le nombre de `workers`) des entrainements séquentiels (sur le merger) des classificateurs (~500 s).

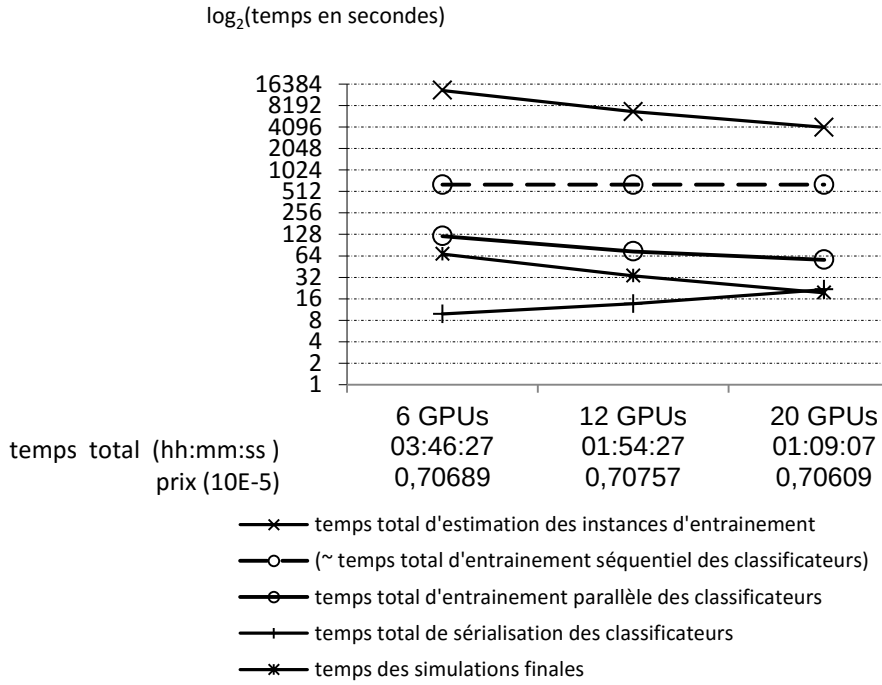

Figure 21 Comparaison des temps d'exécution des différentes étapes de pricing selon le nombre de `workers` des forêts aléatoires de 150 arbres distribués, sans limite de hauteur. Les paramètres de pricing sont les mêmes qu'en **Figure 14**.

En observant la **Figure 21**, nous constatons les mêmes prix qu'avec AdaBoost via les forêts aléatoires. La création des instances d'entrainement (plus de 1h avec 20 GPUs) requiert plus de temps qu'avec AdaBoost (~30min avec 20 GPUs), dû au coût de classification. En effet, il est plus coûteux pour une thread GPU/CPU de balayer les 150 arbres d'une forêt aléatoire, de hauteur non limitée, que de parcourir les 150 arbres à un seul niveau d'un classificateur AdaBoost. En revanche, nous tirons bénéfice des CPUs distribués durant l'entrainement distribué des classificateurs, comme le montre la comparaison des tracés plein et pointillé avec cercles.

3. Exploiter un cluster hybride hétérogène

3.1. Calibrations dynamiques et parallèles de kernel

OpenCL tire bénéfice aussi bien de l'architecture parallèle des GPUs (SIMT) que des CPUs (SIMD), sans modification de code (Chapitre II.4.2). Il devient alors évident d'exploiter cette spécificité pour utiliser toutes les unités de calcul qu'offre un cluster. Notre stratégie dynamique de calibration des paramètres d'un kernel, qui ne nécessite aucune exécution de l'application, est parfaitement adaptée pour une calibration simultanée sur plusieurs nœuds éventuellement hétérogènes d'un cluster. Ainsi, avant l'exécution de la [phase 1] de l'**Algorithme 3**, chaque `worker` calibre les paramètres de son GPU/CPU associé.

Nous renvoyons le lecteur en Chapitre IV.4.3 pour la calibration pour GPU. Les recommandations concernant la calibration pour CPU [19] soulignent l'importance d'adapter la taille des workgroups selon le nombre de barrières de synchronisation. En effet, synchroniser explicitement les threads d'un workgroup, entraine une commutation de contexte, consistant en particulier à sauvegarder/copier l'ensemble des données allouées dans la mémoire privée de chaque thread. Il est donc conseillé de réduire les tailles des workgroups si les barrières de synchronisation sont nombreuses, ce qui est notre cas. Selon les recommandations d'Intel, nous fixons à 32 la taille des workgroups pour chacun de nos trois différents kernels (un pour chaque méthode de classification), ceux-ci présentant de nombreuses barrières. Les tests en **Figure 22** exécutés sur un CPU Intel Xeon E5440 confirment notre choix. Par ailleurs, nous fixons le nombre de workgroups à 8, correspondant au nombre de cœurs reconnus par l'appel à `clGetDeviceInfo(CL_DEVICE_MAX_COMPUTE_UNITS)`.

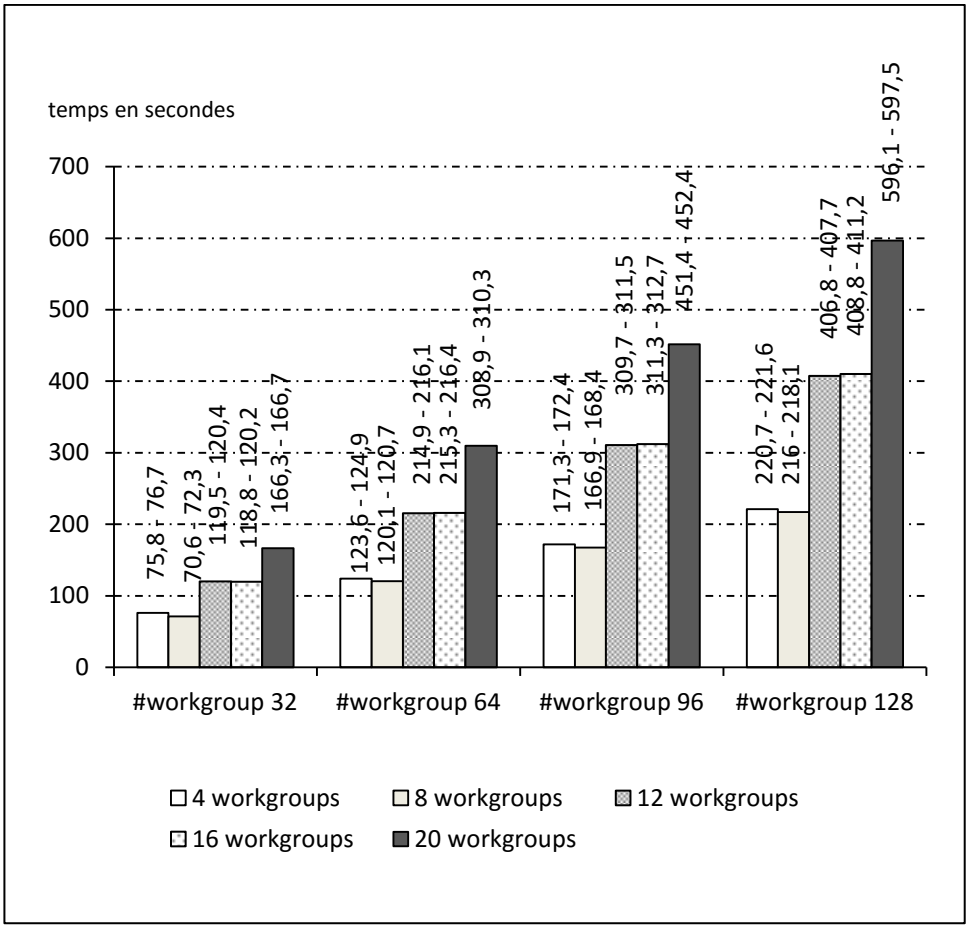

Figure 22 Comparaison des temps d'exécution de pricing d'une option américaine, selon les nombres et tailles des workgroups. Les intervalles de temps constituent les bornes minimales à 10-1 encadrant 5 exécutions. Call américain de moyenne géométrique, $S_{t_0}^i$=110, d= 5, K = 100, N= 10, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision.

3.2. Répartition du calcul des instances d'entrainement

Nous comparons dans cette partie, différentes stratégies de distribution des calculs de création des instances d'entrainement (**Algorithme 3** [phase 1] [étape 1]), parmi les `workers` d'un cluster hétérogène. Les trois premières qui vont être exposées, répartissent les calculs avant de démarrer le pricing. La dernière méthode distribue les calculs par paquet, durant le pricing.

Une première manière de procéder est de ne pas tenir compte de l'hétérogénéité du cluster, et de distribuer uniformément le nombre d'instances d'entrainements sur l'ensemble des `workers`, au démarrage du pricing. Le `worker` w indexé $rank_w$, calcule nb_class_w instances suivant

$$nb\_class_w = \frac{nb\_class}{nbWorkers} + 1_{rank_w < nb\_class \,\%\, nbWorkers}$$

Une meilleure stratégie qualifiée néanmoins de naïve, repose au démarrage sur la distribution des instances d'entrainement, proportionnellement aux nombres calibrés de threads pour chaque unité de calcul. En notant $nbThreads_w$ ce nombre pour l'unité de calcul contrôlée par le `worker` w, on a

$$nb\_class_w = \frac{nbThreads_w}{\sum_{all\ workers\ P} nbThreads_P} \times nb\_class$$

La stratégie dite « avancée », estime simultanément pour l'ensemble des GPUs/CPUs du cluster, le temps de calcul théorique d'une valeur de continuation, dont dépend majoritairement le temps de calcul d'une instance d'entrainement. Ainsi, pour un classificateur H_t tel que $t = 1..(T-1)$, on obtient

$$durationKernel_w(t) = durationKernelInit_w + \frac{nbDtKernel(t)}{throughputDt_w}$$

nous permettant d'en déduire les nb_class de w pour les différents t

$$nb\_class_w(t) = \frac{\frac{1}{durationKernel_w(t)}}{\sum_{all\ workers\ P} \frac{1}{durationKernel_P(t)}} \times nb\_class$$

Cette méthode requiert la mesure préalable du temps requis, pour simuler un pas de temps avec classification, qui sert donc d'unité de mesure. Dans ce but, nous calculons H_{T-1}. Ensuite, nous estimons $durationKernelInit_w$ ainsi que $throughputDt_w$, correspondant respectivement au temps d'exécution moyen d'un kernel hors simulations de MC, et au nombre de pas de temps par seconde que l'unité de calcul parvient à simuler. Notons que nous employons le même procédé pour répartir le nombre nb_MC de simulations finales, bien que cette étape soit relativement courte.

Il est difficile de prédire de manière exacte le nombre moyen de pas de temps exécutés par kernel pour simuler les instances d'entrainement de H_t, noté $nbDtKernel(t)$ avec $t \in [1, T-2]$. En effet, le temps d'arrêt de chaque trajectoire est aléatoire. Nous proposons donc de le considérer comme un processus stochastique suivant une loi uniforme

$$\tau_{[t,T]} \sim \mathcal{U}([t,T])$$

Nous estimons donc le nombre de pas de temps simulés par kernel (donc pour produire une instance de classification) à

$$\begin{aligned} nbDtKernel(t) &= nb\_cont \times \left(\mathbb{E}\left(\tau_{[t,T]}\right) - t\right) \\ &= nb\_cont \times \left(\frac{T+t}{2} - t\right) \\ &= nb\_cont \times \left(\frac{T-t}{2}\right) \end{aligned}$$

L'intérêt de comparer les débits $throughputDt_w$ des unités de calcul, permet de ne pas avoir à quantifier les capacités hardwares propres à chaque type d'unité : fréquence, nombre de cœurs, largeur registres XMM du CPU,… Cette approche plus générale convient aussi tant aux GPUs qu'aux CPUs.

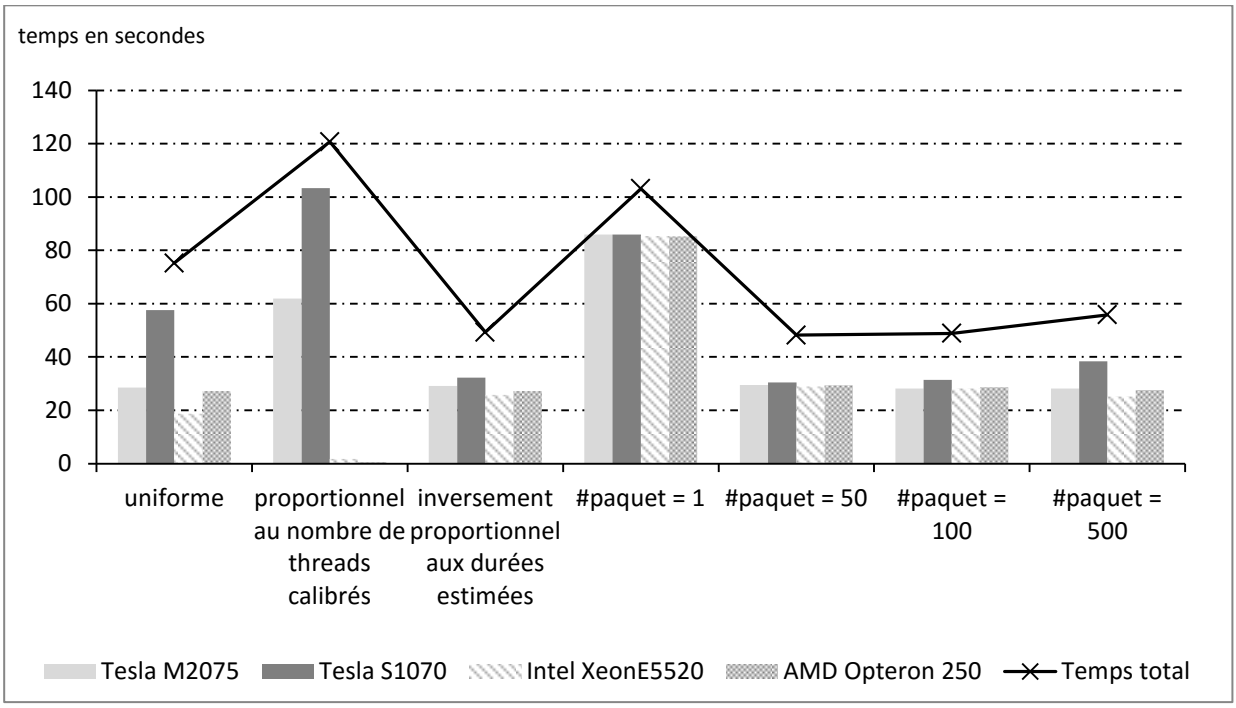

Figure 23 Temps cumulés des créations des instances d'entrainement, pour chaque GPU/CPU, selon différentes stratégies de distribution. Call américain de moyenne géométrique, $S_{t_0}^i$=110, d= 7, K = 100, N= 10, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision.

Notre dernière stratégie possible de distribution des instances d'entrainement s'exécute durant le pricing. Dès lors qu'un `worker` termine l'estimation d'un paquet d'instances restantes, celui-ci requête le `merger` pour en simuler un nouveau. La taille des paquets fixée par l'utilisateur, doit répondre au compromis suivant, pour assurer un temps total d'exécution bas:

une taille trop petite engendre un surcoût des communications entre les `workers` et le `merger`, alors qu'une taille trop grande surchargera les unités de calcul les moins performantes, que les unités les plus rapides attendront.

L'inconvénient de cette stratégie de distribution par paquet, est la nécessité de calibrer la taille des paquets en fonction de la complexité du pricing. Ceci ne peut être effectué qu'empiriquement par l'utilisateur (avec par exemple, des paquets de taille 1, 50, 100, ou 500 comme illustré sur les **Figure 23** et **Figure 24**). Au contraire, la stratégie avancée, c.-à-d. inversement proportionnel aux durées estimées, calibre automatiquement, bien que fondée sur une prédiction, la taille de chaque paquet à allouer à chaque unité de calcul.

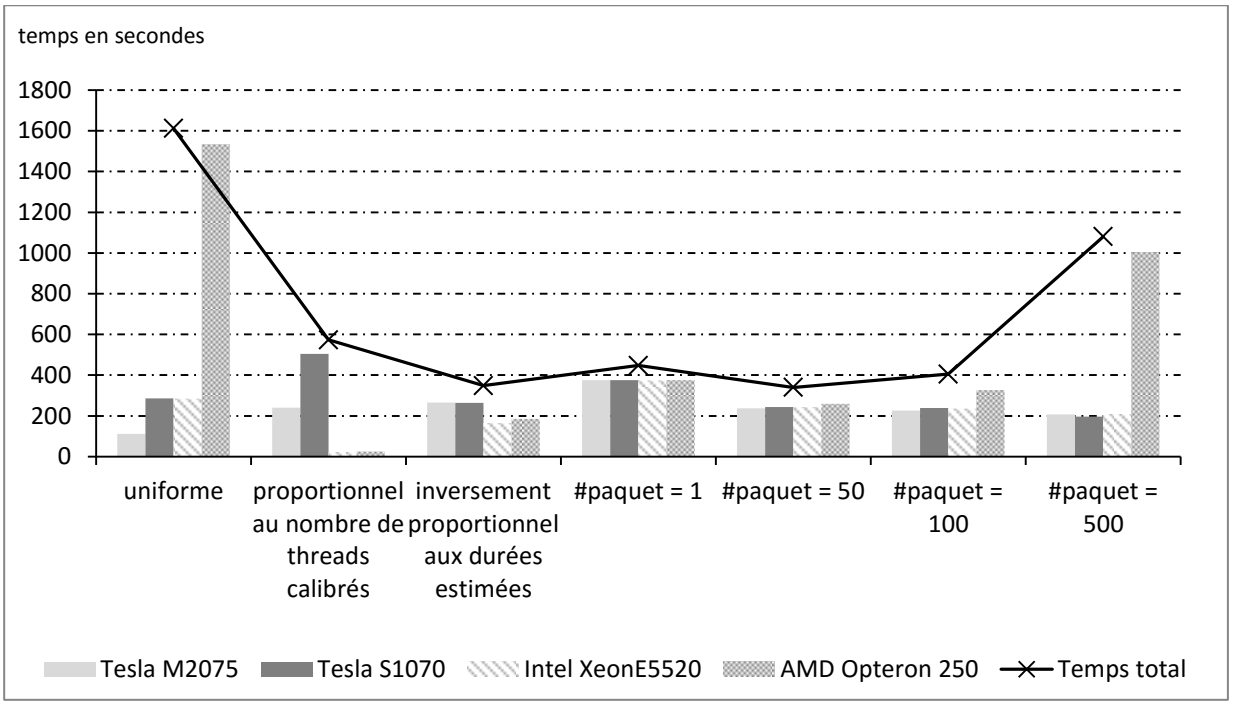

Figure 24 Temps cumulés des créations des instances d'entrainement, pour chaque GPU/CPU, selon différentes stratégies de distribution. Call américain de moyenne géométrique, $S_{t_0}^i$=110, d= 20, K = 100, N= 20, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 10^4, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision.

3.3. Tests sur cluster hétérogène hybride

Maintenant que la stratégie de répartition est au point et s'applique au cas d'unités hétérogènes, nous réalisons ici des expérimentations avec le plus grand

nombre possible de GPUs mobilisables sur Grid'5000, complétés par des CPUs multi-cœur, pour un total ici de 18 unités de calcul.

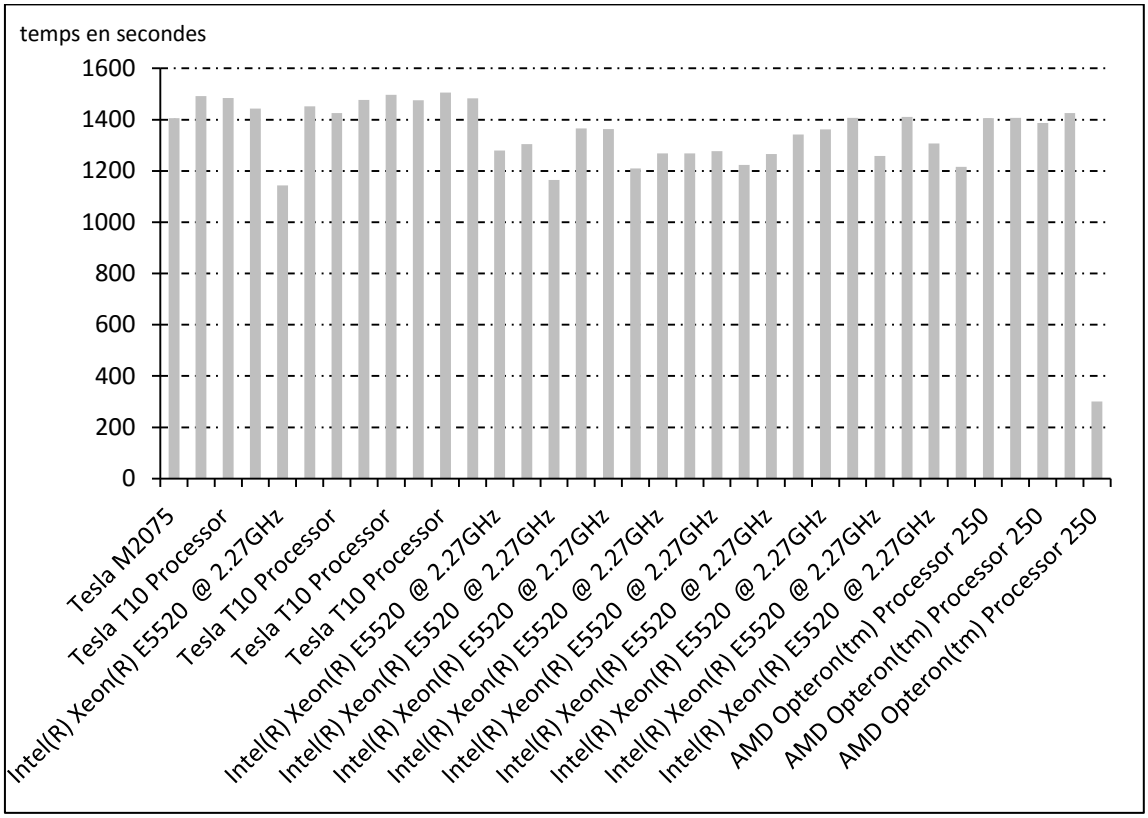

Figure 25 Temps cumulés des créations des instances d'entrainement pour chacun des 18 GPUs/CPUs du cluster. Les paramètres de pricing sont les mêmes qu'en **Figure 14**.

Nous obtenons un prix à 10-5 près de 0.7067 ± 0.00135 (CI à 95%) pour une option de grande dimension, c.-à-d. de taille $d = 40$. Le temps total de valorisation est de 35min 43s. Nous remarquons que le temps associé à la dernière unité de calcul (AMD Opteron) est relativement bas. Ceci est dû au fait que nous arrondissons la part d'instances d'entrainement à faire calculer à chaque GPU/CPU si la proportion calculée a un reste, et attribuons le reste non attribué au dernier GPU/CPU. Le dernier processeur AMD obtient donc une charge plus faible que ses homologues.

4. Autres approches de répartition de tâches pour GPUs

Estimer le partitionnement optimal des données, sur une architecture de type GPU, permet de maximiser son occupation, et donc minimiser le temps global d'exécution. De plus en plus de travaux traitent de ce sujet dans le cadre GPU. L'objectif de cette section est de donner un aperçu, bien que probablement non exhaustif, des propositions récentes dans ce domaine qui ciblent non seulement un seul GPU mais de plus un plus couramment un cluster de GPUs, ou l'usage combiné de CPUs et GPUs. Ce bref bilan nous permet de mieux positionner notre approche de répartition de charge, et les raisons qui nous y ont conduits, voire les alternatives que nous aurions pu suivre.

Résoudre le problème de répartition de charge peut se concevoir soit de manière générique, soit de façon ad-hoc étant donné le calcul à distribuer. L'approche retenue dans notre travail relève de cette dernière. Néanmoins, nous verrons au fil de cette section quelques solutions pouvant constituer des pistes alternatives, souvent génériques. Il ressort que toute stratégie de répartition des tâches nécessite de posséder une estimation de la durée (complexité) de chaque tâche et de la performance de la ressource ciblée pour effectuer une telle tâche. Les travaux diffèrent essentiellement selon que cette durée soit connue à priori (et dépend donc très fortement du problème), ou estimée en préambule du calcul, voire au fur et à mesure (méthode relevant dans ce cas de répartition de charge dynamique).

de Camargo [103] présente une stratégie de répartition des tâches sur cluster hétérogène de GPUs, pour réduire le temps d'exécution d'un simulateur de réseau neuronal. Il estime la quantité de données à attribuer à chaque GPU en formalisant son problème en un système d'équations linéaires. Certaines variables du système sont les fonctions de temps d'exécution des kernels de chaque GPU selon la taille des données. Cela nécessite d'exécuter chaque kernel sur chaque GPU pour différentes tailles de données afin d'estimer les fonctions d'interpolation. Cette stratégie peut s'avérer couteuse si le cluster inclut de nombreux GPUs de différents types, et si le programme inclut des kernels coûteux en temps de calcul. Au contraire, notre stratégie « avancée » de répartition, exécutant des kernels peu couteux, permet de comparer rapidement les niveaux de performance des GPUs sur chaque kernel ; et à partir de là, d'inférer le temps prévisible de calcul de tâches à venir (dont la complexité dépend de la longueur de trajectoires à simuler).

Tse et al [104] proposent un scheduler de simulations de Monte Carlo, pour cluster de GPUs et FPGAs. Chaque accélérateur obtient du distributeur de

simulations, une partie des simulations à exécuter; qui selon le choix de l'utilisateur, augmente linéairement ou exponentiellement à chaque distribution. Ainsi, l'accélérateur le plus rapide simule un plus grand nombre de trajectoires sur une période donnée. Cette stratégie est inadaptée à notre application qui exécute pour chaque temps discret de nombreux kernels, pouvant être peu coûteux selon les paramètres de pricing. Par ailleurs, notre stratégie de répartition du calcul des instances d'entrainement est en mesure de fixer la quantité de travail attribuée à chaque `worker` au lancement de l'application (**Figure 18** fin de [partie I]), quantité qui a l'avantage de rester valable pour toutes les itérations ([partie III]). Cependant, l'auteur emploie cette stratégie pour valoriser l'option asiatique, dont le payoff dépend de la moyenne du sous-jacent sur une période donnée : ainsi son approche pourrait être pertinente pour notre étape finale de calcul [partie IV] de l'option américaine, dans le cas où nous ne réussirions pas, faute d'un nombre assez important de GPUs, à distribuer suffisamment les nb_MC simulations que chaque GPU devra effectuer. En effet, rappelons que la stratégie que nous prônons ne peut se baser que sur une estimation de la longueur de chaque trajectoire. Si l'estimation, forcément stochastique, s'avère ne serait-ce qu'un peu incorrecte par rapport à la réalité, cette erreur d'estimation accumulée sur un nombre important de trajectoires attribuées à une unité pourrait finalement engendrer un déséquilibre de charge perceptible entre les différentes unités. Il pourrait alors être plus pertinent d'adopter une approche où chaque worker viendrait requêter le merger pour obtenir un prochain paquet de simulations (potentiellement longues puisque démarrant en $t = 1$) à réaliser dès le précédent paquet terminé.

Serban et al. [105] élaborent un modèle pour estimer les temps de calcul sur un GPU et CPU, leur permettant ainsi de calculer la proportion de calcul à distribuer à chacun. Le modèle est illustré dans **Figure 26**.

Name	Description
D	The total number of (sub-)tasks to process
S	The size of each task (bytes).
P	The proportion of D processed by the CPU ($0 \leq P \leq 1$).
Q	The proportion of D processed by the GPU ($Q = 1 - P$).
C	The number of CPU cores available.
G	The number of GPU cores available.
T_r	Transfer rate to/from the GPU memory (in bytes/sec).
T_s	Data transfer setup time (CPU to/from GPU) (secs).
T_{f1}	CPU time to compute one task (secs).
T_{f2}	GPU time to compute one task (secs).

Figure 26 [105] Paramètres du modèle de coût.

Les temps de calcul d'un même appel de fonction sur un jeu de données, sur CPU et GPU, sont représentés ainsi

$$T_{CPU} = \frac{D \times P \times T_{f_1}}{C}$$
$$T_{GPU} = 2 \times \left(T_s + \frac{D \times S \times Q}{T_r}\right) + \frac{D \times Q \times T_{f_2}}{G}$$

Pour estimer P, les auteurs posent

$$T_{CPU} = T_{GPU}$$

et obtiennent la proportion de tâches dédiées au CPU ainsi

$$P = \frac{\frac{2 \times T_s}{D} + \frac{2 \times S}{T_r} + \frac{T_{f_2}}{G}}{\frac{T_{f_1}}{C} + \frac{2 \times S}{T_r} + \frac{T_{f_2}}{G}}$$

Dans notre cas, considérant qu'une tâche correspond à la simulation d'une trajectoire, nous ne pourrions appliquer ce modèle. En effet, celui-ci requiert un temps de traitement identique pour chaque tâche, ce qui n'est pas le cas des trajectoires de longueur variable de l'option américaine, de par les temps d'arrêts aléatoires.

En [106] les auteurs proposent une stratégie de découpage dynamique des tâches pour une architecture CPU-GPU, surpassant jusqu'à 45% les performances des autres stratégies de scheduling comparables. Après un premier partitionnement équitable entre les 2 unités de calcul, le scheduler collecte les temps de calcul et en déduit une vitesse de traitement pour chacun. Le scheduler effectue un nouveau partitionnement des tâches selon les vitesses estimées, et continue ainsi jusqu'à ce que la variance du temps d'exécution de 2 partitionnements consécutifs soit suffisamment basse. Dans ce cas, le reste des tâches sera partitionné selon les dernières vitesses estimées du CPU et GPU. Dans notre cas, le nombre d'instances d'entrainements simulées nous parait insuffisant pour considérer cette stratégie auto adaptive. Et donc, nous préférons une approche où dès le départ, chaque GPU connait la quantité totale du travail qui devra être effectuée.

Shirahata et al. [107] proposent un scheduler alternatif à celui d'Hadoop MapReduce, pour cluster de GPUs/CPUs. Celui-ci repose sur un monitoring des tâches de type map, pour collecter périodiquement les informations relatives à l'exécution et estimer un facteur d'accélération α

$$\alpha = \frac{temps\ moyen\ d'exécution\ sur\ coeurs\ CPU}{temps\ moyen\ d'exécution\ sur\ GPUs}$$

En notant N le nombre de tâches de type map, n le nombre de cœurs CPU, m le nombre de GPUs, x et y les valeurs à estimer correspondant aux nombres de tâches sur cœurs CPU et GPUs, les auteurs cherchent au travers de leur stratégie de répartition dynamique des tâches, à minimiser le temps de calcul global de la phase "map" ainsi

Minimiser

$$f(x,y)$$

tel que

$$f(x,y) = max\left\{\frac{x}{n} \times \alpha \times t, \frac{y}{m} \times t\right\}$$
$$x + y = N$$
$$x, y \geq 0$$

Cette stratégie confère un facteur d'accélération de 1.93 comparé au scheduler standard (bien que modifié pour savoir profiter de GPUs) d'Hadoop, pour l'exécution complète de leur job MapReduce de test sur 64 nœuds cumulant 1024 cœurs CPU et 128 GPUs.

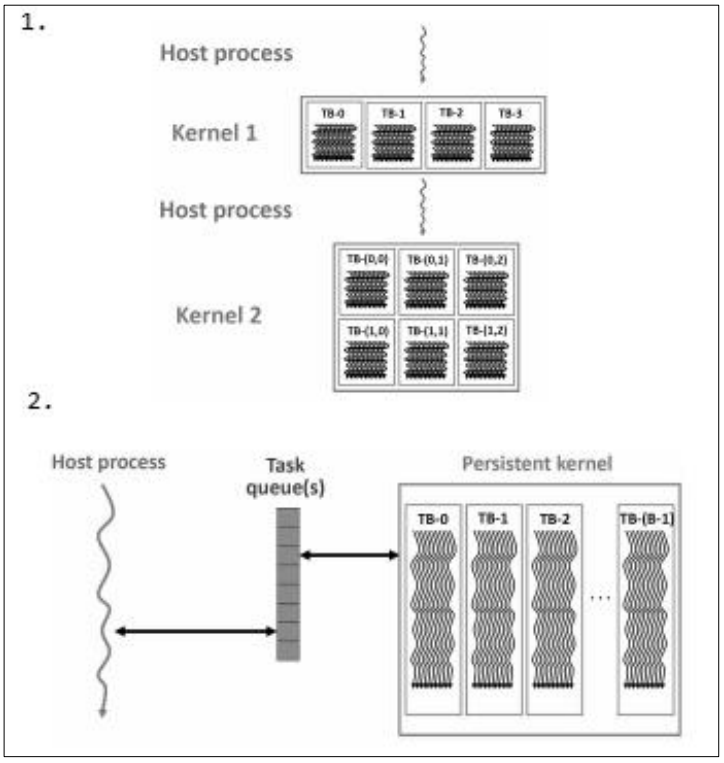

Figure 27 [108] 1. Paradigme de programmation CUDA 2. Paradigme de file de tâches.

Chen et al. [108] traitent du problème de scheduling de tâches sur GPUs à plus bas niveau (comme dans un système d'exploitation), et de manière totalement générique. Ils proposent d'exécuter un kernel persistent, qui dépile les tâches d'une file gérée par le CPU, et les attribue tout au long de son exécution à ses blocs de threads **Figure 27**. Les transferts CPU-GPU et l'exécution du kernel persistant sont évidemment concourants et asynchrones. Les auteurs obtiennent un speedup quasi linéaire en exécutant leur simulateur de dynamique moléculaire sur un cluster allant de 1 à 4 GPUs.

Dans [109] les auteurs s'interrogent sur l'efficacité d'une stratégie de scheduling dynamique, lorsque celle-ci utilise comme paramètres, les débits de traitement de donnée des ressources en jeu (GPU, FPGA). En effet, les débits mesurables peuvent varier au cours du temps, rendant moins efficace l'action du scheduler, si celui-ci ne les réévalue pas. Les auteurs cherchent de plus à optimiser la fréquence de ré-estimation des débits pour limiter son coût. Le module d'apprentissage qu'ils introduisent, estime un temps de calcul qu'ils comparent avec celui relevé précédemment. Si la période entre deux estimations est longue et l'erreur importante, les débits seront réévalués.

Finalement la parallélisation des algorithmes d'apprentissage fait l'objet de nombreux travaux. Dans ce contexte se pose aussi la problématique de la répartition des forêts sur les différentes ressources parallèles. Dans [110] est présentée CudaRF, une version CUDA des forêts aléatoires. Durant la phase d'entrainement, chaque thread du GPU construit un arbre de la forêt. Nous pourrions l'appliquer dans notre phase d'entrainement des forêts distribuées sur CPUs, pour lui faire bénéficier d'un double niveau de parallélisation : chaque `worker` en charge d'une sous-forêt déléguerait à son GPU cette construction. Cependant, attribuer à chaque thread GPU un arbre de la forêt lors de la classification, n'est pas envisageable, du fait que nous exploitons l'ensemble du GPU, pour appeler simultanément différents classificateurs/forêts, selon le pas de temps atteint par la trajectoire courante simulée par chaque thread.

5. Conclusion

Nos travaux proposent une implémentation multi-CPU-GPU de l'algorithme de Picazo, pour valoriser l'option américaine sur panier. Cette implémentation présente les caractéristiques rappelées ci-dessous.

Pour exploiter pleinement le double niveau de parallélisme offert par notre cluster de GPUs/CPUs, nous distribuons le calcul des instances d'entrainement sur le cluster, et sollicitons l'architecture SIMT de chaque unité de calcul pour paralléliser l'ensemble des simulations de Monte Carlo de

l'algorithme. Notre stratégie de calibration des paramètres de kernel que nous lançons évidemment en parallèle en phase initiale, peut s'appliquer à un large panel de GPUs/CPUs, et ainsi à de nombreux clusters.

Pour cluster hétérogène, nous présentons une stratégie dynamique de répartition de charge basée sur une prédiction des temps de calcul des instances d'entrainement réduisant de 36% le temps de pricing parallèle d'une option sur 7 actifs.

L'intégration des forêts aléatoires permet de supprimer le goulot formé par les entrainements séquentiels des classificateurs en les distribuant, mais rallonge le temps de création des instances d'entrainement dû à la nature plus complexe de cette méthode de classification. A terme, il pourrait être pertinent d'évaluer l'applicabilité d'une autre méthode d'apprentissage automatique peu coûteuse en temps de classification dont l'entrainement est scalable [111].

Aussi, augmenter considérablement le nombre de `workers` (100+) permettrait de réduire le nombre d'instances d'entrainement attribuées à chacun, pour limiter le surcoût de classification. Cependant, augmenter les ressources du cluster complexifie les opérations de broadcast/merge, impactant alors le temps global de pricing. Pour résoudre ce dernier problème, nous pourrions intégrer l'une des versions de broadcast détaillée en [112] qui distribue la propagation des données entre nœuds adjacents sur le réseau. Nous pourrions aussi remplacer les opérations de merge par des réductions parallèles selon un arbre binaire.

Néanmoins, nous franchissons le seuil symbolique d'une heure avec 12 GPUs, pour une option sur 40 actifs (cf. 2.3). Nous surperformons la version pour cluster de CPUs dépassant les 8 heures avec 64 cœurs CPU pour une option d'une telle taille. Comparé à une version d'un tel pricing d'une option américaine, certes moins complexe et exécuté en séquentiel, nous obtenons un speedup de 140 sur 4 GPUs. Grâce aux résultats déjà obtenus et en considérant les pistes d'amélioration de l'implémentation citées plus haut, notre travail démontre qu'il est possible de valoriser une option américaine de grande dimension, en un temps extrêmement raisonnable (quelques dizaines de minutes, voire une poignée de minutes). Ceci nous permet aisément d'envisager l'extension de ce travail au calcul de la Monte Carlo VAR d'un portefeuille, composé d'instruments aussi complexes que de telles options américaines.

Chapitre VI. Monte Carlo Value at Risk d'un portefeuille d'options sur cluster hétérogène de GPUs/CPUs

1. Introduction

La première partie de ce chapitre généralise les contributions des deux précédents, en proposant une approche pragmatique de valorisation d'un portefeuille composé d'instruments, tels des options américaines et européennes. Nous y discutons des différentes manières de distribuer la valorisation des instruments d'un portefeuille.

Puis ce chapitre traite de la dernière partie de nos travaux concernant le calcul de la VaR d'un portefeuille d'options européennes et américaines, compliquant considérablement les calculs, déjà couteux pour la valorisation d'un seul instrument comme une option américaine. Nous proposons ici plusieurs optimisations pour réduire le temps d'exécution. Nous abordons dans un premier temps la question du surcoût de la revalorisation de la frontière d'exercice nécessaire lorsque l'on doit valoriser de multiples fois la même option américaine, et apportons une solution qui tire simplement partie de propriétés de l'algorithme de Picazo. Ensuite, nous décrivons comment valoriser nb_VaR fois un portefeuille sur cluster, et calculer ainsi la MC VaR. Les tests illustrent et mettent en évidence la pertinence de nos choix dans les différentes sections.

2. Optimisation du pricing sur cluster d'un portefeuille d'options

L'ordonnancement des tâches est une étape clé dans notre optimisation du pricing d'un portefeuille. Cette section définit une stratégie de répartition de tâche à la fois temporelle et spatiale, qui dresse une fois pour toute, une liste des ressources à solliciter pour le pricing de chaque instrument, présentant l'avantage d'être réutilisable. La complexité du problème réside dans les possibilités multiples d'ordonnancement et de placement, en partie dus aux découpages possibles d'un pricing sur plusieurs ressources. Cette solution ne décrit pas forcément l'ordonnancement idéal, mais tente de s'y rapprocher en un temps raisonnable.

Commençons par dissocier les méthodes de classification utilisables par notre application : AdaBoost et SVM d'une part, dont l'entrainement est centralisé sur une ressource, les forêts aléatoires d'autre part, dont l'entrainement est distribué. On rappelle que le choix de telle ou telle méthode d'entrainement ne modifie pas fondamentalement l'approche de pricing: celle-ci reste basée sur un principe de classification pour établir la frontière d'exercice.

La **Figure 28** illustre un exemple simple de pricing d'un portefeuille de deux options américaines (supposées pour simplifier, de même complexité) sur un cluster de 2 devices, via une classification à entrainement centralisé. Distribuer chaque option sur l'ensemble du cluster (**Figure 28** avec découpage, ou autrement dit avec distribution), nécessite de traiter les options l'une à la suite de l'autre, et d'entrainer donc successivement (sur un seul device) les classificateurs respectifs. Comme on peut le constater, ceci engendre un gaspillage de ressource de calcul puisque l'entrainement du classificateur est séquentiel. Ce surcoût est pallié en valorisant chaque option simultanément chacune sur un des 2 devices (**Figure 28** sans découpage) : sur chaque device, on procède donc à l'entrainement naturellement séquentiel auquel s'ajoute le temps, également séquentiel, de création des instances d'entrainement.

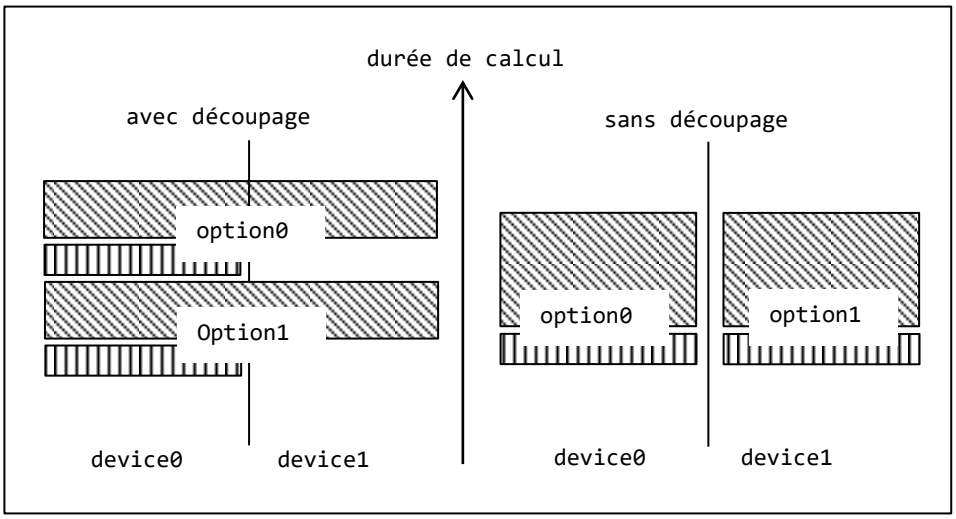

Figure 28 Représentation des temps cumulés des estimations des instances d'entrainement (blocs à rayures obliques) et des entrainements des classificateurs non distribués (blocs à rayures verticales) de deux options américaines de même complexité sur un cluster de 2 devices, avec et sans découpage.

Nous présentons en **Figure 29** les temps d'exécution de pricing d'un portefeuille de 4 calls américains avec et sans découpage, sur un cluster de 4 CPUs. Dans cet exemple, lancer indépendamment sur chaque device le pricing de chaque option permet de réduire de 33% le temps de pricing du portefeuille comparé à celui obtenu en distribuant au maximum le pricing de chaque instrument (169s contre 252s). Cela permet aussi de solliciter le Xeon E5520 que 60% (100s) du temps de valorisation du portefeuille, et ainsi lui donner l'opportunité de se voir attribuer des tâches supplémentaires sans pour autant dépasser le temps de calcul avec découpage. Evidemment, ce comportement

favorable au cas sans découpage ne se présenterait pas forcément dans le cas d'un portefeuille de 5 options et un cluster de 4 devices ; plus généralement dans le cas où sans découper une option, un ou plusieurs devices sont inutilisés, il est possible que la solution avec découpage malgré que l'entrainement soit séquentiel, devienne meilleure.

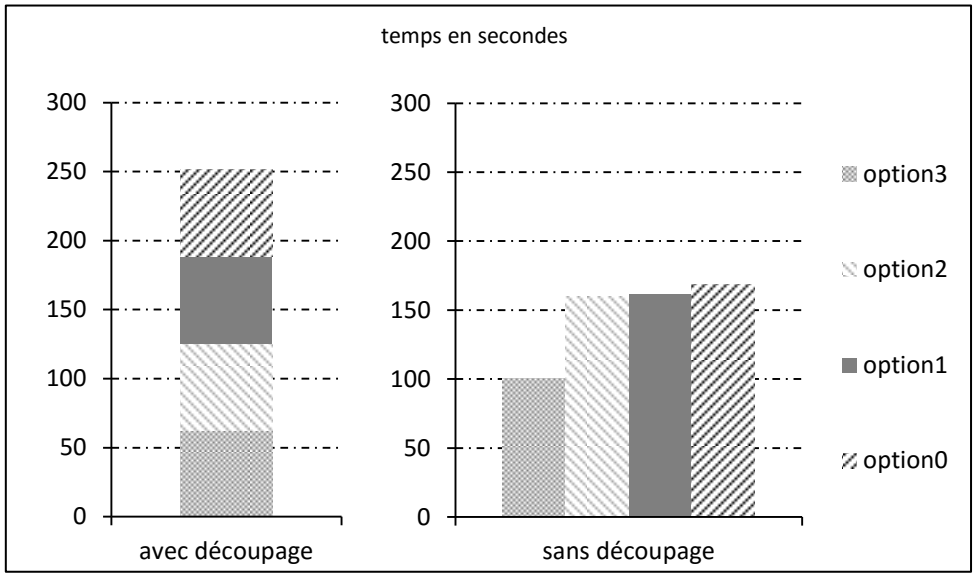

Figure 29 Comparaison des temps d'exécution de pricing d'un portefeuille d'options américaines avec et sans découpage, sur un cluster composé de 3 AMD Opteron 250 et de 1 Xeon E5520. Dans la stratégie avec découpage, chaque option est distribuée sur l'intégralité du cluster. Dans la stratégie sans découpage, chacune est valorisée sur un seul CPU. l'option 3 est valorisée sur le Xeon E5520. Calls américains de moyenne géométrique, $S_{t_0}^i$=100, d= 10, K = 100, N= 10, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision.

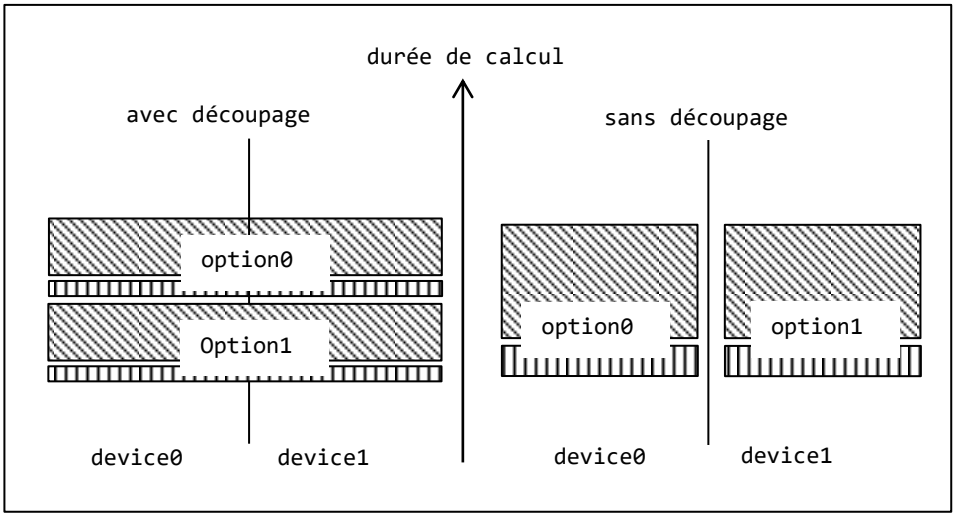

Figure 30 Représentation des temps cumulés des estimations des instances d'entrainement (blocs à rayures obliques) et des entrainements distribués des classificateurs (blocs à rayures verticales) de deux options américaines sur un cluster de 2 devices, avec et sans découpage.

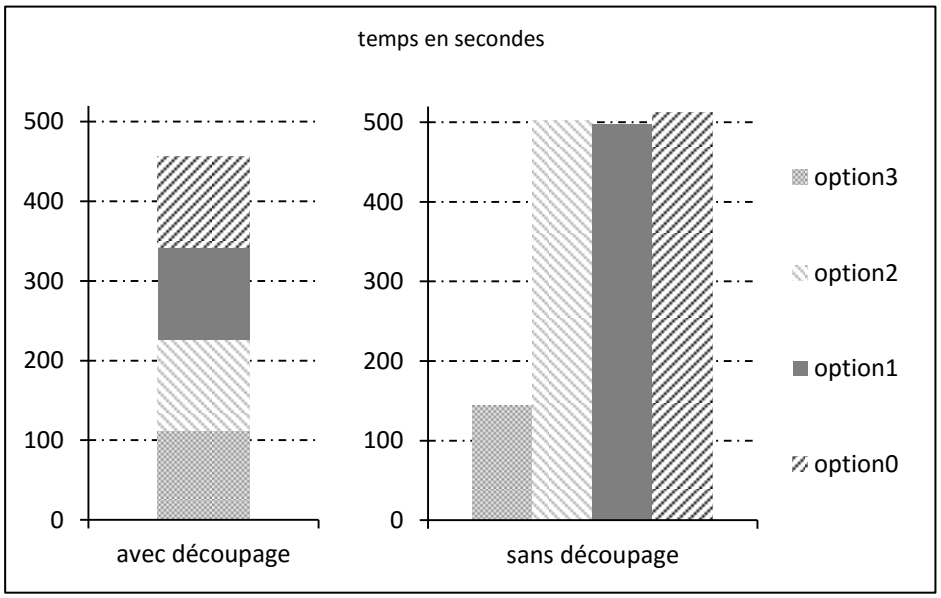

Figure 31 Comparaison des temps d'exécution de pricing d'un portefeuille d'options américaines avec et sans découpage. Les conditions sont les mêmes qu'en figure **Figure 28**. Forêts aléatoires de 150 arbres ayant 10 de hauteur maximale.

La **Figure 30** reprend l'exemple de la **Figure 28** dans le cas des forêts aléatoires autorisant du parallélisme pour la phase d'entrainement. Ceci met exergue un choix moins évident entre distribuer ou non le pricing de chaque instrument sur l'intégralité des ressources du cluster.

Les tests de la **Figure 31** révèlent que distribuer le pricing de chaque option permet de réduire de 10% le temps de pricing du portefeuille (457s contre 512s). Cependant, le Xeon E5520 n'est sollicité que 28% (144s) du temps de valorisation du portefeuille. Sans doute occuper utilement le Xeon en valorisant d'autres instruments peu couteux permettrait d'avantager la stratégie sans découpage.

L'utilisation des forêts aléatoires peut entrainer un surcout selon ses paramètres, durant l'estimation des instances d'entrainements (Chapitre V.2.2). L'utilisation d'un cluster suffisamment grand divise néanmoins ce coût, et permet d'introduire une stratégie d'ordonnancement du pricing des instruments du portefeuille simple et adaptée à l'entrainement non centralisé des classificateurs, basée sur la distribution systématique de chaque instrument sur l'ensemble du cluster.

Au contraire, une stratégie d'ordonnancement adaptée aux cas AdaBoost ou SVM, doit tenter de limiter le degré de distribution de chaque valorisation d'instrument, car cela engendre des périodes de non utilisation de ressource durant les entrainements centralisés. Limitons signifiant ne pas forcément utiliser toutes les ressources du cluster pour valoriser un instrument. Nous pourrions introduire un simple ordonnancement de tâche consistant à attribuer une nouvelle tâche (le pricing d'un instrument en séquentiel) à chaque nouvelle ressource qui devient libre. Cette solution est envisageable si les derniers instruments valorisés n'occasionnent pas de surcoût comme l'illustre la **Figure 32**. Dans notre cas où l'on considère un portefeuille de taille raisonnable mais composé d'options américaines en général couteuses à estimer, le surcout d_1 peut être non négligeable par rapport à d_2, d'autant plus dans un cluster hétérogène.

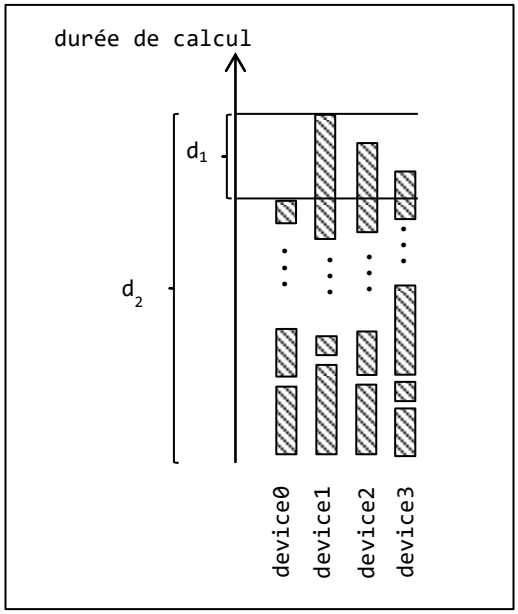

Figure 32 Représentation d'un scheduling simple des valorisations d'instruments sur un cluster de 4 devices. On attribue une nouvelle tâche à chaque device libre. Le surcoût d_1 est négligeable si $d_1 \ll d_2$.

Pour le cas de classificateurs dont l'entrainement est centralisé, notre idée est donc de proposer une stratégie étendant celle illustrée en **Figure 32**, mais atténuant la perte de temps occasionnée par les derniers instruments valorisés, c-à-d s'autoriser à distribuer plus ou moins largement le pricing des derniers instruments. Pour ce faire, l'ordonnanceur a besoin d'estimer des coûts simplifiés de pricing parallèle, simplifiés car basé uniquement sur les temps théoriques des kernels (Chapitre V.3.2), et ce au démarrage de l'application (puisqu'on rappelle qu'on cherche à calculer l'ordonnancement et le placement de toutes les tâches du portefeuille à l'avance, et une fois pour toutes). Ce cout est calculé ainsi, pour le pricing d'un instrument i sur m devices

$$pricingCost_{i,d_{1\dots m}} = \max_{1\le k\le m}\left(\sum_{t=1}^{T-1} nb\_class_{i,d_k}(t) \times durationKernel_{i,d_k}(t)\right)$$

Notons que les valeurs de nb_class_{i,d_k} permettent déjà d'équilibrer les calculs sur les devices, et le calcul du max s'effectue sur des coûts presque équivalents. Ainsi, notre ordonnanceur reproduit la même stratégie de la **Figure 32**, avant de s'intéresser aux derniers instruments du portefeuille à pricer, et décider alors de les découper sur plusieurs ressources pour lisser l'histogramme final. L'**Algorithme 6** décrit une heuristique pour produire un empilement de tâches le

plus lisse possible. Cet empilement sera ultérieurement exécuté en partant du bas.

Algorithme 6 Algorithme d'ordonnancement du pricing de portefeuille, supposant l'utilisation d'un classificateur centralisé tel AdaBoost pour toutes ses options américaines.

Entrée: $P = \{a_1, \dots, a_n\}$ portefeuille de n instruments
Entrée: $C = \{d_1, \dots, d_m\}$ cluster de m devices
Sortie: $F = \{(a_1, c_1, p_1), \dots, (a_n, c_n, p_n)\}$, $c_i \subset C$ et p notre coût théorique de pricing de a sur c_i

Selon notre modèle, dresse le même schéma d'allocation centralisée que suivrait le scheduler en **Figure 31**, c.-à-d. plaçant sur un device libre un des n instruments
1 :$F = \emptyset$
2 :Créer $T = \{(d_1, t_{d_1}), \dots, (d_m, t_{d_m})\}$ tel que t_d représente le total des coûts d'utilisation de d et initialiser $t_{d_1} \dots t_{d_m} = 0$
3 :**Pour** i **de** 1 **à** n
4 : Trier T par ordre croissant selon t_d
5 : $F = F \cup \{(a_i, d_j, pricingCost_{i,d_j})\}$ tel que (d_j, t_{d_j}) figure en tête de T
6 : $Update(T, j, pricingCost_{i,d_j})$ c.-à-d. dans T faire $t_{d_j} \mathrel{+}= pricingCost_{i,d_j}$
7 :**Fin pour**

On découpe alors le pricing des derniers instruments plutôt que de les garder centralisés, afin de réduire le temps total de pricing du portefeuille
8 :**Tant que** $\exists(a_j, c_j, p_j)$ dans F tel que $\#c_j = 1$ (non distribué) et dernier à utiliser c_j
9 : $currentMaxCost = \max(t_{d_0}, \dots, t_{d_m})$
10 : $Fsave = F$
11 : Retirer (a_j, c_j, p_j) de F
12 : Mettre à jour T avec $t_{d_j} \mathrel{-}= p_j$ tel que $d_j \in c_j$, c.-à-d. soustraire de T le coût de a_j
13 : Trier T par ordre croissant selon t_d
14 : $kSave = -1$
Découper a_j afin de diminuer $currentMaxCost$. Etant donné les devices libres (selon informations dans T) découpage n'utilise pas forcément les m devices
15 : **Pour** k **de** 1 **à** m
16 : $newMaxCost = getMaxPricingCost(T, k, pricingCost_{i,d_{1\dots k}})$, nouveau coût maximal sur C si on distribuait a_j sur les k premiers devices
17 : **Si** $newMaxCost < currentMaxCost$

18 : $currentMaxCost = newMaxCost$
19 : $kSave = k$
20 : **Fin si**
21 : **Fin Pour**
22 : **Si** $kSave <> -1$
23 : $F = F \cup \{(a_j, \{d_1, \dots, d_{kSave}\}, pricingCost_{i,d_{1 \dots kSave}})\}$, on planifie l'exécution de a_j , cette fois distribuée
24 : $Update(T, 1 \dots kSave, pricingCost_{i,d_{1 \dots kSave}})$, on met à jour T
25 : **Sinon**
26 : **Retourner** $Fsave$, on ne peut plus lisser
27 : **Fin Si**
28 : **Fin Tant que**
29 : **Retourner** F

L'option européenne, peu couteuse en temps de calcul, peut se satisfaire de la stratégie évoquée en **Figure 32**, consistant à. appliquer l'**Algorithme 6** jusqu'à la ligne 7. Dit autrement, nous ne tenterons pas de la distribuer sur plus d'un device, afin de lisser l'histogramme. Le lissage ne s'effectue qu'avec l'aide des options américaines.

La **Figure 33** illustre la partie lissage décrite dans l'algorithme qui débute à la ligne 8.

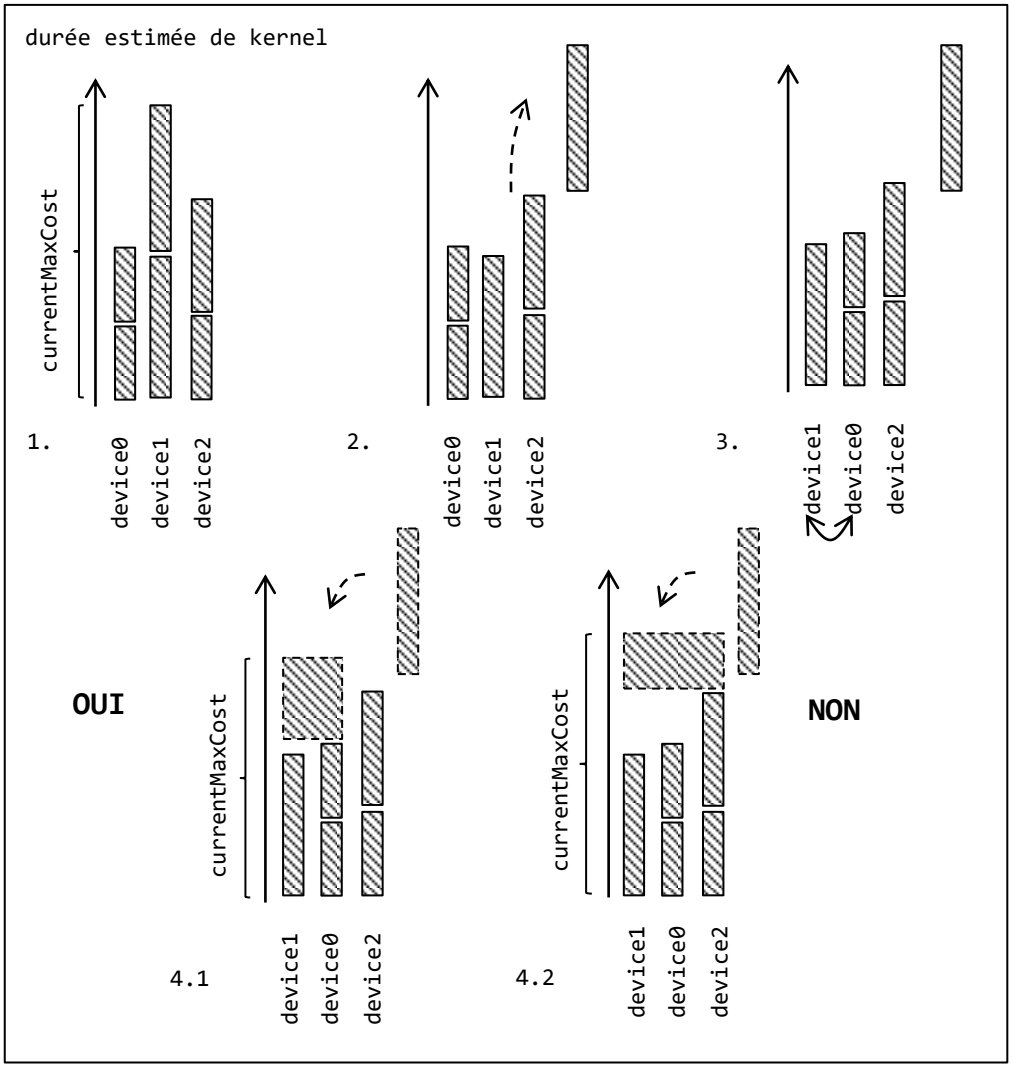

Figure 33 Exemple des phases d'exécution de l'**Algorithme 6** calculant l'ordonnancement de la valorisation d'un portefeuille de 6 options sur un cluster de 3 devices. Etat de l'ordonnancement en **1**. ligne 8 **2**. ligne 11 **3**. Ligne 13 **4.1**. ligne 16 puis ligne 18. **4.2**. ligne 16.

Pour nos tests comparatifs, nous valorisons sur un cluster de 3 devices, un portefeuille composé de 6 options américaines (dont la valorisation se fonde sur un entrainement de classificateur centralisé) : call américains de moyenne géométrique, $S_{t_0}^i$=110, d={1,1,5,5,10,10}, K = 100, N= 10, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision. Le cluster inclut : 1 Tesla M2075, 1 Tesla S1070 et 1 AMD Opteron250. Nous comparons les 3 méthodes d'ordonnancement : une distribution systématique de chaque instrument, celle illustrée en **Figure 32**, et celle décrite par l'**Algorithme 6** dont l'exécution est décrite par la **Figure 33**.

Nous exécutons plusieurs fois la même stratégie pour confirmer l'ordre relatif des temps d'exécution de chacune. Nous obtenons un prix de 79.27397 ($\sim 10^{-5}$) en 342s avec notre stratégie, contre 371s en distribuant chaque instrument, et 484s via un ordonnancement de chaque instrument sans aucune distribution (**Figure 32**).

Nous avons ainsi défini une stratégie complète d'ordonnancement du pricing d'un portefeuille d'instruments européens et américains. Et nous avons pu vérifier expérimentalement qu'elle améliore les temps d'exécution comparé aux autres alternatives, consistant à pricer chaque instrument en séquentiel ou sur l'intégralité du cluster.

3. Optimisation du calcul de la MC VaR d'un portefeuille d'options

3.1. Réutilisation de la frontière d'exercice

En prémices de l'estimation de la Monte Carlo VaR, nous devons déterminer les étapes de la valorisation de l'option américaine pesant sur le temps de calcul. Nous renvoyons le lecteur au Chapitre II.1.1 pour un rappel de l'estimation de la VaR. Dans l'exemple de la **Figure 14** illustrant la valorisation d'une option américaine sur un seul GPU, l'estimation de la frontière d'exercice représente plus de 99% du temps global d'exécution. L'utilisation d'un cluster permet de réduire son coût mais n'est pas une optimisation suffisante si cette étape est ré exécutée de nombreuses fois.

L'avantage de l'algorithme de Picazo est d'estimer une frontière d'exercice indépendamment des simulations de MC finales. En effet, un point de la frontière d'exercice est représenté par un classificateur H_t. Ce dernier est entrainé par nb_class instances d'entrainement, chacune requérant la simulation d'un payoff Ψ_t et d'une valeur de continuation C_t. Ψ_t et C_t et sont estimés à partir des prix S_t^i du panier d'actifs, eux-mêmes résultant d'une simulation des prix initiaux du panier, c.-à-d. $S_{t_0}^i$. Pour la valorisation de l'option en un temps t autre que t_0, l'étape finale s'effectue à partir de nouveaux prix du panier simulés en t à partir de t_0, indépendamment donc de la frontière d'exercice. Ainsi, l'option peut être revalorisée tout au long de sa vie avec cette même frontière. La figure ci-dessous schématise l'opportunité ainsi offerte d'accélérer le calcul de la VaR d'une option américaine du fait de ne nécessiter qu'une seule fois l'estimation de la frontière d'exercice.

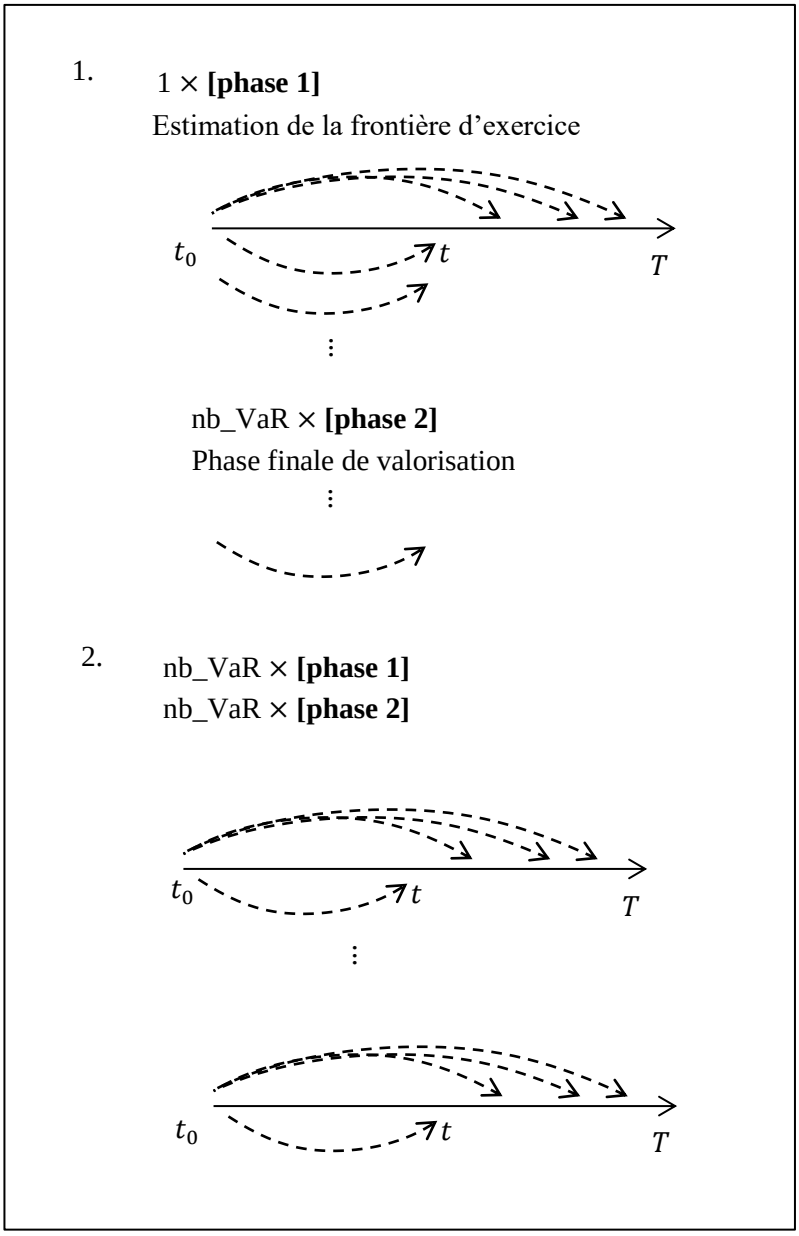

Figure 34 Valorisation multiple de l'option américaine sur panier avec 1. Réutilisation de la frontière d'exercice 2. Ré estimation de la frontière d'exercice. Les flèches sur les axes représentent les trajectoires qu'empruntent les prix du panier simulés de t_0 à $t_i \in]t_0, T[$, pour l'entrainement du classificateur H_i. En effet, pour chaque variable d'entrainement de H_i est calculé un Ψ et C à partir de S_{t_i}. Les flèches sous les axes désignent les trajectoires de prix simulés de t_0 à t. On obtient ainsi dans 1. ou 2. nb_VaR prix de départ S_t pour démarrer les simulations de MC finales et obtenir nb_VaR prix à l'horizon t.

Nous proposons de mesurer l'impact de cette optimisation sur un call américain (**Figure 35**), en représentant les fonctions de répartition des prix simulés en un horizon donné, avec et sans ré estimation de la frontière d'exercice pour chaque pricing. Le test de Kolmogorov-Smirnov (KS) permet de conclure qu'avec une certitude de 99%, nos deux séries de 1000 prix répartis en 200 classes suivent statistiquement la même loi ($D = 0.036 < D_{critical} = 0.073$).

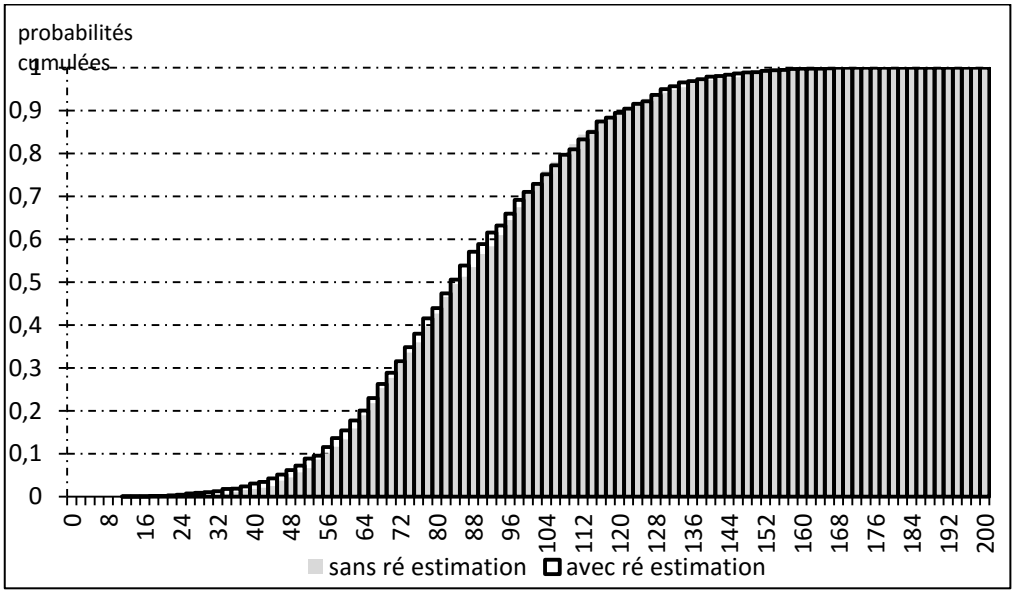

Figure 35 Fonctions de répartition de 1000 prix d'une option américaine simulés en t = 0.5T avec et sans ré estimation de la frontière d'exercice. Intel Xeon E5520. Call américain de moyenne géométrique, $S_{t_0}^i$=200, d= 5, K = 100, N= 10, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 105, AdaBoost/150 arbres binaires de décision.

3.2. Distribution de la MC VaR d'un portefeuille

L'estimation de la MC VaR de notre portefeuille requiert nb_VaR tirages de son prix à un horizon donné. Il est cependant difficile d'orchestrer pour chaque estimation du portefeuille, l'ensemble des estimations de chaque instrument. En effet, pour simuler une trajectoire d'un portefeuille composé d'options américaines sur panier d'actions, il nous faut simuler l'ensemble des processus stochastiques des actifs des paniers de chaque option. De plus, l'estimation de chaque instrument requiert l'initialisation de certains objets spécifiques à sa distribution. Il est donc préférable d'effectuer les nb_VaR valorisations de chaque instrument indépendamment, et ensuite de reconstituer les nb_VaR estimations du portefeuille, comme le décrit l'**Algorithme 7**.

Algorithme 7 Valorisation multiple d'un portefeuille à horizon t.

Entrée: $P = \{a_0, \dots, a_n\}$ portefeuille des n instruments
Entrée: $c_i \subset C$ avec $C = \{d_0, \dots, d_m\}$ ensemble des m devices du cluster
Entrée: temps t
Entrée: $nb\_pricings$
Sortie: $prices[nb\_pricings]$ tableau des prix de P en t

1 : Initialiser $prices$ avec 0
2 : **Pour** i **de** 1 **à** n
3 : **Pour** j **de** 1 **à** $nb\_pricings$
4 : $prices[j] += price(a_i, c_i, t)$
5 : **Fin pour**
6 : **Fin pour**
7 : **Retourner** $prices$

Ainsi, pour le calcul de la VaR, l'**Algorithme 7** est exécuté en fixant le paramètre d'entrée t à l'horizon, et $nb\_pricings$ à nb_VaR.

Dans le cas du pricing multiple, le temps d'exécution ne dépend pas uniquement de la phase finale de pricing exécutant nb_MC simulations, mais aussi des nb_VaR exécutions. Rappelons que la phase 1 qui permet de déterminer la frontière d'exercice n'est à exécuter qu'une seule fois puisque nous pouvons la réutiliser autant de fois que voulu. Donc même si elle est en soi couteuse, son coût est réduit vis-à-vis du temps cumulé à exécuter la phase 2. du pricing américain. Nous distribuons donc la prise en compte de chaque instrument sur l'intégralité du cluster ; et utilisons notre estimateur de temps d'exécution de kernel $durationKernel_{i,w}$ pour répartir sur le cluster hétérogène les nb_VaR exécutions de pricing.

La **Figure 36** décrit la MC VaR à 95% sur l'horizon $0.5T$, d'un portefeuille de 10 options sur un cluster de 30 devices. Celle-ci est estimée à 100.37534 ($\sim 10^{-5}$) en 1h 52 min 22s. Ceci conclut à la faisabilité de réaliser en un temps raisonnable un tel calcul si complexe que l'est la Monte Carlo VaR d'un portefeuille composé d'instruments incluant des options américaines sur panier multidimensionnelles, grâce à l'exploitation d'un cluster d'unités de calcul hétérogènes et d'aussi grande taille que nécessaire.

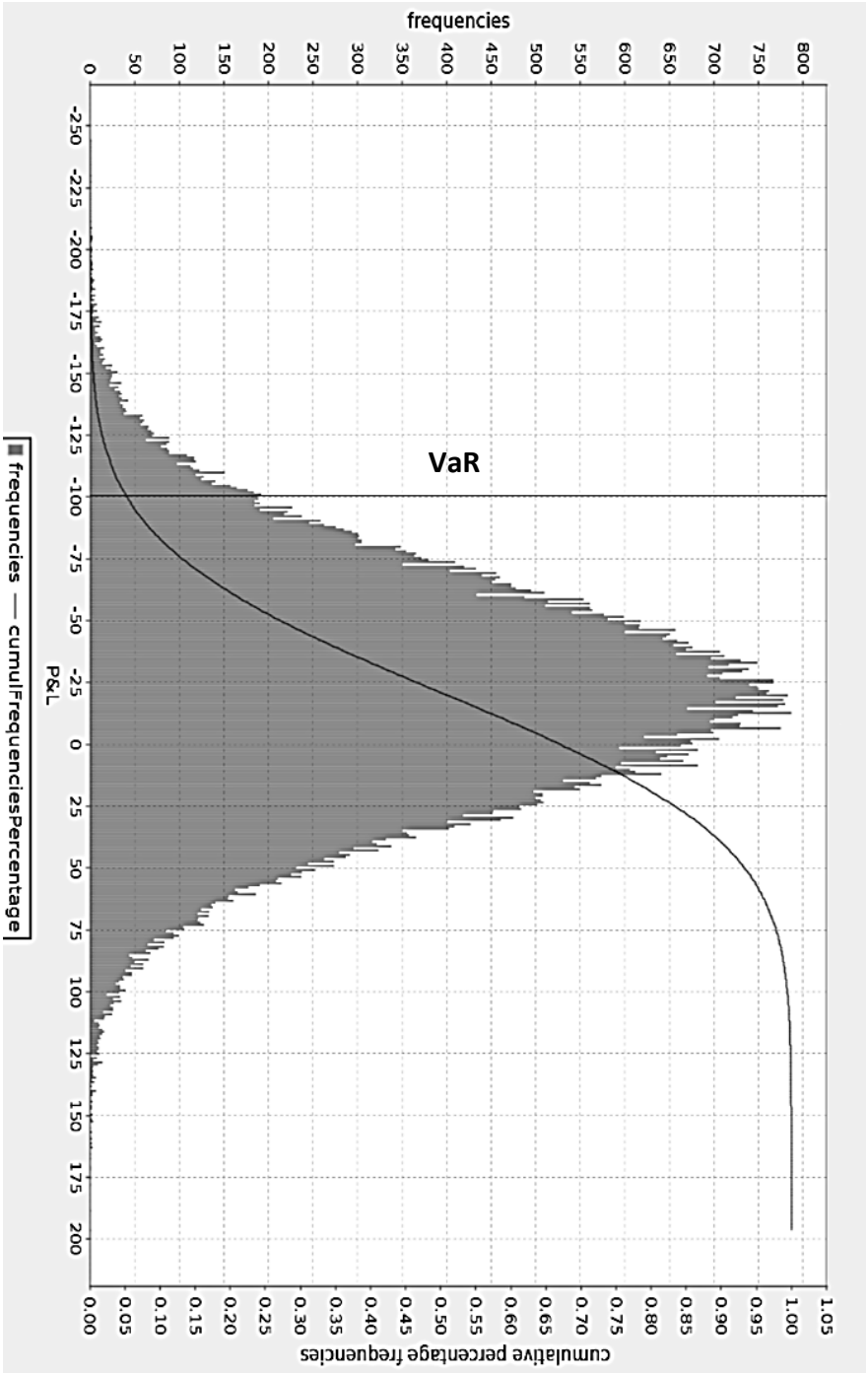

Figure 36 VaR à 95% sur 0.5T d'un portefeuille de 5 calls européens et 5 calls américains de moyenne géométrique, sur un même panier d'actifs. nb_VaR=10^5, pertes et profits du portefeuille répartis sur 500 classes, $S_{t_0}^i$=200, d= 10, K = 100, N= 20, T = 1, r = 3%, δ_i = 5%, σ_i = 40%, nb_class = 3000, nb_cont = 300, nb_MC= 10^5, AdaBoost/150 arbres binaires de décision. Le cluster exploité inclut :1 Tesla M2075, 20 Tesla S1070, 5 Intel Xeon E5440, 4 AMD Opteron 250.

4. Autres approches pour le calcul de la MC VaR

La MC VaR a fait l'objet de nombreux travaux pour réduire son coût d'estimation, et entre autre en recourant à du calcul sur GPU. Dans cette section, nous décrivons brièvement ces travaux, afin de positionner notre solution.

Une des difficultés rencontrée lors de son calcul est la simulation d'évènements rares, c.-à-d. la simulation des prix du portefeuille en queue de distribution. Considérons par exemple l'ensemble des évènements rares d'un portefeuille dont la distribution des prix est dissymétrique à droite. Il est nécessaire de simuler un nombre suffisamment important de pertes du portefeuille, pour calculer la VaR d'un niveau de confiance élevé. Dans notre exemple ci-dessous, on note P_i une estimation de portefeuille et P^{seuil} le seuil de prix sous lequel toute estimation inférieure est considérée comme rare. L'évènement rare x est caractérisé ainsi

$$x \in A \text{ tel que } A = \left\{P^i \middle| P^i < P^{seuil}\right\}$$

Estimer la probabilité d'un tel évènement en procédant à nb_VaR tirages du portefeuille revient à estimer p tel que

$$p = \frac{1}{nb\_VaR} \sum_{nb\_VaR} \mathbb{1}_{x \in A}$$

L'effet sur l'écart-type relatif est

$$\lim_{p \to 0} \frac{\sigma_p}{p} = +\infty$$

De nombreux travaux [113] [114] traitent de l'Importance Splitting (IS) et y apportent des améliorations (Adaptive Splitting). La technique d'IS repose sur la décomposition de la probabilité de survenue d'un évènement rare

$$\mathbb{P}(x \in A) = \prod_{k=1}^{n} \mathbb{P}(x \in A_k | x \in A_{k-1})$$

Plusieurs seuils de prix sont donc définis et notés

$$A_0 \supset \cdots \supset A_k \supset \cdots \supset A_n = A$$
$$A_k = \left\{P^i \middle| P^i < P^{seuil_k}\right\}$$

Avec

$$P^{seuil_0} > \cdots > P^{seuil_k} > \cdots > P^{seuil_n}$$

Entre $P^{seuil_{k-1}}$ et P^{seuil_k}, on simule N trajectoires en partant aléatoirement de celles ayant franchi $P^{seuil_{k-1}}$. On estime ainsi

$$\mathbb{P}(x \in A_k | x \in A_{k-1}) = \frac{nb\ trajectoires\ atteignant\ P^{seuil_k}}{N}$$

Dans notre cas, les trajectoires sont des processus P_t^i représentant la valeur du portefeuille en t. Pour les simuler et conserver leurs états entre deux seuils, il faudrait sauvegarder les valeurs de tous les processus stochastiques des actifs sous-jacents de chaque option du portefeuille. Ne pas procéder par IS requiert donc plus de simulations pour obtenir une meilleure estimation de l'évènement rare, contrainte que nous comblons en exploitant un nombre important de ressources de calcul.

Un autre moyen de calculer la VaR est d'estimer préalablement les grecs (approche delta-gamma). Le delta et gamma d'un portefeuille étant respectivement la sensibilité de son prix et la sensibilité de son delta, face aux variations de cours du sous-jacent. Matthew Dixon et al. [115] proposent plusieurs optimisations sur ce sujet. Dans un premier temps, les auteurs reformulent la fonction de perte du portefeuille, ce qui leur permet d'isoler et de pré calculer une partie de son delta. Les auteurs intègrent le générateur quasi-aléatoire Sobol fourni par le CUDA SDK et l'adaptent à leur pricer, afin qu'une séquence soit générée en blocs contigus de taille uniforme, optimisant le calcul dans leurs cas. Leur implémentation atteint un speedup de 148 comparé à la version non optimisée pour GPU.

Dans [116] est détaillée la MC VaR d'un portefeuille d'actions, et son implémentation sur différents accélérateurs. En particulier, les auteurs obtiennent les meilleurs résultats en terme de trajectoires par seconde via une implémentation combinant CPUs, GPU et FPGA (**Figure 37** Mapping 5): la génération pseudo-aléatoire et la transformation en gaussienne des variables sont effectuées simultanément sur GPU et FPGA (étapes 1 et 2), la corrélation des variables et la simulation des processus de prix simultanément sur CPUs et GPU (étapes 3 et 4), le calcul des pertes et profits sur un seul cœur CPU (étape 5). Bien que nous abordions communément les mêmes étapes pour déterminer la VaR, nous travaillons sur un problème plus complexe : notre portefeuille est constitué d'options, européennes ou américaines qui plus est, dont chacune peut porter sur un panier d'actions possiblement grand. La VaR d'un portefeuille d'options, ou uniquement le pricing d'une option américaine sur panier, requiert un nombre important de ressources pour être estimé en un temps

raisonnable (Chapitre V). Cependant, ce travail ouvre la perspective qu'au-delà des CPUs et GPUs que nous utilisons déjà, nous pourrions intégrer les FPGA exploitables sous OpenCL.

Map	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	C	C	C	C	C
2	8C	8C	8C	8C	C
3	G	G	G	G	C
4	F	F	G	G	C
5	F,G	F,G	7C,G	7C,G	C

Mapping	Performance
1	0.45 Mwalks/s
2	1.1 Mwalks/s
3	80 Mwalks/s
4	60 Mwalks/s
5	81 Mwalks/s

Figure 37 [116] Mapping des différentes versions du moteur de calcul de la VaR et performances. C = CPU, nC = n cœurs CPU, F = FPGA, G = GPU. NVIDIA GTX 260, Xilinx Virtex-4 LX80, et chaque AMD Opteron 2.2Ghz fournit 2 cœurs.

La méthode de quasi MC combinant simulations de MC et génération quasi-aléatoire, présente l'avantage de converger plus rapidement que la méthode de MC selon la complexité du problème (Chapitre II.1.1). Kreinin et al. [117] comparent les deux approches sur le calcul de la VaR d'un portefeuille d'obligations, en calculant le speedup entre les deux méthodes

$$S_\beta(\alpha,\varepsilon) = \frac{N_\beta(\alpha,\varepsilon)}{N^*(\alpha,\varepsilon)}$$

en considérant la VaR à α% en tolérant une erreur de ε% par rapport à une valeur qu'ils considèrent de référence. Du fait que le résultat de la MC VaR diffère suivant la graine du générateur de nombres pseudo-aléatoires, plusieurs exécutions sont effectuées, et β détermine le pourcentage de certitude du résultat. Les auteurs révèlent un speedup variant de 2 à 10 et progressant selon les trois paramètres β, α et ε. Les auteurs concluent sur les limites de la méthode de QMC pour des problèmes aux dimensions plus complexes, et proposent plusieurs pistes comme les techniques de réduction de dimension ou encore combiner les méthodes de QMC et MC.

5. Conclusion

Ce chapitre a détaillé différentes optimisations pour réduire le temps de pricing d'un portefeuille d'options, ainsi que d'estimater sa MC VaR.

Dans un premier temps, après analyse des phases de non utilisation des ressources, en fonction de la méthode de classification considérée, nous adaptons l'ordonnancement naturel (tout centralisé ou tout distribué) des pricings des instruments du portefeuille, et réussissons ainsi à réduire le temps total de pricing du portefeuille. Nos tests mettent ainsi en exergue l'intérêt d'une telle stratégie d'ordonnancement appropriée pour le cas de la valorisation d'option américaine basée sur de l'entrainement centralisé AdaBoost, par rapport à une distribution systématique de chaque instrument. Cette distribution systématique reste par contre pertinente dans le cas d'un entrainement de classificateur distribué tel que basé sur les forêts aléatoires.

Dans un deuxième temps, nous tirons bénéfice de l'algorithme de Picazo permettant la réutilisation de la frontière d'exercice, lors de la valorisation multiple d'un instrument. Nous intégrons cette optimisation à notre moteur de calcul de VaR pour cluster hybride hétérogène. Grâce à ce moteur, nous sommes capables par exemple, et à titre illustratif, d'estimer la MC VaR d'un portefeuille de 10 options sur panier (5 américaines et 5 européennes), requérant au total $10 \times nb\_VaR \times nb\_MC \times d$ simulations de trajectoires finales, et 5 estimations de frontières d'exercice, en moins de 2h.

Chapitre VII. Développement logiciel

1. Introduction

L'objectif de ce chapitre est d'appuyer les choix d'architecture et de programmation OpenCL faits lors de nos développements logiciels, qui ont été nécessaires pour tester et valider les algorithmes décrits dans les précédents chapitres. De plus, l'ensemble de ces développements logiciels contribue à obtenir un unique moteur de valorisation (*pricer*) d'une option, qu'elle soit européenne ou américaine par la méthode de Picazo, configurable selon différents critères et plateformes d'exécution. Ce même pricer est ensuite réutilisé dans la valorisation de portefeuille ainsi que dans le calcul de VaR.
La section 2 décrit l'organisation de nos classes et leurs rôles dans l'application. En section 3 nous présentons les spécificités techniques relatives à l'exploitation du cluster, au niveau inter et intra nœud.

2. Architecture logicielle du *pricer*

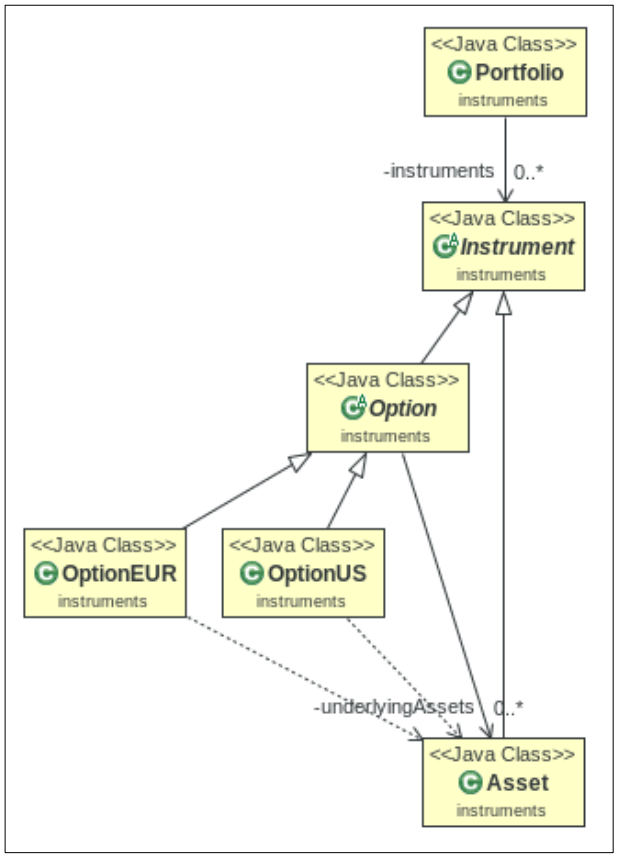

Figure 38 Diagramme de classes du package instruments.

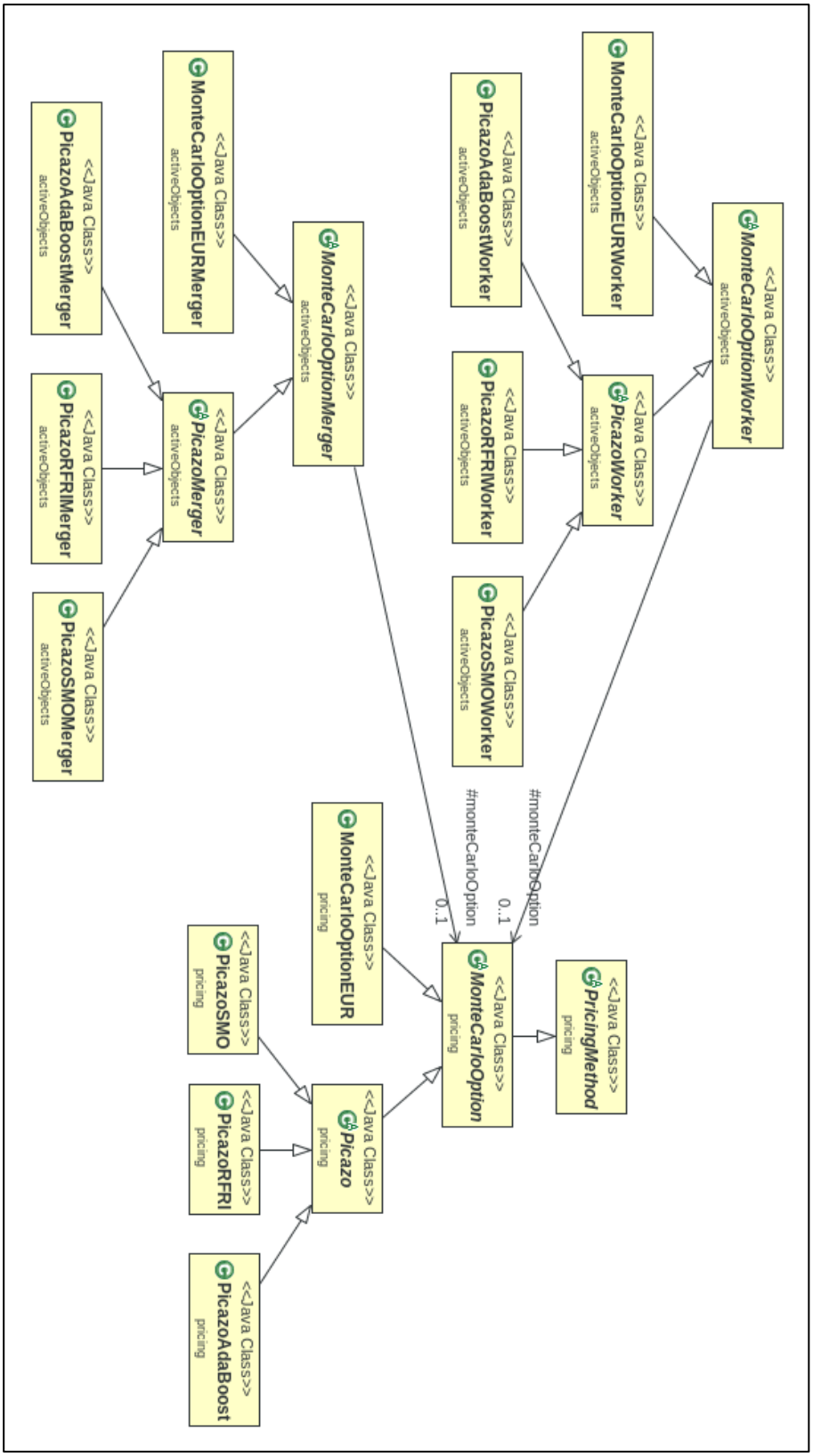

Figure 39 Diagramme de classes des packages pricing et activeObjects.

Notre application financière repose sur le pricing d'instruments financiers de type optionnel par méthode de Monte Carlo, bien que d'autres méthodes et instruments puissent l'étoffer dans le futur, en y ajoutant les classes nécessaires et en redéfinissant certaines méthodes. La classe `Portfolio` comporte une liste d'instruments. Tous les instruments héritent de la classe abstraite `Instrument`, en particulier `Asset` et la classe abstraite `Option`, elle-même héritée dans `OptionUS` et `OptionEUR`. L'intérêt d'implémenter une classe intermédiaire `Option` est de factoriser les attributs et méthodes propres au contrat optionnel : payoff, sous-jacents, strike, maturité,.. L'ensemble de ces classes résident dans le package instrument (**Figure 38**).

Toutes les méthodes de calcul héritent directement ou indirectement de la classe abstraite `PricingMethod` (**Figure 39**), et résident dans le package pricing. Ces classes intègrent les paramètres de pricing ainsi que les fonctions non distribuées, c.-à-d. non appelées par un `worker`/`merger`. Cependant, c'est à partir de ces classes que les `workers`/`merger` sont créés et orchestrés. `MonteCarloOptionEUR`, `PicazoAdaBoost`, `PicazoSMO`, `PicazoRFRI` sont les classes les plus spécialisées, et intègrent chacune en outre une méthode `getWorkersParams()` utilisée lors de la création des `workers`. Par exemple, l'appel à `getWorkersParams()` de `PicazoAdaBoost` permet ensuite de créer des `workers` par l'instanciation de `PicazoAdaBoostWorker`.

La difficulté de travailler sur une application orientée objet avec une parallélisation inter/intra noeud, est d'assurer une bonne lisibilité du code tout en maintenant son extensibilité. La librairie ProActive nous facilite la tâche en masquant les mécanismes de distribution, nous permettant de nous concentrer uniquement sur l'aspect fonctionnel. Un objet java distant, est donc simplement une instance des classes `xxxWorker` et `xxxMerger`, avec `xxx` le nom d'une méthode de pricing (par exemple `PicazoWorker`, `PicazoAdaBoostWorker`,..). Toutes ces classes sont dans le package activeObject. Ces classes et l'ensemble de leurs membres doivent êtres sérialisables pour pouvoir transiter et recevoir des appels de méthode sur le réseau.

La classe `MonteCarloOptionMerger` inclut principalement les méthodes de *merge* (fusion des résultats des workers) et d'orchestration de haut niveau : `getNbWorkersFinished()`, `getNbSimusDone()`,.. En effet, certaines informations sur l'état d'avancement du travail des `workers`, ont ensuite besoin d'être remontées par le `merger` au thread principal. `PicazoMerger` comporte les méthodes relatives à l'orchestration et à la classification : `trainNewClassifier(…)`, `broadcastAndSerializeClassifier(…)`,… Enfin, les classes les plus spécialisées comme `PicazoAdaBoostMerger`, `PicazoSMOMerger` intègrent l'entrainement à proprement parler des classificateurs, sauf pour `PicazoRFRIMerger` qui dans sa méthode

`trainNewClassifier(…)` délègue la tâche aux `workers` avant de récolter et fusionner les sous-classificateurs.

La classe `PicazoWorker` inclut les méthodes de calcul des instances d'entrainement et d'estimation de prix. Celle-ci introduit aussi l'initialisation de la majorité des données des kernels. `PicazoAdaBoostWorker`, `PicazoSMOWorker`, `PicazoRFRIWorker` intègrent la définition des données propres à chaque classificateur : entrainement, sérialisation. L'ensemble des paramètres de kernels, hérités ou non, sont affectés au kernel de la méthode de calcul dans ces classes.

La classe d'un `worker`/`merger` doit pouvoir accéder aux informations de deux autres classes. Par exemple, `PicazoWorker` doit à la fois hériter de la classe `MonteCarloOptionWorker`, mais aussi avoir accès aux paramètres de pricing de Picazo. L'héritage multiple en Java n'est pas possible, et ne serait pas tout à fait logique, car le `worker` contribuant à la mise en œuvre d'une méthode de pricing n'est pas une instance d'une classe héritant de `PricingMethod`, mais y est seulement lié. Ainsi, une instance de classe `xxx` héritant de `PricingMethod`, est déclarée comme membre des classes `xxxWorker` et `xxxMerger`.

Les premières versions de notre logiciel exploitaient un unique GPU pour valoriser l'option américaine et européenne. L'intégration des routines de l'API ProActive pour exploiter une architecture distribuée, a nécessité la séparation du code fonctionnel et de déploiement, pour gagner à la fois en lisibilité et en extensibilité. L'utilisation d'un langage orienté objet nous a permis de bénéficier de l'héritage et du polymorphisme, pour limiter toute redondance de code : initialisation des `workers`, simulations de MC finales selon chaque classificateur, entrainement des classificateurs,… C'est d'ailleurs sur ce dernier point que l'intégration de l'entrainement parallèle des forêts aléatoires (contrairement aux autres entrainements séquentiels) n'a pas eu d'incidence sur l'architecture logicielle. La valorisation d'un type d'instrument requiert un kernel spécifique, et c'est aussi le cas pour chaque type de classification. Seules certaines parties du code non spécifiques à la valorisation sont factorisées et réutilisées (`Correlate`, `BoxMuller`, `geometricBrownian`, …) d'un kernel à l'autre. Les kernels exécutés par les `workers` de type `PicazoxxxWorker` se différencient par leurs paramètres et la fonction de classification appelée. Concernant la prise en charge des CPUs comme ressource de calculs, cela n'a pas eu d'impact notable sur l'architecture de par la portabilité d'OpenCL. Seule la partie dédiée à la calibration des kernels, et celle de monitoring (requise pour le load balancing) s'en distingue. Bien qu'initialement nous avions prévu de calculer uniquement le prix spot (actuel) d'un instrument, le calcul de la VaR a nécessité de paramétrer l'instant t de valorisation (initialement t_0) pour l'exécuter à un horizon donné.

3. Déploiement et programmation en OpenCL

Depuis l'interface graphique, pour composer un portefeuille d'options, les paramètres de chaque instrument et ceux de leur méthode de valorisation associée, doivent être renseignés avant d'être enregistrés. D'un point de vue implémentation, on distingue aussi les paramètres de l'instrument, et ceux de la méthode de valorisation

```
Portfolio ptf = new Portfolio(…);
ArrayList<PricingMethod> pricingMethods = new ArrayList<PricingMethod>();
```

On associe un `Instrument` à une `PricingMethod`, en les intégrant respectivement dans un `Portfolio` et une liste de `PricingMethod` dans le même ordre. Par exemple si le premier instrument est une option américaine, et qu'on veut la valoriser par la méthode de Picazo via la classification AdaBoost, on procède ainsi

```
ptf.add(new OptionUS(…));
pricingMethods.add(new PicazoAdaBoost(…);
```

L'exécution d'une mesure, que ce soit la valorisation d'un portefeuille, ou le calcul de sa VaR (`Price`, `MonteCarloVaR`), s'effectue ainsi : on instancie un objet représentant la mesure à effectuer, après quoi on se base sur un autre objet capable d'initier les déploiements nécessaires selon les ressources présentes puis de lancer les opérations correspondants à cette mesure:

```
PortfolioMeasure price = new Price(ptf, pricingMethods, …);
PortfolioMeasure mCVaR = new MonteCarloVaR(ptf, pricingMethods, …);
Orchestrator.orchestrate(price);
Orchestrator.orchestrate(mCVaR);
```

`Orchestrator.orchestrate(…)` agrège les appels de fonctions permettant de 1. Détecter les nœuds du cluster 2. Détecter les ressources (GPUs/CPUs) de chaque nœud 3. Estimer les paramètres des kernels de chaque ressource (Chapitre IV.4.3 et Chapitre V.3.1) 4. Estimer les durées des kernels sur chaque ressource (Chapitre V.3.2) 5. Etablir le schéma de répartition des valorisations des instruments, selon qu'il s'agit de valoriser le portefeuille ou calculer la VaR (Chapitre VI.2 et Chapitre VI.3.2) 6. Exécuter le calcul.

Un objet actif dit `launcher` est exécuté pour chaque instrument, et distribue son pricing en exécutant autant d'objets actifs (`workers`) que de devices spécifiés dans le schéma de répartition pour cet instrument.

Les objets actifs `workers` exécutés par le `launcher`, interviennent dans la distribution d'un pricing. Par exemple, la valorisation d'une option américaine, via les forêts aléatoires, sollicite des objets actifs de type `PicazoRFRIWorker`.

Les `workers` ont à charge la création des instances d'entrainement, mais aussi la sérialisation puis le transfert de chaque classificateur vers le device. Nous détaillons ci-après la manière dont nous avons conçu cette sérialisation dans le cas le moins simple (forêts aléatoires).

3.1. Détails de la sérialisation d'une forêt aléatoire (Java)

Cette sérialisation s'opère d'arbre en arbre. Chaque appel à `treeToArrays(…)` (sur un arbre `currentTree`) met à jour l'indice de position de l'arbre sérialisé, dans le tableau de la forêt sérialisée

```
int[] iteration = new int[]{-1};
for (…; i < forestSize; …)
{
	treesPosInForest[treesPosInForestIndex++] = nodeInForest[0]+1;
	PicazoRFRI.treeToArrays(currentTree, …, nodeInForest, -1,
NodeType.parent, iteration);
}
```

La sérialisation d'un arbre s'effectue dans les tableaux passés en paramètres de la méthode `treeToArrays`

```
public static void treeToArrays(
	Tree tree,
	int[] attributes,
	double[] splitPointsOrClassDistributions0,
	double[] classDistributions1,
	int[] leftNodesInForest,
	int[] rightNodesInForest,
	int[] nodeInForest,
	int parentNodeInTree,
	NodeType nodeType,
	int[] iteration)
{
	iteration[0]++;
	int currentIteration = iteration[0];
```

On calcule l'indice du nœud courant

```
	nodeInForest[0]++;
```

Le nœud est potentiellement une feuille, traduit par son fils gauche et son fils droit à -1

```
	leftNodesInForest[currentIteration] = -1;
```

```
rightNodesInForest [currentIteration] = -1;
```

On indique que le nœud est le fils gauche ou droit de son parent

```
if (nodeType == NodeType.leftChild)
       leftNodesInForest[parentNodeInTree] =
              nodeInForest [0];
else if (nodeType == NodeType.rightChild)
       rightNodesInForest[parentNodeInTree] =
              nodeInForest[0];
```

On sérialise les attributs du nœud

```
attributes[currentIteration] = tree.m_Attribute;
```

Dans le cas où le nœud n'est pas une feuille

```
if (tree.m_Attribute > -1)
{
```

On sérialise la splitvalue du nœud (valeur de comparaison)

```
       splitPointsOrClassDistributions0[currentIteration] =
              tree.m_SplitPoint;
       classDistributions1[currentIteration] = -1;
       int parentNodeNotByRef = nodeInForest[0];
```

On procède ensuite récursivement sur les fils du nœud

```
       treeToArrays(tree.m_Successors[0],
                    attributes,
                    splitPointsOrClassDistributions0,
                    classDistributions1,
                    leftNodesInForest,
                    rightNodesInForest,
                    nodeInForest,
                    parentNodeNotByRef,
                    NodeType.leftChild,
                    iteration);
       treeToArrays(tree.m_Successors[1],
                     attributes,
                     splitPointsOrClassDistributions0,
                     classDistributions1,
                     leftNodesInForest,
                     rightNodesInForest,
                     nodeInForest,
                     parentNodeNotByRef,
                     NodeType.rightChild,
                     iteration);
```

Si le nœud est une feuille, on sérialise sa valeur de distribution (indiquant la probabilité pour une instance à classifier, de parcourir l'arbre, de la racine jusqu'à cette feuille)

```
        } else{
                splitPointsOrClassDistributions0[currentIteration] =
                        tree.m_ClassDistribution[0];
                classDistributions1 [currentIteration] =
                        tree.m_ClassDistribution[1];
        }
}
```

3.2. Détails des kernels (OpenCL)

Notre moteur de calcul englobe autant de kernels différents que de méthodes de valorisation. Le kenel MonteCarloOptionEUR.cl valorise l'option européenne et les kernels PicazoXXX.cl valorisent l'option américaine.

Nous présentons ici PicazoAdaBoost.cl, qui se distingue de PicazoSVM.cl, PicazoRFRI.cl par les paramètres du kernel et l'appel de la méthode de classification. Ces kernels sont aussi bien exécutés pour simuler les nb_cont simulations de chaque valeur de continuation, que pour les nb_MC simulations finales. Les nombreuses exécutions de kernel ne nous permettent pas de découper davantage les calculs en kernels plus spécifiques, de par les nombreux transferts entre CPUs et GPUs que cela induirait. Nous détaillons ici le kernel et ses arguments.

```
__kernel void pricing (
        int basketSize_reg,
        int basketSizeNextEven_reg,
        int prevPow2_reg,
        char payOffType_reg,
        char optionType_reg,
        __local double* array_loc,
        __local double* array2_loc,
        __constant int* kernelSimuStartAsNbDt_const,
        __constant int* maturityAsNbDt_const,
        __constant float* dt_const,
        __constant float* riskFreeRate_const,
        __constant float* strike_const,
        __constant float* volatRates_const,
        __constant float* dividRates_const,
        __constant float* choleski_const,
        __global char* prng_glob,
        __global long* seedPRNG_glob,
        __global int* nbMC_glob,
        __global int* numIterPerf_glob,             // Paramètres AdaBoost …
        __global int* firstStumpIndexes_glob,       // …
        __global int* attIndexes_glob,              // …
        __global double* betas_glob,                // …
        __global double* distributions_glob,        // …
        __global double* splitPoints_glob,          // …
        __global int* nbSimusDone_glob,
        __global double* gauss_glob,
        __global double* gaussCorr_glob,
        __global double* st_glob,
        __global double* stNext_glob,
```

```
        __global double* sumsDone_glob,
        __global double* sumsXXDone_glob)
{
        char predictedLabel_reg = '0';
        char lastSimus_reg = 'n';
        int tAsNbDt_reg = kernelSimuStartAsNbDt_const[0];
        int nbSimuWorkgroupDone_reg = 0;
        int nbSimusDone_reg = 0;
        int i_reg;
        double prePayOff_reg;
        double discPayOff_reg;
        double sum_reg = 0.0f;
        double sumXX_reg = 0.0f;
```

Comme on a pu le constater ci-dessus, nous privilégions les registres (_reg), la mémoire partagée (_global), et la mémoire globale cachée (_const), pour les variables utilisées intensivement. Par ailleurs, nous distribuons uniformément les simulations sur les workgroups, comme le montrent les 2 instructions qui suivent

```
        int nbMCworkGroup_reg = nbMC_glob[0] / get_num_groups(0);
        if (get_group_id(0) < nbMC_glob[0] % get_num_groups(0))
                nbMCworkGroup_reg++;
```

On initialise ensuite le générateur pseudo aléatoire MWC64X si celui-ci est désigné.

```
        mwc64x_state_t rng;
        if (prng_glob[0]=='m')
                MWC64X_SeedStreams(&rng, (ulong) seedPRNG_glob[0],
                        1000000000);
```

Le kernel itère dans un premier temps, sur un **nombre de pas de temps** calculé (Chapitre IV.3), et dans un second temps, sur le **nombre de simulations restantes** (pour atteindre nb_cont ou nb_MC). On simule ainsi un nombre exact et identique de trajectoires pour chaque valeur de continuation, sans introduire de biais dans le prix de l'option. $nbDtReductionOrNbSimus\_reg$ est utilisée comme compteur dans ces deux phases

```
        int nbDtReductionOrNbSimus_reg =
                maturityAsNbDt_const[0] - kernelSimuStartAsNbDt_const[0];
        array_loc[get_local_id(0)] = 0.0f;
```

st_glob représente les prix initiaux du panier en t et $stNext\_glob$ les prix du panier simulés à chaque instant entre t et T. Dès lors qu'un temps d'arrêt est atteint, $stNext\_glob$ est réinitialisée à st_glob.

```
        resetToSt (     get_global_id(0),
                        get_global_size(0),
                        basketSize_reg,
```

```
                st_glob,
                stNext_glob);
```

Détaillons à présent le corps de la première boucle : tant que les simulations de chaque workgroup n'ont pas été toutes exécutées

```
while(nbSimuWorkgroupDone_reg < nbMCworkGroup_reg)
{
```

Si le nombre maximal de simulations pouvant être exécutées permet d'atteindre $nbMCworkgroup\_reg$ à la prochaine itération, les prochaines et dernières itérations portent sur les simulations restantes et non les pas de temps

```
        if(nbSimuWorkgroupDone_reg +
                nbDtReductionOrNbSimus_reg * get_local_size(0) >
                        nbMCworkGroup_reg)
        {
                nbDtReductionOrNbSimus_reg =
                        (nbMCworkGroup_reg -nbSimuWorkgroupDone_reg) /
                                get_local_size(0);

                if (get_local_id(0) <
                        (nbMCworkGroup_reg -nbSimuWorkgroupDone_reg) %
                                get_local_size(0))
                        nbDtReductionOrNbSimus_reg++;

                lastSimus_reg = 'y';
        }
```

Concernant la seconde boucle imbriquée dans la première : tant qu'un nombre de pas de temps ou de simulations n'a pas été atteint,

```
        i_reg = 0;
        while ( i_reg < nbDtReductionOrNbSimus_reg)
        {
                tAsNbDt_reg++;
```

on génère autant de variables aléatoires que d'actifs dans le panier pour chaque thread, en prenant la valeur paire immédiatement supérieure, si la taille du panier est impaire, pour la transformation de Box-Muller

```
                // PRNGs
                if (prng_glob[0]=='m')
                        MWC64X(get_global_id(0),
                                get_global_size(0),
                                basketSizeNextEven_reg,
                                &rng,
                                gauss_glob);

                else if (prng_glob[0]=='x')
                        XOR128(get_global_id(0),
                                get_global_size(0),
                                basketSizeNextEven_reg,
                                seedPRNG_glob,
```

```
                gauss_glob);

else if (prng_glob[0]=='p')
        PMLCG(  get_global_id(0),
                get_global_size(0),
                basketSizeNextEven_reg,
                seedPRNG_glob,
                gauss_glob);
```

La transformation de Box-Muller génère des variables aléatoires normalement distribuées à partir de variables aléatoires uniformément distribuées.

```
BoxMuller (     get_global_id(0),
                get_global_size(0),
                basketSizeNextEven_reg,
                gauss_glob);
```

La décomposition de Choleski de la matrice de corrélation des actifs du panier s'opère côté CPU. Le produit des gaussiennes et des facteurs de Choleski assure la corrélation des actifs.

```
Correlate (     get_global_id(0),
                get_global_size(0),
                basketSize_reg,
                choleski_const,
                gauss_glob,
                gaussCorr_glob);
```

Le mouvement brownien géométrique simulant les actifs du panier est fonction des paramètres de pricing et des gaussiennes corrélées.

```
geometricBrownian (get_global_id(0),
                get_global_size(0),
                basketSize_reg,
                dt_const,
                riskFreeRate_const,
                volatRates_const,
                dividRates_const,
                gaussCorr_glob,
                stNext_glob);
```

Nous calculons la moyenne arithmétique ou géométrique, selon le paramètre de pricing.

```
prePayOff_reg =
        prePayOff (get_global_id(0),
                get_global_size(0),
                basketSize_reg,
                payOffType_reg,
                stNext_glob);
```

Le classificateur, ici AdaBoost, prend en paramètre les structures représentant le classificateur en t, ainsi que les prix des actifs et le prepayoff. Cet appel ne s'effectue que pour $t < T$

```
if (tAsNbDt_reg != maturityAsNbDt_const[0])
{
   predictedLabel_reg =
     classifyInstance(
       get_global_id(0),
       get_global_size(0),
       basketSize_reg,
       prePayOff_reg,
       &numIterPerf_glob[tAsNbDt_reg - 1],
       &firstStumpIndexes_glob[tAsNbDt_reg - 1],
       attIndexes_glob,
       betas_glob,
       distributions_glob,
       splitPoints_glob,
       stNext_glob);
}
```

Si la trajectoire est terminée,

```
if (predictedLabel_reg == '1'
       || tAsNbDt_reg == maturityAsNbDt_const[0])
{
```

on enregistre ici les longueurs de trajectoire pour calibrer par la suite le nombre de pas de temps devant précéder chaque réduction intermédiaire

```
   if (nbSimuWorkgroupDone_reg == 0)
      array_loc[get_local_id(0)] =
         tAsNbDt_reg - kernelSimuStartAsNbDt_const[0];
```

On calcule le payoff actualisé

```
   discPayOff_reg =
      payOff (optionType_reg,
              prePayOff_reg,
              strike_const)
      * native_exp(-riskFreeRate_const[0]
                 * (tAsNbDt_reg -
                     kernelSimuStartAsNbDt_const[0])
                 * dt_const[0]);
```

avant de l'accumuler en registre

```
   sum_reg += discPayOff_reg;
   sumXX_reg += discPayOff_reg * discPayOff_reg;
```

Les prix sont réinitialisés pour les prochaines simulations

```
   resetToSt ( get_global_id(0),
```

```
                                get_global_size(0),
                                basketSize_reg,
                                st_glob,
                                stNext_glob);

                    tAsNbDt_reg = kernelSimuStartAsNbDt_const[0];
                    nbSimusDone_reg++;
                    predictedLabel_reg = '0';
```

Dans ce cas *i_reg* désigne le nombre de simulations

```
                    if(lastSimus_reg == 'y')
                        i_reg++;
                }
```

Ici *i_reg* désigne le nombre de pas de temps

```
                if(lastSimus_reg == 'n')
                i_reg++;

            } // FIN BOUCLE 1
```

On calcule en parallèle pour chaque workgroup, le maximum des longueurs des trajectoires précédemment enregistrées dans *array_loc*

```
            if (nbSimuWorkgroupDone_reg == 0)
            {
                reductionMax(prevPow2_reg, array_loc);
                nbDtReductionOrNbSimus_reg = array_loc[0];
                barrier(CLK_LOCAL_MEM_FENCE);
            }
```

Cette partie calcule en parallèle, le nombre de simulations effectuées au sein de chaque workgroup, et l'écrit en registre. La condition de la première boucle sera ainsi réévaluée avec cette nouvelle valeur.

```
            array_loc[get_local_id(0)] = nbSimusDone_reg;
            reductionSum (prevPow2_reg,array_loc);
            nbSimuWorkgroupDone_reg += array_loc[0];

            barrier(CLK_LOCAL_MEM_FENCE);
            nbSimusDone_reg = 0;

        } // FIN BOUCLE 1
```

On calcule en parallèle au sein de chaque workgroup, la somme des payoffs actualisés et la somme des carrés des payoffs actualisés. Ces résultats serviront côté CPU, à calculer leur espérance et variance.

```
        array_loc[get_local_id(0)] = sum_reg;
        reductionSum (prevPow2_reg, array_loc);
        array2_loc[get_local_id(0)] = sumXX_reg;
        reductionSum (prevPow2_reg, array2_loc);
```

On termine en copiant en mémoire globale les résultats

```
        if (get_local_id(0) == 0)
        {
                nbSimusDone_glob[get_group_id(0)] = nbSimuWorkgroupDone_reg;
                sumsDone_glob[get_group_id(0)] = array_loc[0];
                sumsXXDone_glob[get_group_id(0)] = array2_loc[0];
        }
}
```

4. Conclusion

Nous avons détaillé dans ce chapitre certains aspects conceptuels et techniques de notre application, dans laquelle plusieurs paradigmes cohabitent : SIMT, approche maitre-esclave. L'utilisation d'un langage orienté objet permet d'assurer à la fois la lisibilité et l'extensibilité du code, qui s'exécute dynamiquement selon deux niveaux de parallélisation via les librairies OpenCL et ProActive. Les difficultés rencontrées côté Java sont majoritairement liées à l'architecture et algorithmique. Côté OpenCL, celles-ci sont principalement d'ordre algorithmique et technique. Ce dernier point est directement dû au coût occasionné lors de l'intégration d'un tel algorithme en OpenCL : quantité limitée de mémoire locale et globale cachée, nombre de registres disponibles, surcoût des synchronisations implicites (warp divergence) et explicites,.. Nous imposant un compromis entre lisibilité et optimisation du code.

Chapitre VIII. Conclusion

1. Bilan

Les différentes étapes successives de cette thèse traitent de deux problématiques majeures en finance : le pricing de l'option américaine sur panier, et le calcul de la MC VaR. Ces deux problèmes combinés donnent naissance à un troisième : le calcul de la MC VaR d'un portefeuille d'options américaines sur panier. Ces travaux traitent de la réduction du temps de calcul par une approche algorithmique, permettant de déléguer les calculs sur différentes ressources : CPU, GPU, cluster de CPUs, cluster de GPUs, cluster hybride de GPUs/CPUs, dans un environnement homogène ou hétérogène. Le pricing d'option américaine sur panier d'actifs pose les premières pierres du problème suivant, dont la solution apportée peut se généraliser à d'autres instruments financiers : valoriser un instrument financier à payoff non linéaire par simulations de MC, tout en assurant un temps d'exécution bas, en déportant le calcul sur GPU.

L'implémentation de l'algorithme de Picazo sur GPU a ouvert la porte à d'autres optimisations possibles, liées aux spécificités de l'architecture d'un GPU : calibrations dynamique d'un kernel, approche SIMT d'une partie de l'algorithme, classification en OpenCL. Les différents efforts réalisés ont permis de diviser par deux le temps de calcul d'une option américaine sur panier, en utilisant un unique GPU, comparé à une version 64 cœurs CPU (Chapitre IV.5). L'évolution naturelle de ces travaux était alors de permettre à l'application de se déployer dynamiquement sur une architecture distribuée hybride ou non, hétérogène ou non. Les nouveaux axes se sont dès lors portés sur la redistribution de l'algorithme d'une part, et la définition d'une stratégie de répartition des tâches d'autre part, sans laquelle le gain de performance serait fortement atténué (Chapitre V.3.2). Les résultats obtenus (Chapitre V.3.3) nous ont encouragés à étendre la complexité du problème sur plusieurs niveaux : d'une part, ne pas considérer uniquement la valorisation d'une option américaine sur panier, mais celle d'un portefeuille d'options européennes ou américaines sur panier; d'autre part, estimer la MC VaR d'un tel portefeuille. Les nouveaux enjeux se sont dès lors tournés vers la réduction du temps de calcul lors de la valorisation multiple d'instrument : réutilisation de la frontière d'exercice lors de la revalorisation de l'option américaine, réorganisation des calculs pour l'estimation de la VaR. Toutes ces optimisations rendent possible l'estimation de la VaR d'un portefeuille dont l'estimation est complexe (Chapitre VI.3.2). Les difficultés techniques rencontrées lors de la réalisation du moteur de calcul ont été de maintenir la lisibilité et la modularité du code, tout en manipulant les différentes librairies à l'origine du parallélisme intra

nœud et inter nœud. Les paradigmes de la programmation orienté objet nous ont grandement aidé.

2. Perspectives

De par les différents domaines abordés dans ces travaux, les perspectives sont variées et nombreuses.

Comme évolution directe de notre application, nous pourrions intégrer de nouveaux instruments (obligation, future, ..) et l'enrichir de nouveaux modèles stochastiques : volatilité et taux sans risque stochastiques, modèle à saut,... Aussi, d'autres mesures de risque pourraient y être intégrées : VaR conditionnelle (CVaR), expected short fall ; ainsi que les sensibilités (grecs). L'intégration des fonctions de classification en OpenCL seraient sollicitées pour l'analyse technique en temps réel : les données de marché seraient récupérées à une fréquence donnée pour être traitées par un kernel, ou possiblement plusieurs exécutés sur un cluster. Notre application permet au sein d'un même device, d'effectuer autant de classifications sur des paramètres différents, que de threads. Ceci pourrait servir à analyser simultanément les tendances des différents actifs constituant le portefeuille, ou plus généralement analyser différents indicateurs de marché.

Notre application exploite les CPUs et GPUs comme unité de calcul mais pourrait prendre en charge les FPGA comme le permet OpenCL [118].

Bibliographie

[1] M. Giles, «Computational Finance on GPUs,» Intel Finance Forum, 2009.

[2] Cognizant Technology Solutions, «How Cloud Computing Impacts Trade Finance,» 2011. [En ligne]. Available: http://www.cognizant.com/InsightsWhitepapers/How-Cloud-Computing-Impacts-Trade-Finance.pdf.

[3] G. Tom, «Faster, faster … HPC in financial services,» 2011. [En ligne]. Available: http://www.bankingtech.com/47927/Faster-faster-HPC-in-financial-services/.

[4] NVIDIA, «NVIDIA Tesla GPUs used by J.P. Morgan Run Risk Calculations in Minutes, Not Hours,» 2011. [En ligne]. Available: http://www.nvidia.co.uk/object/nvidia-tesla-jpmorgan-press-20110804-uk.html.

[5] V. Doan, «Grid computing for Monte Carlo based intensive calculations in financial derivative pricing applications,» 2010.

[6] M. Benguigui et F. Baude, «Towards parallel and distributed computing on GPU for American basket option pricing,» chez *International Workshop on GPU Computing in Cloud in conjunction with 4th IEEE international conference on Cloud Computing Technology and Science*, Taipei, 2012.

[7] M. Benguigui et F. Baude, «Fast American Basket Option Pricing on a multi-GPU Cluster,» chez *Proceedings of the High Performance Computing Symposium*, Tampa, 2014.

[8] J. Picazo, «American Option Pricing: A Classification-Monte Carlo (CMC) Approach,» chez *Monte Carlo and Quasi-Monte Carlo Methods*, 2001.

[9] M. Kowalski, «La loi des grands nombres et le théorème de la limite centrale,» [En ligne]. Available: http://webpages.lss.supelec.fr/perso/kowalski/downloads/Enseignement/2008_2009/MASS/MSHD01/coursLGN.pdf.

[10] M. Dion et P. L'Ecuyer, «American option pricing with randomized quasi-Monte Carlo simulations,» chez *Simulation Conference (WSC), Proceedings of the 2010 Winter*, 2010.

[11] P. Bratley, B. Fox et H. Niederreiter, «Implementation and Tests of Low Discrepancy Sequences,» *Transactions on Modeling and Computer Simulation,*

vol. 2, n°13, pp. 195-213, 1992.

[12] Y. Freund et R. Schapire, «A Short Introduction to Boosting,» *Journal of Japanese Society for Artificial Intelligence,* vol. 14, n°15, pp. 771-780, 1999.

[13] J. Platt, «Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines,» 1998.

[14] L. Breiman, «Random Forests,» *Machine Learning,* vol. 45, n°11, pp. 5-32, #oct# 2001.

[15] Michael Wolfe, «Understanding the CUDA Data Parallel Threading Model A Primer,» 2012. [En ligne]. Available: https://www.pgroup.com/lit/articles/insider/v2n1a5.htm.

[16] NVIDIA, «NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110,» [En ligne]. Available: http://www.nvidia.fr/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf.

[17] NVIDIA, «CUDA C Programming Guide,» [En ligne]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[18] Khronos, «The OpenCL Specification,» [En ligne]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[19] Intel, «Writing Optimal OpenCL™ Code with Intel® OpenCL SDK - Performance Guide,» 2011. [En ligne]. Available: https://software.intel.com/sites/default/files/m/d/4/1/d/8/Writing_Optimal_OpenCL_28tm_29_Code_with_Intel_28R_29_OpenCL_SDK.pdf.

[20] F. Black et M. Scholes, «The Pricing of Options and Corporate Liabilities,» *Journal of political economy,* vol. 81, n°13, p. 637, 1973.

[21] R. Merton, «Theory of rational option pricing,» *The Bell Journal of Economics and Management Science,* vol. 4, n°11, pp. 141-183, 1973.

[22] R. Geske et H. Johnson, «The American Put Option Valued Analytically,» *The Journal of Finance,* vol. 39, n°15, pp. 1511-1524, 1984.

[23] L. MacMillan, «An Analytic Approximation for the American Put Option,» chez *Advances in Futures and Options Research*, vol. 1, 1986, pp. 119-139.

[24] G. Barone-Adesi et R. E. Whaley, «Efficient Analytic Approximation of American Option Values,» *The Journal of Finance,* vol. 42, p. 301–320, 1987.

[25] K. Joon, «The Analytic Valuation of American Options,» *Review of Financial Studies,* vol. 3, n°14, pp. 547-572, 1990.

[26] S. Jacka, «Optimal stopping and the American put,» *Mathematical Finance,* vol. 1, pp. 1-14, 1991.

[27] P. Carr, R. Jarrow et R. Myneni, «Alternative characterizations of American put options,» *Mathematical Finance,* vol. 2, n°12, pp. 87-106, 1992.

[28] M. Broadie et J. Detemple, «American option valuation: new bounds, approximations, and a comparison of existing methods,» *Review of Financial Studies,* vol. 9, n°14, pp. 1211-1250, 1996.

[29] N. Ju, «Pricing by American option by approximating its early exercise boundary

as a multipiece exponential function,» *Review of Financial Studies,* vol. 11, n°13, pp. 627-646, 1998.

[30] D. Bunch et H. Johnson, «The American Put Option and Its Critical Stock Price,» *The Journal of Finance,* vol. 55, n°15, pp. 2333-2356, 2000.

[31] P. Carr, «Randomization and the American Put,» *Review of Financial Studies,* vol. 11, n°13, pp. 597-626, 1998.

[32] S. Zhu, «An exact and explicit solution for the valuation of American put,» *Quantitative Finance,* vol. 6, n°13, pp. 229-242, 2006.

[33] J. Cox, S. Ross et M. Rubinstein, «Option Pricing: A Simplified Approach,» *Journal of Financial Economics,* vol. 7, pp. 229-263, 1979.

[34] R. Rendleman et B. Bartter, «Two-State Option Pricing,» *The Journal of Finance,* vol. 34, n°15, pp. 1093-1110, 1979.

[35] K. Amin et A. Khanna, «Convergence of American option values from discrete to continuous-time financial models,» *Mathematical Finance,* vol. 4, n°14, pp. 289-304, 1994.

[36] P. Michael, «Option Pricing: The American Put,» *The Journal of Business,* vol. 50, n°11, pp. 21-36, 1977.

[37] P. Phelim, «A Lattice Framework for Option Pricing with Two State Variables,» *The Journal of Financial and Quantitative Analysis,* vol. 23, pp. 1-12, 1988.

[38] B. Phelim, E. Jeremy et G. Stephen, «Numerical Evaluation of Multivariate Contingent Claims,» *The Review of Financial Studies,* vol. 2, n°12, pp. 241-250, 1989.

[39] M. Rubinstein, «Rainbow Options,» *Risk,* vol. 7, pp. 67-71, 1994.

[40] R. Breen, «The Accelerated Binomial Option Pricing Model,» *The Journal of Financial and Quantitative Analysis,* vol. 26, n°12, pp. 153-164, 1991.

[41] S. Figlewski et B. Gao, «The Adaptive Mesh Model: A New Approach to Efficient Option Pricing,» *Journal of Financial Economics,* vol. 53, p. 313–351, 1999.

[42] S. Heston et G. Zhou, «On the Rate of Convergence of Discrete-Time Contingent Claims,» *Mathematical Finance,* vol. 10, n°11, pp. 53-75, 2000.

[43] M. Brennan et E. Schwartz, «The Valuation of American Put Options,» *The Journal of Finance,* vol. 32, pp. 449-462, 1977.

[44] P. Jaillet, D. Lamberton et B. Lapeyre, «Variational inequalities and the pricing of American options,» *Acta Applicandae Mathematicae,* vol. 21, n°13, p. 263–289, 1990.

[45] G. Courtadon, «A more accurate finite difference approximation for the valuation of options,» *Journal of Financial and Quantitative Analysis,* vol. 17, n°15, p. 697–703, 1982.

[46] P. Wilmott, J. Dewynne et S. Howison, Option Pricing: Mathematical Models and Computation, Oxford Financial Press, 1993.

[47] P. A. Forsyth et K. R. Vetzal, «Quadratic convergence for valuing American

options using a penalty method,» *SIAM Journal on Scientific Computing,* vol. 23, n°16, p. 2095–2122, 2002.

[48] S. Ikonen et J. Toivanen, «Operator splitting methods for American option pricing,» *Applied mathematics letters,* vol. 17, n°17, pp. 809-814, 2004.

[49] K. Ito et K. Kunisch, «Parabolic variational inequalities: The Lagrange multiplier approach,» *Journal de Mathématiques Pures et Appliquées,* vol. 85, n°13, p. 415–449, 2006.

[50] A. Mayo, «High Order Accurate Implicit Method for Valuing American Options,» *The European Journal of Finance,* vol. 10, n°13, pp. 212-237, 2004.

[51] C. Oosterlee, C. Leentvaar et X. Huang, «Accurate American option pricing by grid stretching and high order finite differences,» Delft Institute of Applied Mathematics, Delft University of Technology, Delft, 2005.

[52] D. Tangman, A. Gopaul et M. Bhuruth, «Numerical pricing of options using high-order compact finite difference schemes,» *Journal of Computational and Applied Mathematics,* vol. 218, n°12, p. 270–280, 2008.

[53] D. Kelly, «Valuing and Hedging American Put Options Using Neural Networks,» 1994.

[54] N. Chidambaran, J. Triqueros et C. Lee, «Option Pricing Via Genetic Programming,» chez *Evolutionary Computation in Economics and Finance*, Physica-Verlag HD, 2002, pp. 383-397.

[55] Z. Yin, A. Brabazon et C. O'Sullivan, «Adaptive genetic programming for option pricing,» chez *the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, 2007.

[56] C. Keber et M. Schuster, «Option valuation with generalized ant programming,» chez *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 2002.

[57] S. Kumar, R. Thulasiram et P. Thulasiraman, «Ant Colony Optimization for Option Pricing,» chez *Natural Computing in Computational Finance*, Springer Berlin Heidelberg, 2009, pp. 51-73.

[58] B. Sharma, R. Thulasiram et P. Thulasiraman, «Normalized particle swarm optimization for complex chooser option pricing on graphics processing unit,» *The Journal of Supercomputing,* vol. 66, n°11, pp. 170-192, 2013.

[59] P. Boyle, «Options: A Monte Carlo approach,» *Journal of Financial Economics,* vol. 4, n°13, pp. 323-338, 1977.

[60] P. Bossaerts, «Simulation Estimators of Optimal Early Exercise,» 1989.

[61] J. A. Tilley, «Valuing American Options in a Path Simulation Model,» *Transactions of the Society of Acturies,* vol. 45, pp. 83-104, 1993.

[62] D. Grant, G. Vora et D. Weeks, «Simulation and the Early Exercise Option Problem,» *Journal of Financial Engineering,* vol. 5, n°13, 1996.

[63] J. Barraquand et D. Martineau, «Numerical Valuation of High Dimensional Multivariate American Securities,» *The Journal of Financial and Quantitative*

Analysis, vol. 30, n°13, pp. 383-405, 1995.

[64] J. Carriere, «Valuation of the early-exercise price for options using simulations and nonparametric regression,» *Insurance: Mathematics and Economics,* vol. 19, n°11, p. 19–30, 1996.

[65] J. Tsitsiklis et B. Van Roy, «Regression Methods for Pricing Complex American-Style Options,» *IEEE Transactions on Neural Networks,* vol. 12, pp. 694-703, 2000.

[66] F. Longstaff et E. Schwartz, «Valuing American options by simulation: A simple least-squares approach,» *Review of Financial Studies,* pp. 113-147, 2001.

[67] A. Ibáñez et F. Zapatero, «Monte Carlo Valuation of American Options through Computation of the Optimal Exercise Frontier,» *Journal of Financial and Quantitative Analysis,* vol. 39, n°12, pp. 253-275, 2004.

[68] M. Broadie et P. Glasserman, «Pricing American-Style Securities Using Simulation,» *Journal of Economic Dynamics and Control,* vol. 21, pp. 1323-1352, 1997.

[69] M. Broadie et P. Glasserman, «A stochastic mesh method for pricing high-dimensional American options,» *Journal of Computational Finance,* vol. 7, pp. 35-72, 2004.

[70] V. Bally et G. Pagès, «A quantization algorithm for solving discrete time multidimensional optimal stopping problems,» *Bernoulli,* vol. 9, n°16, p. 1003–1049, 2003.

[71] A. Gerbessiotis, «Architecture independent parallel binomial tree option price valuations,» *Parallel Computing,* vol. 30, n°12, pp. 301-316, 2004.

[72] V. Podlozhnyuk, «Binomial option pricing model,» 2012. [En ligne]. Available: http://docs.nvidia.com/cuda/samples/4_Finance/binomialOptions/doc/binomialOptions.pdf.

[73] M. Zubair et R. Mukkamala, «High Performance Implementation of Binomial Option Pricing,» *Computational Science and Its Applications,* vol. 5072, pp. 852-866, 2008.

[74] G. Jauvion et T. Nguyen, «Parallelized trinomial option pricing model on GPU with CUDA,» 2008. [En ligne]. Available: http://www.arbitragis-research.com/cuda-in-computational-finance/coxross-gpu.pdf.

[75] Y. Peng, B. Gong, H. Liu et Y. Zhang, «Parallel Computing for Option Pricing Based on the Backward Stochastic Differential Equation,» chez *High Performance Computing and Applications*, Springer, 2010, pp. 325-330.

[76] N. Zhang, A. Roux et T. Zastawniak, «Parallel Binomial American Option Pricing under Proportional Transaction Costs,» *Applied Mathematics*, vol. 3 , n°111A, pp. 1795-1810, 2012.

[77] D. Kaya, «Pricing a Multi-Asset American Option in a Parallel Environment by a Finite Element Method Approach,» 2011.

[78] D. M. Dang, C. C. Christara et K. R. Jackson, «An Efficient Graphics Processing Unit-Based Parallel Algorithm for Pricing Multi-Asset American Options,»

Concurrency and Computation: Practice and Experience, vol. 24, n°18, pp. 849-866, 2012.

[79] L. Khodja, M. Chau, R. Couturier, J. Bahi et P. Spitéri, «Parallel solution of American option derivatives on GPU clusters,» *Computers & Mathematics with Applications*, vol. 65, n°111, pp. 1830-1848, 2013.

[80] I. Toke et J. Girard, «Monte Carlo Valuation of Multidimensional American Options Through Grid Computing,» chez *Lecture notes in computer science*, vol. 3743, Springer-Verlag, 2006, pp. 462-469.

[81] A. Choudhury, A. King, S. Kumar et Y. Sabharwal, «Optimizations in financial engineering: The Least-Squares Monte Carlo method of Longstaff and Schwartz,» chez *IEEE Internatiional Symposium on Parallel and Distributed Pricessing*, 2008.

[82] L. Abbas-Turki et B. Lapeyre, «American Options Pricing on Multi-core Graphic Cards,» chez *Business Intelligence and Financial Engineering, 2009. BIFE '09. International Conference on*, 2009.

[83] L. Abbas-Turki, S. Vialle, B. Lapeyre et P. Mercier, «Pricing derivatives on graphics processing units using Monte Carlo simulation,» *Concurrency and Computation: Practice and Experience*, vol. 26, n°19, p. 1679–1697, 2012.

[84] Y. Peng, H. Liu, S. Yang et B. Gong, «Parallel Algorithm for BSDEs Based High Dimensional American Option Pricing on the GPU,» *Journal of Computational Information Systems*, vol. 10, p. 763–771, 2014.

[85] B. Dai, Y. Peng et B. Gong, «Parallel Option Pricing with BSDE Method on GPU,» chez *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, 2010.

[86] G. Pagès et B. Wilbertz, «GPGPUs in computational finance: massive parallel computing for American style options,» *Concurrency and Computation: Practice and Experience*, vol. 24, n°18, pp. 837-848, 2012.

[87] N. Zhang et K. Man, «Parallel Valuation of the Lower and Upper Bound Prices for Multi-asset Bermudan Options,» chez *Network and Parallel Computing*, Springer Berlin Heidelberg, 2012, pp. 453-462.

[88] J. Wan, K. Lai, A. Kolkiewicz et K. Tan, «A Parallel Quasi-Monte Carlo Approach to Pricing American Options on Multiple Assets,» *International Journal of High Performance Computing and Networking*, vol. 4, pp. 321-330, 2006.

[89] S. Jain et C. Oosterlee, «The stochastic grid bundling method: Efficient pricing of Bermudan options and their Greeks,» 2013. [En ligne].

[90] Á. Leitao Rodríguez et K. Oosterlee, «GPU Acceleration of the Stochastic Grid Bundling Method for Early-Exercise options,» *Submitted to International Journal of Computer Mathematics*, 2015.

[91] Y. Hu, Q. Li, Z. Cao et J. Wang, «Parallel simulation of high-dimensional American option pricing based on CPU versus MIC,» *Concurrency and Computation: Practice and Experience*, vol. 27, n°15, pp. 1110-1121, 2015.

[92] V. Surkov, «Parallel option pricing with Fourier space time-stepping method on graphics processing units,» *Parallel Computing*, vol. 36, n°17, pp. 372-380, 2010.

[93] B. Zhang et C. Oosterlee, «Acceleration of option pricing technique on graphics processing units,» *Concurrency and Computation: Practice and Experience*, vol. 26, n°19, pp. 1626-1639, 2014.

[94] Khronos Group, «OpenCL Best practises guide,» [En ligne]. Available: http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf.

[95] JOCL, [En ligne]. Available: http://www.jocl.org.

[96] Machine Learning Group at University of Waikato, [En ligne]. Available: http://www.cs.waikato.ac.nz.

[97] AMD Corporate, [En ligne]. Available: http://developer.amd.com.

[98] M. Fatica et E. Phillips, «Pricing American options with least squares Monte Carlo on GPUs,» chez *6th Workshop on High Performance Computational Finance*, 2013.

[99] N. Zhang, E. Lim, K. Man et C. Lei, «CPU-GPU Hybrid Parallel Binomial American Option Pricing,» chez *International MultiConference of Engineers and Computer Scientists*, Hong Kong, 2012.

[100] N. Ganesan, R. Chamberlain et J. Buhler, «Acceleration of binomial options pricing via parallelizing along time-axis on a GPU,» chez *Symposium on Application Accelerators in High Performance Computing*, 2009.

[101] S. Frey, G. Reina et T. Ertl, «SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms,» chez *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2012.

[102] T. Han et T. Abdelrahman, «Reducing branch divergence in GPU programs,» chez *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011.

[103] R. de Camargo, «A load distribution algorithm based on profiling for heterogeneous GPU clusters,» chez *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, 2012.

[104] A. Tse, D. Thomas, K. Tsoi et W. Luk, «Dynamic Scheduling Monte-Carlo Framework for Multi-Accelerator Heterogeneous Clusters,» chez *Field-Programmable Technology (FPT), 2010 International Conference on*, 2010.

[105] T. Serban, M. Danelutto et P. Kilpatrick, «Autonomic scheduling of tasks from data parallel patterns to CPU/GPU core mixes,» chez *High Performance Computing & Simulation*, 2013.

[106] Z. Wang, L. Zheng, Q. Chen et M. Guo, «CAP: co-scheduling based on asymptotic profiling in CPU+GPU hybrid systems,» chez *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, 2013.

[107] S. Koichi, S. Hitoshi et M. Satoshi, «Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters,» *International Conference on Cloud Computing*

Technology and Science, vol. 0, pp. 733-740, 2010.

[108] L. Chen, O. Villa, S. Krishnamoorthy et G. Gao, «Dynamic load balancing on single-and multi-GPU systems,» chez *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.

[109] M. Bogdanski, P. Lewis, T. Becker et X. Yao, «Improving Scheduling Techniques in Heterogeneous Systems with Dynamic, On-Line Optimisations,» chez *Complex, Intelligent and Software Intensive Systems (CISIS), 2011 International Conference on*, 2011.

[110] H. Grahn, N. Lavesson, M. Lapajne et D. Slat, «"CudaRF": A CUDA-based Implementation of Random Forests,» chez *9th ACS/IEEE International Conference on Computer Systems and Applications*, 2011.

[111] M. Abualkibash, A. ElSayed et A. 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, n°13, p. 29, 2013.

[112] J. Matienzo et N. Jerger, «Performance Analysis of Broadcasting Algorithms on the Intel Single-Chip Cloud Computer,» chez *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[113] F. Cerou, «Importance splitting for rare event simulation,» 2010. [En ligne]. Available: http://cermics.enpc.fr/~stoltz/Hybrid2010/Presentations/Cerou.pdf.

[114] D. Jacquemart-Tomi, J. Morio et F. Le Gland, «A combined importance splitting and sampling algorithm,» chez *Simulation Conference (WSC), 2013 Winter*, 2013.

[115] M. Dixon, J. Chong et K. Keutzer, «Acceleration of market value-at-risk estimation,» chez *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, 2009.

[116] N. Singla, M. Hall, B. Shands et R. Chamberlain, «Financial monte carlo simulation on architecturally diverse systems,» chez *Workshop on High Performance Computational Finance*, 2008.

[117] A. Kreinin, L. Merkoulovitch, D. Rosen et M. Zerbs, «Measuring Portfolio Risk Using Quasi Monte Carlo Methods,» *Algo Research Quarterly*, vol. 1, n°11, 1998.

[118] «Altera SDK for OpenCL,» [En ligne]. Available: https://www.altera.com.

[119] Swiss Federal Institute of Technology Zurich, «Random Forest,» 2012. [En ligne]. Available: http://stat.ethz.ch/education/semesters/ss2012/ams/slides/v10.2.pdf.

[120] J. Thompson et K. Schlachte, An Introduction to the OpenCL Programming Model, Person Education, 2012.

[121] M. Hasan et F. Boris, «Svm :Machines à vecteurs de support ou séparateurs à vastes marges,» 2006.

[122] The Pennsylvania State University, «Applied Data Mining and Statistical Learning,» 2014. [En ligne]. Available: https://onlinecourses.science.psu.edu/statprogram/stat897d.

[123] L. Abbas-Turki, S. Vialle, B. Lapeyre et P. Mercier, «High dimensional pricing of

exotic European contracts on a GPU Cluster, and comparison to a CPU cluster,» chez *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[124] L. Abbas-Turki, «Parallel Computing for linear, nonlinear and linear inverse problems in finance,» 2012.

[125] G. Boero, J. Smith et K. Wallis, «Sensitivity of the Chi-Squared Goodness-of-Fit Test to the Partitioning of Data,» *Econometric Reviews*, vol. 23, pp. 341-370, 2004.

[126] N. Dickson, K. Karimi et F. Hamze, «Importance of Explicit Vectorization for CPU and GPU Software Performance,» *Journal of Computational Physics*, vol. 230, n°113, pp. 5383-5398, 2011.

[127] Essays, «Comparing Binomial Tree, Monte Carlo Simulation And Finite,» 2013. [En ligne]. Available: http://www.ukessays.com/essays/finance/comparing-binomial-tree-monte-carlo-simulation-and-finite-finance-essay.php.

[128] G. Cocco et A. Cisternino, «Device specialization in heterogeneous multi-GPU environments,» chez *Imperial College Computing Student Workshop*, 2012.

[129] J.-P. Chancelier, B. Lapeyre et J. Lelong, «Using Premia and Nsp for constructing a risk management benchmark for testing parallel architecture,» chez *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[130] M. Fatica et E. Phillips, «Pricing American options with least squares Monte Carlo on GPUs,» chez *Proceedings of the 6th Workshop on High Performance Computational Finance*, 2013.

[131] A. Gaikwad et I. Toke, «GPU based sparse grid technique for solving multidimensional options pricing PDEs,» chez *2nd Workshop on High Performance Computational Finance*, 2009.

[132] R. Genuer, «Forêts aléatoires: aspects théoriques, sélection de variables et applications,» 2010.

[133] P. Glasserman, Monte Carlo Methods in Financial Engineering, Springer, 2004.

[134] P. L'Ecuyer, D. Munger, B. Oreshkin et R. Simard, «Random Numbers for Parallel Computers: Requirements and Methods,» 2015.

[135] D. Lamberton et B. Lapeyre, Introduction Au Calcul Stochastique Appliqué À La Finance (3e édition), ellipses, 2012.

[136] F. Lu, J. Song, F. Yin et X. Zhu, «Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters.,» *Computer Physics Communications*, vol. 183, n°16, pp. 1172-1181, 2012.

[137] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo et J. Lee, «SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters,» chez *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012.

[138] N. Kumar et R. Sengupta, «Value at Risk (VaR): A new technique using Multilevel Monte Carlo Simulation for a derivative portfolio,» chez *3rd Asia-Pacific Business Research Conference*, 2013.

[139] A. Montillo, «Random Forests,» 2009. [En ligne]. Available: http://www.dabi.temple.edu/~hbling/8590.002/Montillo_RandomForests_4-2-2009.pdf.

[140] O. Rosenberg, «Optimizing OpenCL on CPUs,» 2010. [En ligne]. Available: https://www.khronos.org/assets/uploads/developers/library/2010_siggraph_bof_opencl/OpenCL-BOF-Intel-SIGGRAPH-Jul10.pdf.

[141] S. Rul, H. Vandierendonck, J. D'Haene et K. De Bosschere, «An experimental study on performance portability of OpenCL kernels,» chez *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, 2010.

[142] S. Tan, «Towards Efficient Risk Quantification,» 2013.

[143] I. Toke, «Résolution de modèles d'évaluation de produits dérivés financiers sur des architectures de grilles informatiques,» 2006.

[144] L. Van, «Métaheuristiques parallèles sur GPU,» 2011.

[145] G. Virginie, M. Constantinos et V. Stephane, «A Javaspace-based Framework for Efficient Fault-Tolerant Master-Worker Distributed Applications,» chez *19th International Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2011.

[146] L. Wan, K. Li, J. Liu et K. Li, «A Novel CPU-GPU Cooperative Implementation of A Parallel Two-List Algorithm for the Subset-Sum Problem,» chez *Proceedings of Programming Models and Applications on Multicores and Manycores*, 2014.

[147] Z. Yao et J. Owens, «A Quantitative Performance Analysis Model for GPU Architectures,» chez *17th International Symposium on High Performance Computer Architecture*, 2011.

[148] K. Zhang et B. Wu, «Task Scheduling for GPU Heterogeneous Cluster,» chez *IEEE international conference on cluster computing workshops*, 2012.

[149] N. Zhang, E. Lim, K. Man et C. Lei, «CPU-GPU Hybrid Parallel Binomial American Option Pricing,» chez *International MultiConference of Engineers and Computer Scientists*, 2012.