Approche de conception haut-niveau pour l'accélération matérielle de calcul haute performance en finance

Valentin Mena Morales

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UNIVERSITE BRETAGNE LOIRE

THÈSE / IMT Atlantique

sous le sceau de l'Université Bretagne Loire

pour obtenir le grade de

DOCTEUR D'IMT Atlantique

Mention : Sciences et Technologies de l'Information et de la Communication

École Doctorale Sicma

Présentée par

Valentin Mena Morales

Préparée dans le département Electronique

Laboratoire Labsticc

Approche de conception haut-niveau pour l'accélération matérielle de calcul haute performance en finance

Thèse soutenue le 12 juillet 2017

devant le jury composé de :

Frédéric Rousseau
Professeur, Université Grenoble Alpes / président

Olivier Sentieys
Professeur, Enssat - Lannion / rapporteur

Virginie Fresse
Maître de conférences (HDR), Université Jean Monnet / rapporteur

Cédric Cano
Ingénieur, Interface Concept - Quimper / examinateur

Pierre-Henri Horrein
Maître de conférences, IMT Atlantique / examinateur

Amer Baghdadi
Professeur, IMT Atlantique / directeur de thèse

Remerciements

Une thèse occupe de l'espace dans une vie : plus que le résultat d'un travail, elle est la représentation physique des efforts réalisés pour la mener à bien. Elle se construit dans le temps et au travers d'échanges avec les personnes qui l'entourent et ont contribué à son évolution. Je ne peux donc pas présenter ces travaux sans chaleureusement remercier toutes les personnes qui m'ont aidé - et supporté - durant ces quelques années.

Je tiens à remercier toutes les personnes qui ont travaillé à Adacsys à mes côtés : Érik, Linlin, Matthieu, Olivier, Pascal, Sarah, Vincent et Yahia. Cette expérience à vos côtés m'a transformé pour le meilleur et m'a permis de créer des souvenirs doux-amers que je chéris précieusement. De même, l'encadrement de Sandrine Vaton m'a été d'une grande aide lors de cette première étape de ma thèse, notamment pour s'assurer de la justesse des modèles mathématiques employés et m'aider à acquérir les bases de l'évaluation d'options.

Je souhaite également à remercier Bruno Rolland pour la chance qu'il m'a accordée. Sans le financement de ma thèse soutenu par Interface Concept, la liquidation judiciaire d'Adacsys aurait signifié la fin prématurée de cette aventure.

Amer Baghdadi et Pierre-Henri Horrein ont tous deux réalisés des efforts considérables pour me donner les moyens d'achever mes travaux, au delà de ce qui pourrait être attendu d'une équipe d'encadrement. Je ne saurais exprimer à quel point leur soutien m'a été précieux.

Je tiens enfin à remercier l'ensemble du laboratoire pour l'excellente ambiance de travail, notamment les doctorants, stagiaires et post-doc de mon corridor ainsi que Michel Jezequel pour son approche composée et son soutien, et Catherine Blondé pour sa patience envers tous les membres du labo touchés par le syndrome Tournesol (moi le premier).

Enfin, je me dois de mentionner ma famille et mes amis proches, qui m'ont accompagné tout au long de ces travaux.

Résumé

Les applications de calcul haute-performance (HPC) nécessitent des capacités de calcul conséquentes, qui sont généralement atteintes à l'aide de fermes de serveurs au détriment de la consommation énergétique d'une telle solution. L'accélération d'applications sur des plateformes hétérogènes, comme par exemple des FPGA ou des GPU, permet de réduire la consommation énergétique et correspond donc à un compromis architectural plus séduisant. Elle s'accompagne cependant d'un changement de paradigme de programmation et les plateformes hétérogènes sont plus complexes à prendre en main pour des experts logiciels. C'est particulièrement le cas des développeurs de produits financiers en finance quantitative. De plus, les applications financières évoluent continuellement pour s'adapter aux demandes législatives et concurrentielles du domaine, ce qui renforce les contraintes de programmabilité de solutions d'accélérations. Dans ce contexte, l'utilisation de flots haut-niveaux tels que la synthèse haut-niveau (HLS) pour programmer des accélérateurs FPGA n'est pas suffisante. Une approche spécifique au domaine peut fournir une réponse à la demande en performance, sans que la programmabilité d'applications accélérées ne soit compromise.

Nous proposons dans cette thèse une approche de conception haut-niveau reposant sur le standard de programmation hétérogène OpenCL. Cette approche repose notamment sur la nouvelle implémentation d'OpenCL pour FPGA introduite récemment par Altera. Quatre contributions principales sont apportées : (1) une étude initiale d'intégration de cœurs de calculs matériels à une librairie logicielle de calcul financier (QuantLib), (2) une exploration d'architectures et de leur performances respectives, ainsi que la conception d'une architecture dédiée pour l'évaluation d'option américaine et l'évaluation de volatilité implicite à partir d'un flot haut-niveau de conception, (3) la caractérisation détaillée d'une plateforme Altera OpenCL, des opérateurs élémentaires, des surcouches de contrôle et des liens de communication qui la compose, (4) une proposition d'un flot de compilation spécifique au domaine financier, reposant sur cette dernière caractérisation, ainsi que sur une description des applications financières considérées, à savoir l'évaluation d'options.

Mots-clés : Conception haut-niveau, OpenCL, FPGA, GPU, Finance, Accélération matérielle, HPC, HLS, Prototypage

Abstract

The need for resources in High Performance Computing (HPC) is generally met by scaling up server farms, to the detriment of the energy consumption of such a solution. Accelerating HPC application on heterogeneous platforms, such as FPGAs or GPUs, offers a better architectural compromise as they can reduce the energy consumption of a deployed system. Therefore, a change of programming paradigm is needed to support this heterogeneous acceleration, which trickles down to an increased level of programming complexity tackled by software experts. This is most notably the case for developers in quantitative finance. Applications in this field are constantly evolving and increasing in complexity to stay competitive and comply with legislative changes. This puts even more pressure on the programmability of acceleration solutions. In this context, the use of high-level development and design flows, such as High-Level Synthesis (HLS) for programming FPGAs, is not enough. A domain-specific approach can help to reach performance requirements, without impairing the programmability of accelerated applications. We propose in this thesis a high-level design approach that relies on OpenCL, as a heterogeneous programming standard. More precisely, a recent implementation of OpenCL for Altera FPGA is used. In this context, four main contributions are proposed in this thesis : (1) an initial study of the integration of hardware computing cores to a software library for quantitative finance (QuantLib), (2) an exploration of different architectures and their respective performances, as well as the design of a dedicated architecture for the pricing of American options and their implied volatility, based on a high-level design flow, (3) a detailed characterization of an Altera OpenCL platform, from elemental operators, memory accesses, control overlays, and up to the communication links it is made of, (4) a proposed compilation flow that is specific to the quantitative finance domain, and relying on the aforementioned characterization and on the description of the considered financial applications (option pricing).

Keywords : High-level design, OpenCL, FPGA, GPU, quantitative finance, hardware acceleration, HPC, HLS, prototyping

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

Quel que soit le domaine d'application, le calcul haute performance (HPC) comporte une problématique intrinsèque à laquelle il n'est pas possible d'échapper : le besoin croissant en termes de performances s'accompagne d'une augmentation de la complexité des architectures matérielles et logicielles utilisées. Les solutions déployées sont composées de serveurs présentant un grand nombre de cœurs de calcul, ce qui comporte aussi un coût énergétique non négligeable. Ceci ajoute donc une complexité de l'infrastructure physique et logistique, pour répondre aux besoins en place, en approvisionnement énergétique et en dissipation de chaleur.

De plus, les contraintes en termes de compétitivité génèrent une augmentation constante de la charge de calcul. Des solutions hétérogènes existent, basées sur des accélérateurs matériels discrets, pour fournir un rendement en puissance de calcul plus élevé à consommation énergétique identique. Différents accélérateurs sont utilisés en industrie, les plus connus étant les processeurs graphiques (*Graphics Processing Unit*, GPU), les accélérateurs matériels dédiés (*Application Specific Integrated Circuits*, ASIC), ou des accélérateurs matériels reconfigurables (en particulier les *Field Programmable Gates Array*, FPGA). Cette complexité accrue des architectures matérielles de calcul a toutefois un coût en termes de programmabilité ; l'utilisateur doit faire évoluer son paradigme de programmation, que ce soit pour l'utilisation d'algorithmes parallèles ou la gestion explicite de transfert de données entre les différents éléments de l'architecture ciblée. Cette évolution est en fait une accumulation de compétences : les différents modèles de programmation doivent être maintenus, et coexistent entre eux au sein d'une même machine, voire d'une même application.

Dans le domaine financier, ces difficultés sont renforcées par l'évolution constante des algorithmes utilisés. Ceux-ci changent en effet pour s'adapter à l'évolution des réglementations et pour s'accorder à des demandes de sécurité de plus en plus drastiques. A cela s'ajoute une forte compétition, qui demande aux acteurs financiers de réagir rapidement à l'évolution du marché. Ce besoin de réactivité est primordial, mais la complexité en programmation est un point de blocage pour l'adoption de solutions hétérogènes d'accélération.

Problématique

Le domaine se retrouve donc face à un problème complexe : les machines de calcul basées uniquement sur le logiciel deviennent trop complexes et chères à utiliser et à intégrer, mais changer de paradigme de programmation pour pouvoir profiter d'une accélération serait trop coûteux et ne fournirait plus la réactivité requise dans le domaine. Une des solutions à ce problème serait donc de réduire la complexité de la programmation des architectures hétérogènes. Dans ce contexte, l'objectif de la thèse est d'explorer et de proposer de nouvelles approches donnant un accès en programmabilité à des architectures hétérogènes performantes pour le domaine de la finance quantitative.

Le choix d'un ou de plusieurs types d'accélérateur est complexe. Le GPU est l'un des plus populaires : sa programmation s'apparente à une programmation logicielle, et il offre une puissance de calcul théorique impressionnante, à plusieurs TFlop/s [NVi13]. Cependant, cette puissance théorique est difficile à atteindre en pratique. Il impose également des contraintes fortes sur le type d'algorithme implémentable. En particulier, le parallélisme de données doit être massif pour espérer atteindre une performance acceptable. La consommation énergétique d'un GPU est également comparable à celle d'un CPU, ce qui maintient voire aggrave les contraintes posées sur la complexité de l'infrastructure des centres de calcul.

L'utilisation de FPGA correspond à une approche différente, plus bas niveau, qui requiert des développements plus spécifiques. Cependant, elle est plus configurable et moins limitée au niveau des algorithmes implémentables. Ce type d'accélérateur est également peu gourmand en énergie, comparativement aux GPU et GPP. Néanmoins, avec ce type d'accélérateur se pose le problème de l'accès en programmabilité.

Une approche générique pour améliorer la programmabilité consiste en une traduction automatique d'un programme écrit pour un modèle de programmation dans un autre programme. Le développeur utilise un modèle connu (dans notre cas, un logiciel standard), qui va ensuite être traité par un outil, afin d'être modifié et adapté pour un autre modèle. Ce type d'approche existe en particulier pour les accélérateurs matériels : la Synthèse de Haut Niveau (*High-Level Synthesis*, HLS) en est un bon exemple. Cependant, même si les développements récents montrent une volonté d'améliorer ces outils, la HLS ne permet pas encore de s'abstraire complètement de l'architecture. Elle impose également l'utilisation d'outils complexes, et souvent une réécriture du code pour optimiser la traduction. De plus, l'intégration au sein d'une application complète est souvent difficilement gérée. C'est donc un bon outil pour la construction d'un environnement, mais qui n'offre pas toute l'abstraction souhaitable pour un développeur logiciel.

D'autres approches visent à unifier au sein d'un même environnement de développement différentes cibles de calcul. L'approche la plus simple est probablement l'approche par librairie. Une librairie présente une interface unique, mais qui peut être déployée sur différents processeurs ou accélérateurs. Cette approche a un intérêt certain du point de vue de l'utilisabilité, mais nécessite une conception qui peut être complexe, en particulier au niveau de la définition de l'interface : une architecture optimale pour une cible logicielle n'est pas nécessairement optimale pour une autre

cible. Elle nécessite également d'implémenter la librairie pour chaque cible.

D'autres approches proposent un mélange des deux solutions : un environnement de programmation unique, qui va ensuite être compilé sur la cible requise. L'exemple le plus parlant est le cas de l'Open Computing Language (OpenCL), qui propose une séparation explicite en contrôle et en noyaux de calcul. Ces derniers sont ensuite compilés pour la cible (GPU, FPGA ou CPU), en utilisant si besoin une approche de traduction du modèle. Cette traduction est facilitée par l'utilisation d'un paradigme particulier, et par une extension du langage logiciel qui permet de guider la traduction en diminuant les degrés de liberté. On s'éloigne cependant du modèle logiciel standard, même si un même code peut être exécuté sur différentes plateformes.

Finalement, la solution optimale consisterait en une écriture d'un programme logiciel simple, qui serait ensuite analysé et transformé pour l'architecture souhaitée. Cette solution est utopique : l'espace de recherche pour un tel outil est trop vaste, en particulier à cause du grand nombre de degrés de liberté. Cependant, une approche alternative pourrait être appliquée en connaissant le domaine d'application. Cette connaissance du domaine permet d'envisager de diminuer les degrés de liberté, et donc de diminuer la taille de l'espace de recherche d'une solution.

Dans le cadre de cette étude, le domaine général considéré est la finance quantitative. Cela reste un domaine très vaste et nous nous sommes donc concentrés sur un sous-domaine de travail approchable, à savoir l'évaluation d'options financières. Les options sont des instruments financiers reposant sur une modélisation de produits sous-jacents côtés en bourse. Pour pouvoir estimer avec fiabilité leur valeur, de multiples modèles mathématiques existent, ainsi que plusieurs méthodes d'implémentations. Dans un cas d'utilisation usuel, de nombreuses options différentes peuvent être évaluées dans un laps de temps court, que ce soit pour évaluer la valeur de portefeuilles d'options ou pour évaluer le risque pris par un acteur financier sur une position particulière. Les développeurs qui portent ses produits financiers et les algorithmes associés sur des fermes de serveurs HPC utilisent majoritairement des flots logiciels haut-niveau et n'ont pas les compétences nécessaires pour concevoir - voire même utiliser - des architectures hétérogènes.

Contributions

Différentes approches sont donc possibles pour atteindre l'objectif d'accessibilité des architectures hétérogènes. Dans ces travaux, plusieurs contributions sont proposées qui permettent de tendre vers cet objectif final. L'accélérateur principal étudié dans ce cadre est le FPGA, qui représente selon nous le meilleur compromis potentiel entre performance, flexibilité et consommation énergétique dans le cadre du calcul financier. Le développement d'outils de programmation de haut-niveau permet également de diminuer sa complexité de programmation. Cependant, il n'est pas souhaitable d'imposer le choix d'un accélérateur spécifique. L'une des contraintes que nous nous sommes imposées dans ce travail est donc de toujours garder la possibilité de porter le travail sur d'autres cibles.

Cela passe principalement par l'utilisation d'OpenCL comme environnement de développement.

Les contributions de cette thèse reposent sur ce flot de programmation, et visent à rendre ce flot plus accessible, le travail de génération pour la cible étant ensuite laissé à OpenCL. Cette approche a plusieurs avantages. Elle laisse la charge du support d'un accélérateur au concepteur de l'accélérateur, qui est le plus à même de faire ce travail efficacement. Elle permet de travailler sur un niveau d'abstraction intermédiaire, ce qui permet de réduire encore plus l'espace de recherche des solutions. Elle permet également de générer un code en logiciel (malgré ses spécificités), qui reste donc "lisible" pour un développeur, ce qui facilite la compréhension et la validation des modifications réalisées.

OpenCL est actuellement supporté par Intel PSG (ex-Altera, les deux noms sont utilisés indifféremment dans ce document), ce qui permet l'utilisation de FPGA au travers du flot. La plupart des constructeurs de GPU et GPP supportent également OpenCL, à différents niveaux. Finalement, Xilinx, l'un des deux fabricants majeurs (avec Intel PSG) de puces FPGA, travaille également sur ce support. Le choix d'OpenCL permet donc d'envisager une portée large pour ce travail.

Les principales contributions apportées dans ce travail de thèse sont les suivantes :

— Une étude d'intégration de cœurs de calculs matériels au sein d'une librairie logicielle de calcul financier. Cette contribution vise à étudier l'intérêt d'une abstraction basée sur une librairie, en s'appuyant sur une librairie connue (QuantLib), et en cherchant à accélérer des portions critiques du code.
— Une exploration d'architectures et de leur performances respectives, ainsi que la conception d'une architecture dédiée pour l'évaluation d'option américaine et l'évaluation de volatilité implicite à partir d'un flot haut-niveau de conception. À travers cette contribution, l'acceptabilité et les performances d'OpenCL pour un développeur logiciel sont évaluées.
— La caractérisation détaillée d'une plateforme Altera OpenCL, des opérateurs élémentaires, des surcouches de contrôle et des liens de communication qui la compose. L'implémentation d'OpenCL par Altera est prometteuse, mais souffre d'un problème récurrent avec les outils de traduction : il est difficile d'évaluer comment une directive ou une opération écrite dans un langage de haut niveau est traduite sur la cible finale. Cette contribution permet de mieux comprendre le lien entre le code et la plateforme, et d'évaluer la performance théorique d'une plateforme OpenCL d'Altera.
— Une proposition de flot de compilation spécifique au domaine financier, reposant sur un *back-end* de synthèse haut-niveau et adaptée à la plateforme cible caractérisée précédemment. Cette proposition est appuyée par une implémentation de certains éléments, qui permettent de valider l'intérêt de l'approche. Elle s'appuie sur les spécificités du domaine et d'OpenCL, étudiées dans les contributions précédentes, pour proposer un flot réalisable, et tendre vers une abstraction quasi-complète de la cible lors de l'écriture du code.

Organisation du manuscrit

Dans le chapitre 1, nous introduisons les problématiques rencontrées lors de l'accélération de calcul dans le domaine financier. Nous présentons le sous-ensemble du domaine financier considéré, en se concentrant sur les modèles mathématiques employés, les plateformes de calcul utilisables et les méthodes d'implémentation de ces modèles sur matériel.

Dans le chapitre 2, nous présentons une première approche pour rendre accessible des plateformes hétérogènes, en partant du domaine d'application. Nous décrivons une couche logicielle permettant l'intégration de noyaux de calcul accélérés à une librairie de calcul financier *Open Source*, QuantLib. L'approche a pour objectif de conserver les abstractions spécifiques au domaine et déjà présentes dans QuantLib et d'intégrer à celle-ci une accélération matérielle, de manière transparente pour l'utilisateur.

Dans le chapitre 3, nous nous concentrons sur une approche spécialisée d'implémentation d'un modèle financier basée sur un flot HLS. Nous réalisons une étude architecturale reposant sur le flot de programmation OpenCL et étudions les performances obtenues sur plusieurs accélérateurs, en termes de vitesse de calcul et de consommation énergétique. Nous étendons cette application au calcul de volatilité implicite, dont la complexité supérieure demande une gestion fine de la partie contrôle de l'architecture.

Enfin, nous caractérisons une plateforme matérielle spécifique dans le chapitre 4, ce qui fournit une base sur laquelle nous proposons un flot de programmation liant le domaine applicatif au flot de compilation, synthèse et exécution OpenCL d'Altera. Nous définissons un état de l'art sur les langages de programmation spécifiques à un domaine (DSL : *Domain Specific Language*) et nous appliquons les conclusions de l'état de l'art pour déterminer comment appliquer les spécificités du calcul d'options financières pour leur implémentation sur plateforme matérielle. Nous décrivons finalement en détail l'architecture d'un outil de compilation traduisant un programme financier écrit dans un langage de haut-niveau (C) en une représentation intermédiaire qui est utilisée en entrée du flot de compilation d'Altera pour OpenCL.

Chapitre 1

Applications en finance et plateformes de calcul

DANS ce chapitre, nous introduisons les problématiques rencontrées lors de l'accélération de calcul dans le domaine financier. Nous commençons par présenter le domaine de la finance quantitative, réduit au calcul d'options par souci de clarté et pour se restreindre à un espace de problème approchable. Nous définissons ce qu'est une option financière et décrivons différentes modélisations mathématiques d'un tel instrument financier. Nous définissons ensuite les plateformes de calculs employées pour supporter le besoin en puissance de calcul du domaine et nous décrivons leurs caractéristiques ainsi que leurs limites d'utilisation respectives. Ces plateformes matérielles sont complexes et nécessitent des suites logicielles particulières pour pouvoir les exploiter ; nous décrivons donc OpenCL, un des standards de programmation qui répond à cette contrainte. Nous définissons ensuite les différentes méthodes d'implémentation qui permettent de porter les modèles mathématiques d'instruments financiers sur ces plateformes et concluons sur leurs avantages respectifs. Une fois ces points traités, nous montrons enfin en quoi l'accélération de calcul sur plateforme matérielle hétérogène répond à un besoin en accélération au sein de la finance quantitative.

1.1 Calcul d'options

1.1.1 Options financières

Une option est un produit dérivé financier qui fournit à son possesseur le droit d'acheter (*call*) ou de vendre (*put*) un produit à un certain moment, pour un prix fixé d'exercice (*strike*, noté K) à l'avance. Cette option a une durée de vie, qui peut être de l'ordre du mois ou de l'année. On parle de date d'expiration ou de maturité pour décrire la fin de validité de cette option. Si on prend l'exemple d'un trader qui possède une option de type *call* sur une action Google (GOOG). L'évolution de l'action sur la durée de vie de l'option (avril à septembre 2016) est représentée sur la figure 1.1.

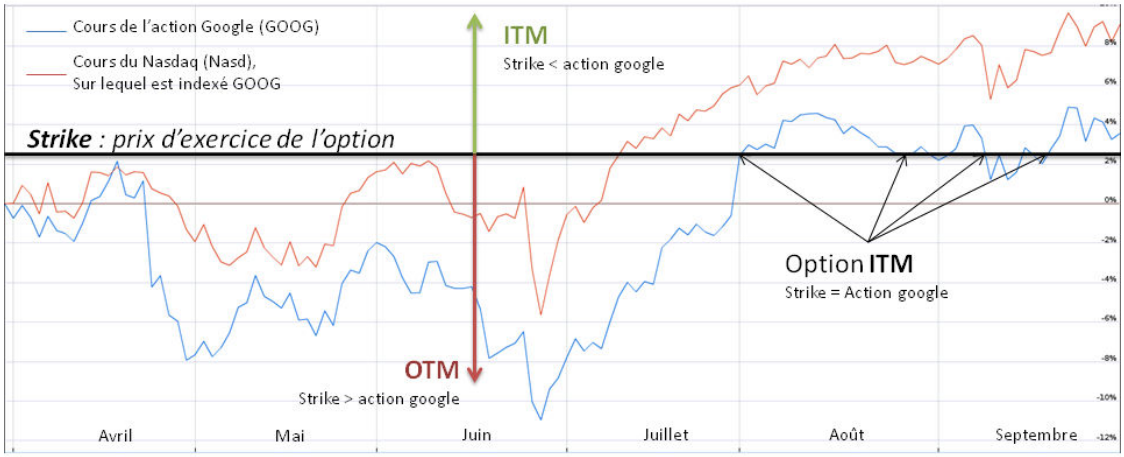

FIGURE 1.1 – Représentation d'une option *call* sur une action Google d'avril à septembre 2016

Cette option lui donne donc le droit d'acheter une action Google pendant la durée de vie de l'option, à un prix égal à la valeur du *strike*. Si l'action monte, le trader a tout intérêt à exercer l'option pour acheter l'action à un prix inférieur au prix du marché. Il aura gagné sur cette opération l'écart entre le *strike* et la valeur en bourse de l'action à la date d'exercice. Au contraire, si l'action diminue, le trader n'a pas intérêt à exercer cette option, le *strike* étant plus élevé que le prix réel de l'action. Cette option lui permet donc de se prémunir contre une augmentation trop importante du prix des actions Google. Le raisonnement inverse s'applique dans le cas d'une option *put*.

Une option a plusieurs caractéristiques, décrites sur la figure 1.1. On dit d'une option qu'elle est *At The Money* (ATM) lorsque la valeur du *strike* est égale à la valeur de l'action dite sous-jacente, à un instant donné. En suivant le même raisonnement, on utilise le terme *In The Money* (ITM) dans le cas où l'option est profitable. Ceci est représenté en vert sur la figure. De même, une option est *Out of The Money* (OTM) dans le cas où l'option n'a pas de valeur (en rouge sur la figure).

Différents types d'options sont utilisés sur les marchés financiers en fonction des besoins des différents acteurs. Ces types diffèrent sur plusieurs degrés de liberté, comme par exemple la date d'exercice de l'option ou la manière de déclencher cet exercice. Les types d'option considérés dans

cette thèse sont les suivants :

- Option européenne : L'option européenne ne peut être exercée qu'à sa date d'expiration, notée T. Si on note $S(T)$ le prix du sous-jacent à expiration, la valeur à échéance d'une option européenne (*call*) est $P(T) = max(S(T) - K, 0)$. Dans la pratique, les options européennes sont utilisées pour calibrer des modèles d'évaluation d'options plus complexes.
- Option américaine : L'option américaine peut être exercée à tout instant entre sa date d'achat et sa date d'expiration. Sa valeur à échéance suit la même formule qu'une option européenne, à l'exception de sa date d'exercice. Les options européennes et américaines sont souvent appelées "options vanilles", du fait de leur simplicité. Par opposition, les autres options sont appelées exotiques.
- Option asiatique : La valeur d'une option asiatique est dite *path-dependent* : sa valeur repose sur la moyenne du prix du sous-jacent jusqu'à la date d'expiration de l'option (aussi définie comme date de maturité de l'option). En conservant les mêmes notations, on obtient la valeur à expiration $P(T) = \max(moy(S(T)) - K, 0)$ pour un call.
- Option à barrière : Il existe deux types d'options à barrière. Une option *knock-in* ne peut être exercée que si la valeur du sous-jacent a atteint un certain seuil avant maturité. Inversement, une option *knock-out* perd toute valeur si l'option franchit une certaine barrière. Ces options permettent de contrôler les gains et pertes potentiels.

Il est également possible de combiner ces types d'options. Cependant, la multiplication d'objets complexes dans les portefeuilles les rend difficiles à modéliser et peu prédictibles. Les crises financières récentes (crise des *subprimes* de 2007 [Hul14]) ont montré les risques associés à cette complexité. Les portefeuilles tendent donc à se simplifier, même si certains d'entre eux, plus anciens, comportent toujours ce type d'options dites exotiques.

1.1.2 Modèles d'évaluation

Afin d'estimer la valeur d'une option, il est nécessaire de modéliser l'évolution du prix du produit sous-jacent au cours de sa vie. Il existe un grand nombre de modèles différents et utilisés sur le marché. De plus, ces modèles sont constamment raffinés pour être au plus proche de la réalité des marchés. Les modèles parmi les plus usités sont décrits dans cette section. Les hypothèses de validité de chacun des modèles ne seront pas explicitées en détail (se référer à [Hul14], [LL96] et à [Mit09] pour plus de précisions).

Chacun des modèles présentés ci-après peut être étendu pour calculer les dérivées partielles du prix de l'option en fonction des différents paramètres du modèle. Ces dérivées partielles sont nommées Grecques et sont essentiellement utilisées dans la couverture de risques.

1.1.2.1 Modèle de Black et Scholes

Le modèle de Black et Scholes [BS73] est considéré comme le point de départ de l'ingénierie financière telle qu'on la connait aujourd'hui. Il est toujours utilisé et est à l'origine de la plupart

des modèles existants.

Dans ce modèle, on considère que l'actif sous-jacent suit une équation différentielle stochastique. Cette équation accepte les paramètres suivants :

— μ, l'espérance de la rentabilité du sous-jacent. Ce paramètre représente le niveau de risque du sous-jacent (plus il est élevé, plus le risque est grand).

— σ, la volatilité du cours du sous-jacent. Ce paramètre représente l'instabilité de son prix (plus il est élevé et plus les fluctuations sont fortes).

— W(t) un mouvement brownien gaussien.

L'équation différentielle du modèle est la suivante [Hul14] :

$$dS(t) = S(t)(\mu dt + \sigma dW(t)) \tag{1.1}$$

Elle a pour solution (application de la formule d'Itô) :

$$S(t) = S(0)e^{(\mu t - \frac{\sigma^2}{2}t + \sigma W(t))}$$

A partir de cette équation et des hypothèses du modèle, il est possible d'obtenir une formule analytique pour évaluer précisément le prix d'une option européenne. Pour les autres types d'options, on ne peut pas trouver de formule analytique et il faut donc employer des méthodes numériques pour les évaluer (voir section 1.3).

Dans ce modèle, la volatilité est supposée constante. Il existe deux manières d'estimer cette volatilité : à partir de données historiques (valeurs du cours observées dans le passé), ou à partir d'une méthode implicite. Cette dernière méthode repose sur la croissance stricte du prix de l'option en fonction de la volatilité, ce qui permet d'associer à un prix une unique volatilité (tout autre paramètre étant fixé). On peut donc inverser le modèle pour trouver la volatilité correspondant à un prix donné, avec des jeux d'options cotées en bourse comme référence. Cette opération est toutefois complexe à calculer (charge de calcul importante) et l'implémentation reste problématique.

L'implémentation de ce modèle sur accélérateur a fait l'objet de nombreux articles. Les travaux de [JLT11a] comparent notamment différentes implémentations de Black et Scholes sur FPGA. L'intérêt du modèle de Black et Scholes est sa simplicité et sa capacité à calculer des volatilités implicites, malgré les différences observées entre les prix effectifs du marché et les prix d'options calculés avec ce modèle.

1.1.2.2 Modèles à volatilité stochastique

Le modèle de Black et Scholes est un modèle très utilisé, mais ses hypothèses de départ sont fortes et ne correspondent pas aux valeurs observées sur les marchés. Notamment, l'hypothèse de constance de la volatilité n'est pas observée en pratique : elle a tendance à évoluer en fonction du prix du sous-jacent et du type d'option évaluée. Le tracé de la volatilité implicite en fonction du prix du sous-jacent est appelé "*smile* de volatilité", représenté sur la figure 1.2. Ce *smile* représente

la corrélation entre la valeur de volatilité implicite et le prix d'exercice *strike* de l'option. L'existence de ce *smile* provient des limitations des hypothèses de travail sur lesquelles reposent les modèles utilisés. La forme de la courbe peut être interprétée par la fébrilité des marchés lorsque l'écart entre la valeur de l'option et le *strike* est élevé. Cette courbe s'est d'ailleurs creusée pour les options sur action après le krach d'octobre 1987 [Hul14]. Elle était quasiment plate auparavant, mais ce krach a érodé la crédibilité des modèles d'options, ce qui a rendu les acteurs financiers plus prudents et plus réticents à s'engager sur des risques importants.

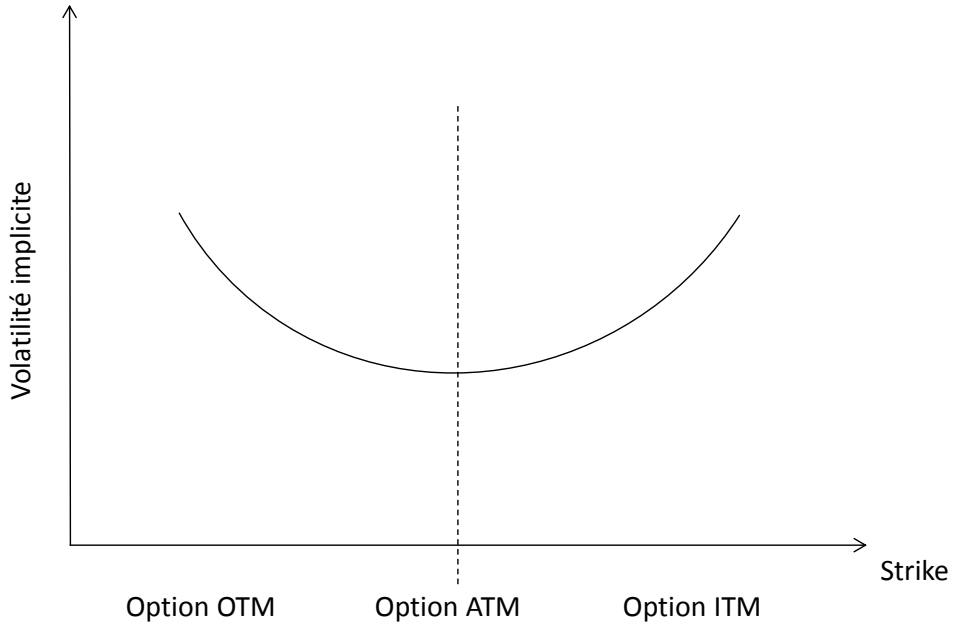

FIGURE 1.2 – Exemple de *smile* de volatilité - option call

Pour tenir compte de l'imprécision sur la volatilité, il est possible d'utiliser un modèle simulant l'évolution de la volatilité au cours du temps à l'aide d'un processus stochastique. Deux modèles principaux reposent sur ce procédé : le modèle Stochastic Alpha Beta Rho (SABR) [HKLW02] et le modèle d'Heston [Hes93]. SABR est une extension stochastique du modèle Constant Elasticity of Variance (CEV) [JT]. Il est défini par les équations suivantes [HKLW02] :

$$\begin{aligned} dF(t) &= \alpha(t)F^{\beta}(t)dW_1(t) \\ d\sigma^2(t) &= \xi\sigma^2(t)dW_2(t) \\ dW_1(t)dW_2(t) &= \rho dt \end{aligned} \tag{1.2}$$

Le nom des différents paramètres diffère ici des autres modèles pour respecter les conventions habituelles de SABR. F représente le sous-jacent, α la volatilité (habituellement σ) et ξ la volatilité de la volatilité α. Le modèle SABR est traditionnellement utilisé sur le marché *London InterBank*

Offered Rate (LIBOR) pour des contrats *forward*, équivalant à des options avec obligation d'exercice.

Un autre modèle basé sur une volatilité stochastique est le modèle d'Heston. Ce modèle est défini à partir du jeu d'équations suivant :

$$\begin{aligned} dS(t) &= rS(t)dt + \sqrt{v_\sigma(t)}S(t)dW_1(t) \\ dv_\sigma(t) &= \kappa(\theta - v_\sigma(t))dt + \xi\sqrt{v_\sigma(t)}dW_2(t) \\ dW_1(t)dW_2(t) &= \rho dt \end{aligned} \tag{1.3}$$

La variance v_σ ($v_\sigma = \sigma^2$) est une grandeur positive. Elle est de plus strictement positive lorsque la condition de Feller est remplie : $2\kappa\theta > v_\sigma^2$. Le modèle d'Heston est notamment connu pour fournir une solution analytique pour les options européennes sous certaines hypothèses. Cette solution analytique est plus réaliste que celle fournie par le modèle de Black et Scholes.

Pour que ces modèles approchent au mieux le *smile* de volatilité, une première étape de calibration est nécessaire. Cette étape est plus complexe que pour le modèle de Black et Scholes où un seul paramètre (la volatilité constante) doit être calibré. Plusieurs méthodes peuvent être utilisées pour obtenir des valeurs de paramètres optimales en fonction des valeurs du marché [GS10] [GR09].

1.1.2.3 Modèles d'actifs avec sauts

Les modèles décrits précédemment reposent sur des distributions log-normales (du type e^{dW} avec W suivant une loi gaussienne) et ne peuvent pas simuler les sauts des prix des sous-jacents observés parfois sur le marché. Merton [Mer76] a proposé une extension du modèle de Black et Scholes simulant des sauts. Ce modèle, appelé *jump-diffusion model* est défini de la manière suivante :

$$dS(t)/S(t) = (\mu - \theta\kappa)dt + \sigma dW(t) + dN_\lambda(t) \tag{1.4}$$

où

- θ est le nombre de sauts par an
- κ est la valeur moyenne d'un saut, mesurée en pourcentage du prix du sous-jacent $S(t)$
- N_λ est un processus de Poisson indépendant du mouvement Brownien

Un processus de Poisson est un processus stochastique qui permet de compter le nombre d'événements ayant eu lieu durant une certaine période. Il possède les propriétés suivantes :

$$\begin{aligned} &\mathbb{P}[N_\lambda(t) = n] = e^{-\lambda t}\frac{(\lambda t)^n}{n!}, \forall n \in \mathbb{N} \\ &\mathbb{E}[N_\lambda(t)] = \lambda t \\ &Var[N_\lambda(t)] = \lambda t \end{aligned}$$

Le modèle d'actifs avec sauts mène lui aussi à une solution analytique possible pour les options européennes.

De même, il existe une extension du modèle d'Heston avec sauts [Bat96] :

$$\begin{aligned} dS(t) &= rS(t)dt + \sqrt{\sigma(t)}S(t)dW_1(t) + dZ(t) \\ dv_\sigma(t) &= \kappa(\theta - v_\sigma(t))dt + \xi\sqrt{v_\sigma(t)}dW_2(t) \\ dW_1(t)dW_2(t) &= \rho dt \end{aligned} \tag{1.5}$$

$Z(t)$ est ici un processus de Poisson composé, indépendant des deux mouvements Brownien, d'intensité λ et de sauts indépendants J suivant la loi :

$$\log(1+J) \sim N(\log(1+\chi) - \frac{1}{2}\alpha^2, \alpha^2)$$

1.2 Plateformes de calcul

Dans la section précédente, nous avons présenté plusieurs grands groupes de modèles utilisés pour le calcul d'options financières. Ces modèles sont implémentés sur des plateformes de calcul spécifiques qui permettent aux acteurs financiers d'obtenir un niveau de performance (temps de calcul, précision des résultats) suffisant pour rester compétitif. Nous présentons ici les composants les plus utilisés dans ces plateformes, ainsi que les limitations et les problèmes liés à leur intégration et utilisation. Ces composants ont chacun leurs caractéristiques qui ont un impact sur la manière d'implémenter les modèles d'évaluation d'option. Dans cette section, les exemples et chiffres cités correspondent à des composants prévus pour le HPC. D'autres composants existent avec des caractéristiques qui peuvent différer, par exemple dans les calculs embarqués.

1.2.1 GPP

Les processeurs à usage générique (*General Purpose Processor*, GPP) sont des processeurs qui privilégient la généricité et la flexibilité par rapport à leur performance énergétique et leur débit de calcul. Ils sont omniprésents de par leur versatilité et leur facilité de programmation. Ce sont les processeurs que l'on retrouve usuellement sur les machines personnelles et les serveurs de calcul, même si ces derniers ont beaucoup évolué. Ils sont la cible privilégiée des programmes logiciels développés dans tout secteur.

Ces GPP sont des composants principalement séquentiels. Ils fonctionnent sur la base d'un jeu d'instructions. Ces instructions définissent les opérations réalisées par le processeur et peuvent être plus ou moins complexes en fonction du processeur utilisé. Les GPP ne peuvent pas fonctionner sans mémoire. Cette mémoire contient le programme (suite d'instructions), ainsi que les données consommées et produites par le processeur en suivant l'exécution du programme. Depuis plusieurs années, les performances requises pour suivre la loi de Moore impliqueraient une fréquence d'horloge et une densité de transistors telles que les puces ne pourraient pas dissiper la chaleur produite. Les processeurs reposent donc principalement sur une architecture multi-cœur (entre 4 et 8 cœurs), qui fonctionnent à des fréquences maîtrisées. Ce passage à une architecture parallèle implique un

changement dans le paradigme de programmation et demande au développeur logiciel de concevoir des programmes multithreadés pour utiliser au mieux cette architecture.

Les GPP fonctionnent généralement à des fréquences hautes (de l'ordre du GHz), pour une puissance consommée de plusieurs dizaines de W (jusqu'à plus de 100 W). Cette fréquence ne correspond pas au débit d'instructions, la plupart des instructions requérant plusieurs cycles. En termes de performances, les processeurs haut-de-gamme actuels atteignent des valeurs de 100 GFLOPS (nombre d'opérations flottantes par seconde).

Une des plus fortes limites de performance des processeurs modernes réside dans les communications avec les autres composants. Selon les méthodes d'interconnexion, les débits atteignables sont de l'ordre du kbps et vont jusqu'à la centaine de Gbps. Ceci est tout particulièrement contraignant pour les accès à la mémoire, qui peuvent notamment avoir une forte latence (quelques dizaines de cycles d'horloge du GPP). Pour compenser, les GPP sont généralement basés sur une architecture hiérarchique de la mémoire. Une petite mémoire rapide à faible latence (le cache) est intégrée au plus près du processeur et contient des copies des instructions et données actuellement utilisées par le processeur. La taille du cache est donc primordiale pour la performance de ces composants.

L'utilisation de ces GPP est maîtrisée et répandue. Les programmes sont écrits dans des langages de haut-niveau, adaptés à l'utilisation souhaitée. Ces programmes sont ensuite compilés en une suite d'instructions compréhensibles par le GPP. Cette étape de compilation intègre des optimisations qui sont le fruit de plusieurs décennies de travaux et permet donc de générer un code bien adapté à l'architecture des GPP. L'exécution s'effectue dans un environnement contrôlé et protégé (le système d'exploitation). Cette surcouche logicielle isole les différents programmes et facilite la gestion du matériel. Elle s'accompagne toutefois d'un coût en performance non négligeable.

Le coût des GPP est aussi financier. Pour le HPC, les architectures déployées sont nécessairement répliquées un grand nombre de fois pour fournir une puissance de calcul suffisante. Ceci se traduit par un coût important et croissant en alimentation électrique, en refroidissement et en maintenance. L'exemple des serveurs de Google récemment installés en Finlande dans une ancienne papeterie est représentatif de l'ingéniosité déployée pour pallier ces difficultés [Bus].

1.2.2 GPU

Les processeurs graphiques (*Graphics Processing Unit*, GPU) ont dépassé leur utilisation initiale (rendu graphique pour leur affichage) et sont désormais fréquemment déployés en tant qu'accélérateurs de calculs. Cette utilisation, appelée *General Purpose Graphic Processing Unit* (GPGPU), a été rendue possible suite à l'augmentation de la complexité des calculs réalisés pour le rendu graphique. Pour répondre aux demandes de l'industrie vidéo, les fabricants ont rendu leurs architectures de plus en plus flexibles. Cette flexibilité a ensuite été détournée pour réaliser des calculs n'ayant plus aucun rapport avec le traitement graphique.

Pour comprendre le fonctionnement d'un GPU, prenons l'exemple d'une puce relativement récente du constructeur NVidia (architecture Kepler).

Cette puce contient plusieurs clusters, eux-mêmes subdivisés en *Streaming Multiprocessors*

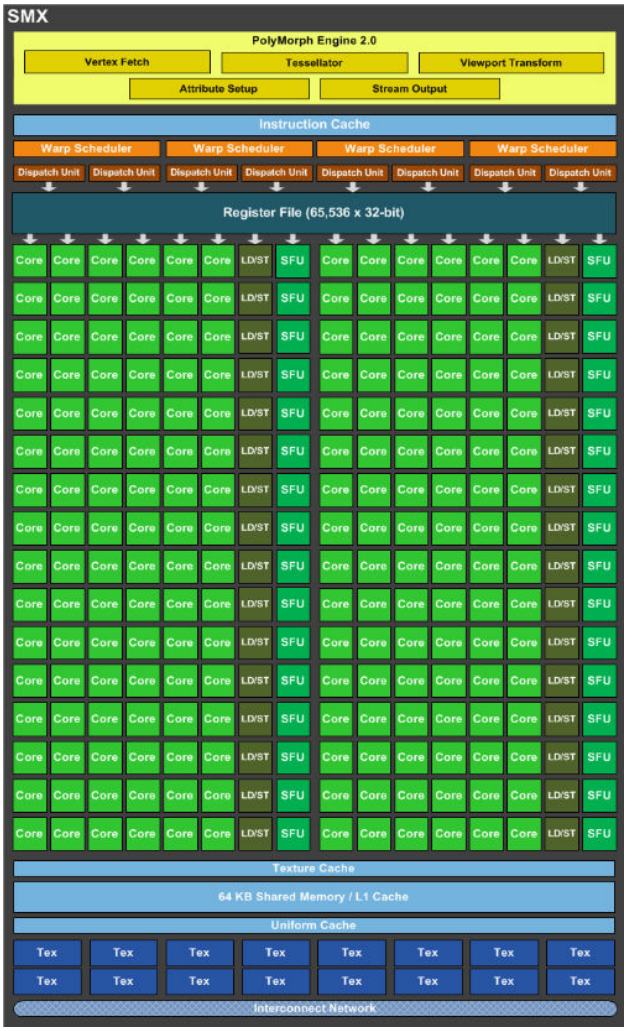

FIGURE 1.3 – Exemple d'architecture d'un SMX - GeForce GTX 680 [NVI12]

(SMX). Un SMX (figure 1.3 [NVI12]) est composé de multiple processeurs travaillant simultanément sur des données différentes, avec des ressources partagées, comme notamment une ALU (*Arithmetic Logic Unit*) par tranche de 8 *stream processors*.

Une architecture classique de GPU présente des *clusters* fonctionnant en *Multiple Instruction, Multiple Data* (MIMD), avec chacun un comportement *Single Instruction, Multiple Data* (SIMD) à plus fine échelle. Un fonctionnement MIMD représente un parallélisme complet, avec différentes instructions exécutées sur différents jeux de données en parallèle. Un fonctionnement SIMD est légèrement dégradé par rapport au MIMD. Il ne peut exécuter qu'une seule instruction, mais sur plusieurs jeux de données en même temps.

Une carte GPU présente typiquement plusieurs régions mémoires, notamment :

— Plusieurs gigaoctets de mémoire dite globale, accessible par tous les éléments du GPU. Elle sert aussi pour le transfert de données avec l'hôte, via PCIe (standard de bus local). Les accès à cette mémoire doivent être groupés par front de données traitées (dans un *cluster*)

et présentent une latence forte (ordre de grandeur typique : ~100 - ~1000 cycles d'horloges).
— Des zones mémoires dites partagées, qui sont accessibles pour tout processeur au sein d'un *cluster*. La mémoire partagée présente une latence plus faible (~10 à ~100 cycles d'horloge), pour une taille mémoire de quelques koctets.
— De bancs de registres, accessibles au sein d'un *cluster* en un cycle d'horloge (s'il n'y a pas de conflit d'accès). Ces registres sont toutefois limités en taille (64Kx32-octets par exemple pour une architecture Kepler).

La puissance requise pour le fonctionnement d'une carte graphique est généralement élevée (de l'ordre de ~100 W), du fait de leur haute fréquence d'horloge (de l'ordre du GHz). Ceci permet d'atteindre un débit de calcul très élevé, de l'ordre de plusieurs TFLOPS. Cependant, ces composants, bien qu'offrant des performances théoriques alléchantes, sont en pratique bien moins performants qu'annoncés. En effet, les applications courantes ne sont généralement pas complètement adaptées au modèle SIMD. Cet écart, conjugué à la complexité du modèle de la mémoire des GPU et aux faibles performances des liens de communication entre les GPU et leur hôte ont un impact négatif sur les performances.

Les industriels supportent l'essor du GPGPU via l'ouverture d'API pour leur programmation (CUDA pour NVIDIA, Stream pour AMD, OpenCL de manière plus générale, tel que décrit en 1.2.4.3), ainsi que des librairies de calcul scientifiques performantes, spécifiquement développées pour leur architecture (BLAS, FFT, etc.). L'écriture d'un programme GPGPU est complexe. Elle passe par un partitionnement explicite du code entre l'hôte (pour le contrôle et l'ordonnancement) et le GPU (pour les calculs proprement dits). De plus, la gestion de la répartition des données en mémoire est laissée au développeur (contrairement à un programme sur GPP). La recherche des paramètres de parallélisation (déroulement de boucle, nombre d'instances, nombre de cœurs impliqués, ...) est également explicite et nécessite une phase de calibration qui peut être longue, voire même ne pas aboutir. Des efforts sont réalisés pour simplifier le développement sur GPU, mais il reste toujours beaucoup de perspectives d'améliorations.

1.2.3 FPGA

Les FPGA (*Field Programmable Gate Array*) sont des circuits intégrés consistant en une matrice de blocs logiques dont l'interconnexion est reconfigurable, ce qui rend le circuit intégré effectivement programmable. Un bloc logique est généralement constitué de LUT (*Look-Up Tables*), de registres, d'additionneurs et de multiplexeurs dans des ratios dépendant de la gamme de FPGA et du vendeur ; les FPGA vendus par Altera présentent par exemple des blocs logiques organisés comme dans la figure 1.4. En plus de ces ressources, les FPGA intègrent d'autres ressources spécialisées les rendant plus polyvalents (intégration de blocs DSP, ~10 MB de blocs de RAM disponibles, entrées/sorties présentant des débits de l'ordre du Gbps). Historiquement utilisés pour du prototypage de circuits intégrés (*Application-Specific Integrated Circuit* : ASIC), la concentration en ressources qu'il est possible d'atteindre aujourd'hui ouvre d'autres possibilités d'utilisation, notamment pour le HPC. Dans ce domaine, les ressources spécialisées intégrées à la matrice du FPGA prennent tout leur

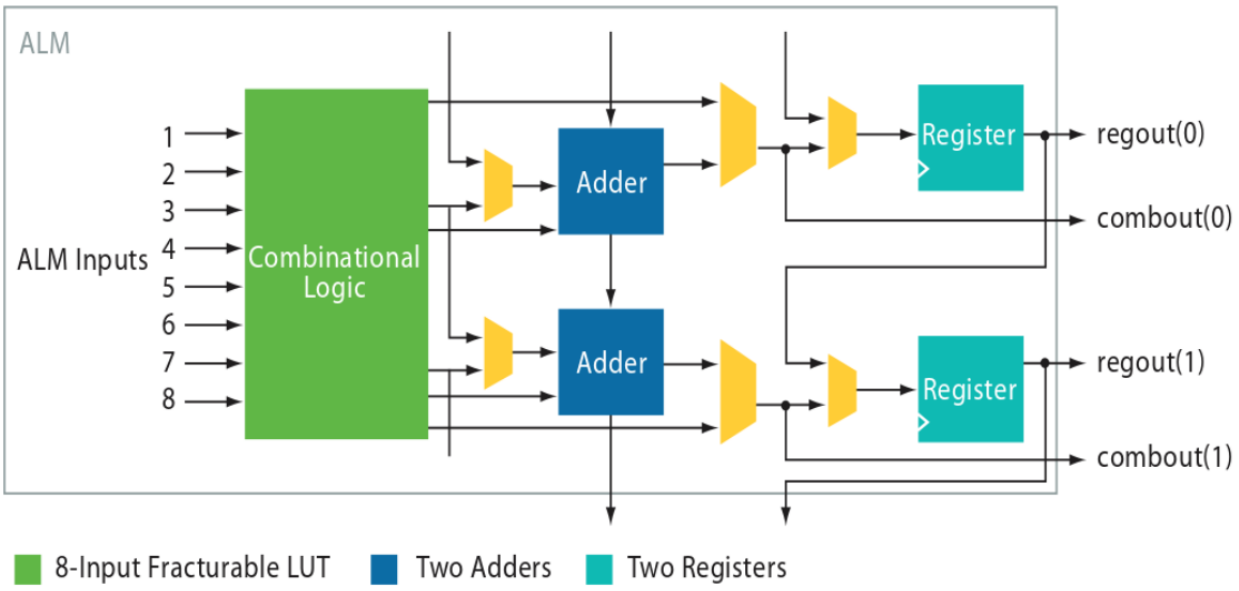

FIGURE 1.4 – Structure d'un ALM - Altera [Alt06]

sens. Le FPGA est difficile à classer comme cible de programmation. L'utilisation d'un FPGA passe par un déploiement d'une architecture matérielle, qui peut se baser par exemple sur un GPP ou un GPU. Cependant, nous nous intéressons ici au FPGA comme support pour des accélérateurs matériels dédiés. Dans ce contexte, ils peuvent donc être comparés aux autres cibles présentées précédemment.

De même que pour les autres cibles, il existe différentes gammes de FPGA. Celles-ci diffèrent de par le nombre et le type de ressources disponibles, la fréquence atteignable, le nombre d'entrées et sorties, la manière de les programmer. On s'intéresse ici uniquement aux FPGA à base de SRAM, qui constituent la plus grosse part du marché mondial, en particulier dans le domaine du HPC. Étant données les contraintes du domaine considéré, les FPGA étudiés font partie des gammes de produits spécifiques pour la performance, ce qui se traduit par des fréquences, tailles et consommation dans la fourchette haute des FPGA existants.

Même si les performances évoluent beaucoup ces dernières années, les FPGA pour le HPC atteignent généralement une fréquence de plusieurs centaines de MHz, pourvu que l'architecture soit optimisée pour atteindre ces performances. A l'inverse des GPP ou des GPU, cette fréquence représente plus fidèlement les performances de l'architecture, celle-ci étant généralement conçue pour traiter une ou plusieurs opérations par cycle d'horloge. Le degré de parallélisme atteignable sur ces architectures est très élevé et dépend des ressources utilisées par les opérations implémentées. En première approximation, on peut multiplier le nombre d'opérations jusqu'à consommer intégralement les ressources de calcul du FPGA. Le contrôle du concepteur sur le format des données, leur localisation par rapport aux opérateurs ainsi que le degré de parallélisme souhaité lui permet d'obtenir une adéquation parfaite entre l'algorithme parallélisable qu'il souhaite implémenter et

l'architecture matérielle du FPGA. L'optimisation spatiale du calcul donne accès à une manipulation du flot de donnée à un grain beaucoup plus fin que sur GPU ou GPP, pour des niveaux de performances importants à des fréquences plus basses, et donc des puissances requises plus faibles (de l'ordre de ~10 W). La taille d'un FPGA haut de gamme est typiquement équivalente à plusieurs dizaines de millions de portes logiques d'un ASIC.

Toutefois, le coût énergétique de l'utilisation de matériel pour l'accélération de calcul commence à être considéré comme un critère décisif de performance, au même titre que le temps de calcul ou la précision des résultats. Ceci fait du FPGA une cible au fort potentiel. En accord avec cette tendance, de Schryver et al. [dSSK+11] ont mis en place un banc de test pour comparer différents accélérateurs de calcul entre eux et pas simplement avec une référence logicielle.

Ils définissent un accélérateur de calcul d'options de la manière suivante :

— un problème, qui consiste généralement en la recherche de la valeur d'un produit financier,
— un modèle mathématique utilisé pour prédire le comportement de ce produit,
— une solution pour évaluer la valeur de ce produit, c'est-à-dire un algorithme et son implémentation.

Ce banc de test inclut la consommation énergétique comme critère discriminant entre plusieurs solutions (J/option). Ils ont appliqué cette méthodologie à l'exploration de l'espace des solutions pour l'évaluation d'options à barrière suivant un modèle d'Heston. Cela les a conduit à sélectionner une méthode de Monte Carlo Multi-Niveaux (présentée dans la section 1.3.2), qui forme le meilleur compromis trouvé via leur banc de test, en termes d'accélération, de précision et de consommation énergétique.

Une architecture sur FPGA est généralement définie dans un langage de bas niveau, dit *Hardware Description Language* (HDL), verbeux, complexe à déboguer et s'accompagnant de temps de développement longs. Des solutions plus haut-niveau récentes (HLS - *High Level Synthesis*) fournissent toutefois une alternative viable bien que demandant toujours un niveau d'expertise important [CLN+11]. Les caractéristiques de ce type de circuit intégré font que les temps de 'compilation' d'une architecture le ciblant sont généralement très longs, de l'ordre de ~1 à ~10 heures en fonction de la taille du FPGA et de l'architecture implémentée. Cette compilation est en effet similaire aux flots de conception matérielle. Elle consiste en la synthèse de l'architecture en des ressources matérielles, puis leur placement sur la matrice du circuit, associé au routage des différentes ressources (tout en obéissant à des contraintes en termes de fréquence minimale à atteindre).

Malgré ces contraintes de programmation, leur structure intrinsèquement parallèle, ainsi que leur capacité à implémenter des pipelines optimisés pour un calcul donné, rendent les FGPA séduisants pour le portage d'applications de calcul scientifique. Cependant, ils sont aujourd'hui très peu utilisés, leur utilisation étant souvent jugée trop complexe pour des experts logiciels.

1.2.4 Intégration des composants de calcul

1.2.4.1 Limites des solutions homogènes

Depuis plusieurs années, les solutions homogènes basées sur des GPP atteignent difficilement les contraintes des calculateurs utilisés dans le HPC. La multiplication de GPP pour augmenter la puissance de calcul atteinte ajoute des difficultés. En particulier, l'alimentation électrique et la dissipation thermique associées à de telles infrastructures peuvent rapidement entraîner des coûts d'exploitation difficilement justifiables. De plus, les gains en performances ne sont pas linéaires. La communication entre les processeurs et la complexité du modèle de programmation entament les performances d'une telle approche.

L'utilisation d'autres composants que le GPP a donc été envisagée. Cependant, à part le GPP, les autres composants décrits précédemment ne peuvent pas être déployés de manière autonome. Il est donc nécessaire de s'orienter vers des architectures hétérogènes. Cette tendance est visible dans le TOP500 des supercalculateurs, où les architectures hétérogènes figurent en bonne place [Top]. Ce constat s'applique également au calcul financier, où les coûts d'exploitation et la course à la performance poussent les entreprises vers ces solutions. Cependant, cette transition est lente et demande de modifier les habitudes de programmation des développeurs. Les difficultés sont telles que la plupart des acteurs maintient différentes solutions par manque de compétence ou de temps pour passer sur des solutions hétérogènes. La solution privilégiée reste le GPGPU, souvent considéré comme plus accessible pour les experts en logiciel que les solutions à base de FPGA.

1.2.4.2 solutions hétérogènes et communication inter-composants

La conception de machines hétérogènes passe par la capacité à communiquer entre différents composants de manière efficace. Cette communication se joue à la fois au niveau des liens physiques et au niveau applicatif (modèle de programmation). Traditionnellement, le protocole de communication le plus utilisé est le PCIe, qui est un bus secondaire permettant d'atteindre des débits élevés (16Go/s pour du Gen3 x16).

Toutefois, ces débits sont loin des débits sur les liens plus proches des processeurs. La latence est également plus élevée. Ce constat a mené à des tentatives de rapprochement des accélérateurs du GPP principal, voire même une intégration sur la même puce. Ainsi, AMD a conçu ses *Application Processing Unit* (APU), qui intègrent sur une seule puce des GPP, un GPU et divers accélérateurs, ainsi qu'un cache partagé. De même, Xilinx propose des *System-on-Chip* (Système sur Puce, SoC) à base de processeurs ARM, de FPGA et d'autres accélérateurs dédiés. Le rachat récent d'Altera par Intel (fin 2015) illustre bien la tendance vers une intégration au sein d'une même entité de technologies diverses et l'intérêt porté par les acteurs industriels à cette problématique.

L'utilisation de ces plateformes passe également par un modèle de programmation adapté. L'intérêt récent porté à ce domaine a entraîné une multiplication des environnements de programmation dédiés au calcul hétérogène. Ces solutions sont souvent proposées par différents constructeurs (ou consortiums). Parmi les plus marquants, on peut notamment citer les extensions d'*Open Multi-*

Processing (OpenMP) [tim], *Compute Unified Device Architecture* (Cuda) [NVIb], *Open Computing Language* (OpenCL) [Gro] et *Heterogeneous System Architecture* (HSA) [Fou].

Ces environnements de programmation s'appuient sur différents modèles. Le plus courant reste le SIMD. La communication entre le maître et les accélérateurs peut être soit explicite (requête spécifique faite par le développeur), soit implicite (gérée automatiquement par l'environnement). L'intégration du calcul réalisé par l'accélérateur dans le reste de l'application est également un point différenciant entre ces approches. Elle peut également s'effectuer explicitement par le développeur ou être détectée par l'environnement.

Dans ces travaux, nous nous intéressons principalement à OpenCL. Cette solution est en effet normalisée et connaît une adoption croissante par les constructeurs. Elle supporte également l'utilisation conjointe de GPU et de FPGA.

1.2.4.3 Standard de Programmation - OpenCL

OpenCL [Gro11] est un *framework* pour la programmation parallèle sur plateforme hétérogène. Ce *framework* est basé sur une librairie d'exécution sur hôte et sur du C99 étendu pour la programmation de l'accélérateur. Ce dernier est adapté pour supporter des types vectoriels de données, des points de synchronisation et d'autres fonctionnalités. OpenCL est un standard pour la programmation parallèle, qui est implémenté sur plusieurs architectures matérielles par leurs fabricants respectifs (ex. : GPU de NVidia et AMD, GPP d'Intel et d'AMD, FPGA d'Altera, etc.).

Un programme OpenCL peut être exécuté sur n'importe lequel de ces accélérateurs avec des modifications mineures d'un accélérateur à l'autre. Cette portabilité d'un programme OpenCL est l'un de ses avantages majeurs. Comme illustré sur la figure 1.5, un accélérateur OpenCL est divisé en *compute units* (unités de calcul), qui exécutent plusieurs copies (*work-items*) d'un code logiciel (i.e. un noyau, ou *kernel*). Toutefois, les performances d'une application OpenCL peuvent varier en fonction de l'adaptabilité de l'application et de l'architecture de l'accélérateur ciblé.

Le maître dans une architecture OpenCL (l'hôte) contrôle le flot de données de l'application à l'aide de files de tâches à exécuter sur les accélérateurs auxquels il est connecté. Le transfert de données est donc explicitement défini par le développeur. La gestion des données repose sur un modèle avec synchronisation et cohérence faible de la mémoire, consistant en trois niveaux de mémoire : global, local et privé. Les *work-items* sur un accélérateur sont organisés en *work-groups* qui partagent une zone mémoire commune (la mémoire dite locale), ainsi que des points de synchronisation. Ce niveau de mémoire sert de mémoire cache cohérente pour le développeur. Chaque *work-item* au sein d'un *work-group* a donc accès à toute donnée stockée dans l'espace mémoire local. La mémoire globale est accessible à partir de l'hôte et de tout *work-group*. En fonction de l'accélérateur, les accès à la mémoire globale peuvent être groupés pour réduire la latence, pourvu que les données demandées soient contiguës. Les accès à la mémoire globale sont toutefois toujours plus lents que les accès à la mémoire locale (jusqu'à un ordre de grandeur en débit). Enfin, chaque *work-item* possède un espace mémoire privé.

Chaque constructeur a la charge de développer un compilateur et le *runtime* qui permettent à

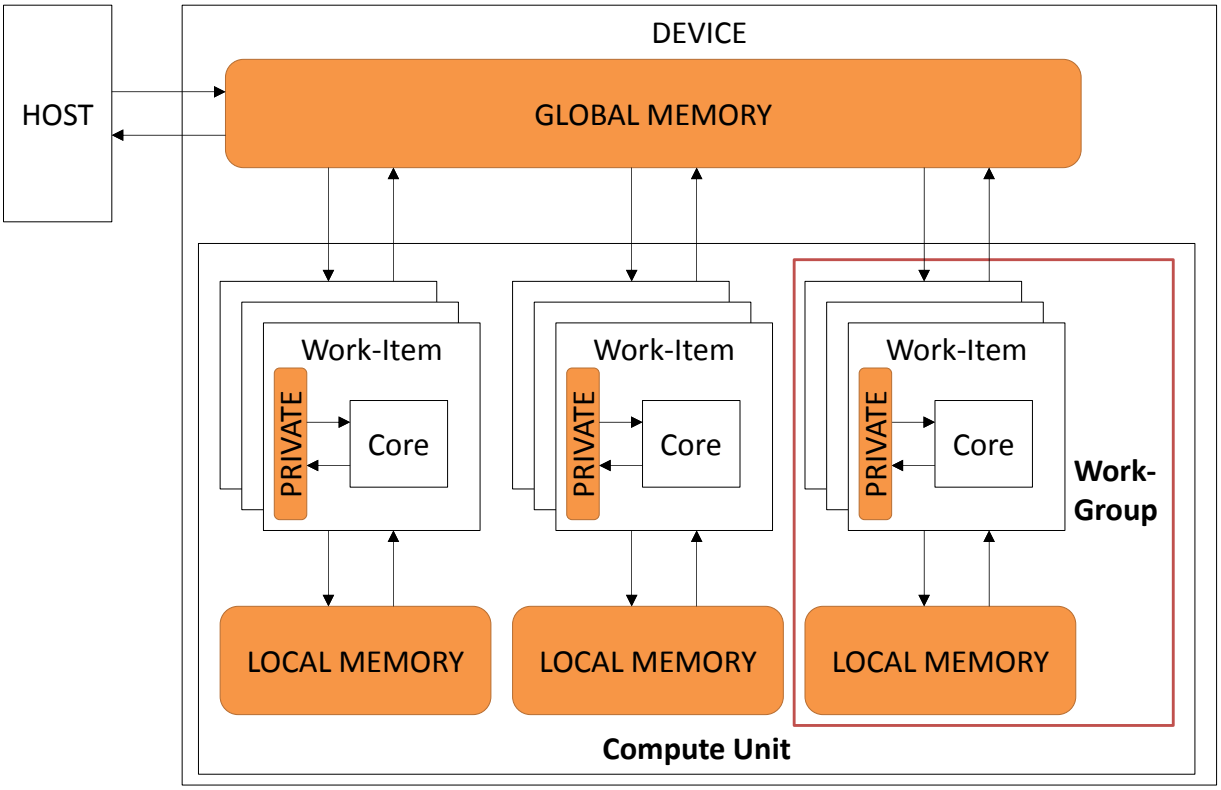

FIGURE 1.5 – Plateforme OpenCL générique

leur composant de respecter la norme OpenCL. Le compilateur d'Altera (*Altera OpenCL Compiler - AOC*) implémente aussi des *channels*, qui sont une implémentation native des FIFO définies dans le standard 2.0 d'OpenCL (*Pipes*). Ces FIFO permettent à des noyaux de s'échanger des données directement, sans copie de données en mémoire globale. Elles donnent aussi aux noyaux un accès au reste du système sans transfert explicite.

1.3 Méthodes d'implémentations des modèles d'évaluation

Les modèles présentés dans la section 1.1.2 sont des modèles mathématiques théoriques qu'il faut ensuite implémenter pour les utiliser. Cette implémentation est complexe et demande parfois de réaliser des compromis par rapport au modèle initial. Dans cette section, nous présentons des méthodes communes d'implémentation. Ces méthodes sont plus ou moins efficaces selon les modèles et l'architecture qui les exécute. Pour chacune de ces méthodes, nous donnons également des exemples d'implémentations de référence sur différentes cibles.

1.3.1 Discrétisation des modèles

Les modèles utilisés dans le calcul financier sont des modèles à temps continus. Il est donc nécessaire de les discrétiser pour les implémenter. Cette discrétisation n'est pas sans conséquence. Elle s'accompagne d'une perte en précision qu'il faut intégrer dans l'évaluation de la pertinence de la valeur modélisée. Qui plus est, le pas de discrétisation est complexe à choisir. Avec un pas trop grand, certains phénomènes ne seront pas pris en compte dans le modèle. Par exemple, une barrière d'option peut être manquée. Avec un pas trop petit, le temps de calcul augmente sensiblement, ce qui diminue l'intérêt de l'implémentation. En pratique, le pas classiquement utilisé est une fraction d'une journée.

Les méthodes de discrétisation existantes sont nombreuses. Elles sont plus ou moins adaptées au modèle, aux méthodes d'implémentation, au type d'option, ainsi qu'à la cible qui exécute cette implémentation. Les travaux d'Andersen et al. [AID12] détaillent des aspects dans le cas d'une cible matérielle. Dans cette section, on s'intéresse principalement à une discrétisation selon la méthode d'Euler, qui est la plus couramment utilisée. Cette méthode est une méthode du premier ordre qui s'appuie sur la connaissance d'un point de la courbe et de la tangente en ce point. La courbe est ensuite approximée en utilisant des segments de droite. Si le pas est suffisamment petit, on considère que l'erreur induite est acceptable.

Dans les sections suivantes, on confondra les modèles continus et les modèles discrétisés selon la méthode d'Euler.

1.3.2 Méthode de Monte Carlo

La méthode de Monte Carlo [Hul14] est couramment employée pour le calcul scientifique. Elle implique cependant une grosse charge de calcul pour atteindre des niveaux de précision acceptables. Dans le contexte financier, celle-ci est majoritairement utilisée pour évaluer des options exotiques. En effet, le niveau de complexité lié à l'utilisation de cette méthode est alors justifié. Dans le cas où l'on souhaite évaluer une option sur un panier de produits sous-jacents qui ne sont pas indépendants (comme par exemple des actions d'entreprises du CAC40), la méthode de Monte Carlo est la seule qui soit utilisable en pratique. Cette méthode se base sur l'utilisation d'un grand nombre de processus aléatoires pour évaluer la valeur numérique d'une variable. Les résultats de ces processus aléatoires sont ensuite moyennés pour obtenir une estimation de la variable recherchée.

Dans le contexte qui nous intéresse, l'objectif est d'évaluer la valeur actualisée d'une option. Pour ce faire, l'évolution du prix du produit sous-jacent est simulée sur un grand nombre de chemins générés à partir de nombres aléatoires. La valeur d'exercice de l'option est calculée sur chacun des chemins. La moyenne de ces valeurs d'exercice est ensuite calculée et actualisée à la date actuelle. Par exemple, dans le cas d'une option asiatique suivant le modèle de Black et Scholes, la valeur de

l'option évaluée par la méthode de Monte Carlo suit la formule suivante [Hul14] :

$$\begin{aligned} P(T) &= e^{-rT} \frac{\sum_{i=1}^{N_{path}} (\max 0, \sum_{j=0}^{N_{MC}} (S_j - K))}{N_{path}} \\ S(t) &= S(t-1) e^{(r - \frac{\sigma^2}{2})\delta t + \sigma\sqrt{\delta t} W(t)} \end{aligned} \tag{1.6}$$

Où $P(T)$ est la valeur de l'option à la date T, N_{path} est le nombre de chemins simulés, N_{MC} est le nombre de pas de chaque simulation et $\delta t = \frac{T}{N_{MC}}$. Dans le cas où le modèle initial est continu en temps, la méthode d'Euler est généralement utilisée pour discrétiser le modèle. Pour évaluer le biais, on utilise la racine carrée de l'erreur quadratique moyenne (*Root Mean Square Error*, RMSE). Pour la diminuer, il est possible d'agir essentiellement sur les variables N_{path} pour la variance et N_{MC} pour l'espérance. La méthode de Monte Carlo est très utilisée de par sa simplicité de mise en œuvre. Elle est également facilement parallélisable : la simulation des différents chemins indépendants peut être répartie sur plusieurs cœurs de calcul.

De nombreux articles traitent de l'accélération d'évaluation d'option par la méthode de Monte Carlo [TTTL10a][McW05][MBN10]. Toutefois, cette méthode fournit des résultats peu précis pour des nombres d'itérations faibles. Elle repose sur la loi des grands nombres (convergence de $\frac{1}{N}\sum_{i=1}^{N} X_i$ vers $\mathbb{E}[X]$ pour des variables aléatoires X_i). Le théorème central limite permet d'évaluer l'écart-type d'une telle méthode, proportionnel à $\frac{1}{\sqrt{N_{path}}}$. Il devient donc vite très coûteux de gagner en précision comparativement au nombre de chemins à simuler. Un certain nombre de méthodes existent pour corriger ce défaut, notamment :

— des méthodes de réduction de la variance [TTTL10b] avec variable de contrôle ;
— la méthode de Monte Carlo dite multi-niveau [dSSK$^+$11][MKK$^+$11] ;
— et la méthode de Quasi-Monte Carlo [TZKH10][AID12].

La méthode de Monte Carlo convient le mieux pour le calcul de modèles complexes, ou pour des problématiques comportant un nombre de dimensions conséquents (i.e. de variables d'entrées). Les modèles les plus complexes sont en général difficiles à calculer avec d'autres méthodes, tandis que les problèmes à grandes dimensions sont avantagés par la croissance linéaire en complexité en fonction de la dimension de la méthode de Monte Carlo.

La méthode de Monte Carlo et ses optimisations ont été minutieusement étudiées du fait de leur fort potentiel de parallélisation. L'accélération de cette méthode sur du matériel présentant une architecture parallèle est toute naturelle : il suffit de calculer différents chemins simultanément. On trouve des exemples sur GPU [AID12, MBN10], sur FPGA [SCH$^+$12, TTTL10b]. Les gains en accélération de cette méthode sont toutefois nuancés par son faible taux de convergence. Les implémentations présentées dans les travaux précédents sont efficaces mais sont développées pour des cas précis. Le moindre changement du cas d'utilisation demande une refonte plus ou moins complète de l'implémentation.

Récemment, des équipes se sont intéressées au développement de *framework* pour l'accélération des méthodes de Monte Carlo pour la finance. Avec une telle approche, le travail peut être réutilisé, ce

qui permet d'envisager un déploiement en production avec des cycles de développement acceptables. Une solution utilisant cette approche est le *framework* HyPer [BDSW15]. Cette implémentation cible un système Zynq et présente de bonnes performances : pour un budget énergétique équivalent à un GPP de serveur classique, leur approche présente un facteur d'accélération de 36 (multiples types d'options calculées, modèle d'Heston).

1.3.3 Arbres binomiaux et trinomiaux

Le premier modèle de Black et Scholes discrétisé a été proposé par Cox, Ross et Rubinstein en 1979 [CRR79] à l'aide d'un treillis, à savoir un arbre binomial. Un arbre binomial est structuré de telle sorte que chaque nœud de l'arbre est connecté à deux nœuds fils, ou aucun pour les nœuds feuilles. L'algorithme est dit *backward* ; il commence par calculer tous les prix possibles pour le sous-jacent à la date de maturité en considérant deux évolutions à chaque pas de temps : $S(t+1) = S(t)d, d < 1$ ou $S(t+1) = S(t)u, u > 1$. u et d correspondent aux facteurs d'augmentation ou de baisse du prix du sous-jacent pour un pas de temps. Ainsi, si le produit sous-jacent augmente entre t et $t+1$, son nouveau prix sera $u \times S(t)$. On a $u * d = 1$, ce qui assure que l'arbre est recombinant (voir figure 1.6). Pour un pas de temps T/N (avec T la date de maturité de l'option), on a donc des valeurs finales de S égales à $S(T,k) = S(t=0)u^{N-k}, k \in [1;N]$, avec k l'indice des feuilles de l'arbre. A $t = T$, les feuilles représentent donc l'ensemble des prix possibles pour le produit sous-jacent. Lorsque tous les nœuds finaux sont déterminés, on peut remonter l'arbre (*backtracking*) et calculer les valeurs de l'option de proche en proche jusqu'à arriver au nœud initial (t=0). On obtient alors la valeur de l'option.

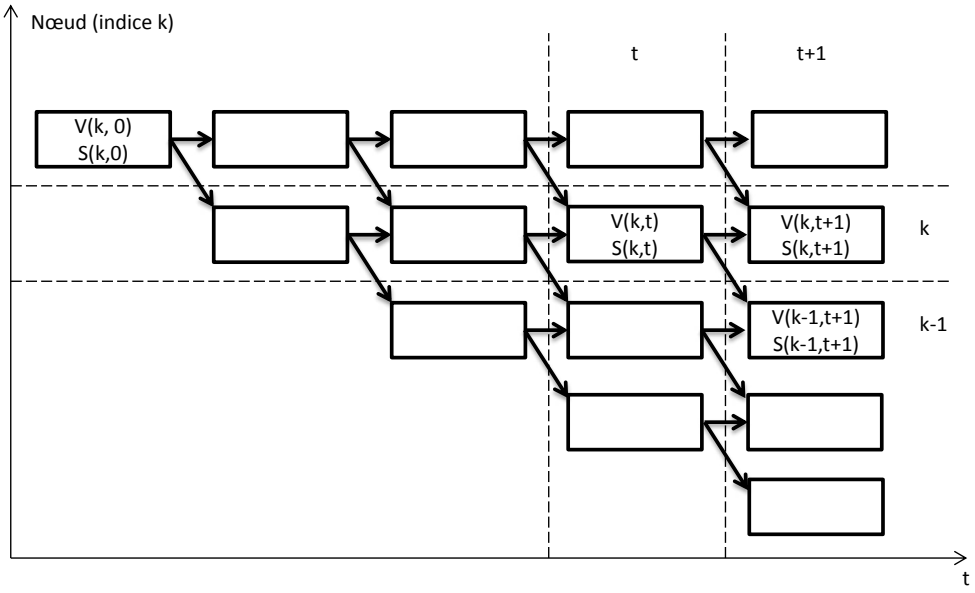

FIGURE 1.6 – Arbre binomial : dépendance des nœuds

Par exemple, dans le cas d'une option américaine (call), la formule de récurrence est la suivante

[Hul14] :

$$P_{call}(t,k) = \max\left(S(t,k) - K, e^{-rT/N}(pP_{call}(t+1,k) + (1-p)P_{call}(t+1,k-1)\right)$$

Avec r le taux d'intérêt sans risque, p et $1-p$ sont respectivement les probabilités que le prix du sous-jacent augmente ou diminue.

Un cas d'étude de l'implémentation de cette méthode sur plusieurs cibles est présenté dans le chapitre 3.

Il est possible d'utiliser un treillis plus complexe, en considérant que le prix du sous-jacent puisse être stable entre deux instants consécutifs (arbre trinomial).

Le modèle binomial d'évaluation d'option est moins étudié, du fait de sa complexité à implémenter de manière efficace (et ce malgré les gains en temps de calcul). Le *backtracking* a en effet un coût non négligeable en termes d'empreinte mémoire et de temps de calcul. Il existe cependant plusieurs travaux d'intérêt sur l'accélération de cette méthode. Une approche pour le GPU est ainsi décrite par Phar *et al.* [PF05]. Sur FPGA, la complexité réside dans le stockage de l'état de l'arbre pendant le *backtracking*. La mémoire disponible est souvent moins importante et plus difficile d'accès que sur GPU. Des approches ont été proposées par plusieurs équipes [JTLC08, WMI09, TT14]. Ces solutions sont présentées plus en détail dans le chapitre 3.

1.3.4 Méthode des différences finies

Les méthodes des différences finies permettent d'approcher des équations différentielles partielles avec des conditions aux limites. Elle repose sur une approximation de la dérivée d'une fonction par la formule de Taylor. Soit une fonction f dérivable n fois, $n \in \mathbb{N}$, sur l'intervalle $[x-h; x+h]$. Dans le cas de la méthode des différences finies explicites, trois approximations similaires sont utilisées [Hul14] :

- L'approximation centrée :

$$\frac{f(x+h) - f(x-h)}{2h} = \frac{df}{dx}(x) + \frac{h^2}{6}\frac{d^3f}{dx^3}(x) + h^2\epsilon(x,h)$$

  où ϵ tend vers 0 quand h tend vers 0.

- L'approximation avec décentrage aval :

$$\frac{f(x) - f(x-h)}{h} = \frac{df}{dx}(x) + \frac{h^2}{6}\frac{d^3f}{dx^3}(x) + h^2\epsilon(x,h)$$

- L'approximation standard (pour le second ordre) :

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} = \frac{d^2f(x)}{dx^2} + \frac{h^2}{6}\frac{d^3f}{dx^3} + h^2\epsilon(x,h)$$

Dans le cas du modèle de Black et Scholes, on peut exprimer l'évolution du prix de l'option

sous la forme d'une équation différentielle partielle :

$$\frac{\partial P}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 P}{\partial S^2} + rS\frac{\partial P}{\partial S} - rP = 0 \tag{1.7}$$

La discrétisation donne alors :

$$\begin{aligned} rP(t,k) =& \frac{-P(t-1,k) + P(t,k)}{\Delta t} + rk\Delta S \frac{P(t,k+1) - P(t,k-1)}{2\Delta S} \\ &+ \frac{1}{2}\sigma^2 (k\Delta S)^2 \frac{P(t,k+1) + P(t,k-1) - 2P(t,k)}{(\Delta S)^2} \end{aligned}$$

Ce qui donne :

$$P(t-1,k) = a_k P(t,k-1) + b_k P(t,k) + c_k P(t,k+1) \tag{1.8}$$

Avec

$$\begin{aligned} a_k &= \frac{1}{2}\Delta t(\sigma^2 k^2 - rk) \\ b_k &= 1 - \Delta t(\sigma^2 k^2 + r) \\ c_k &= \frac{1}{2}\Delta t(\sigma^2 k^2 + rk) \end{aligned}$$

On peut représenter graphiquement cette méthode comme l'application d'un gabarit ("stencil") sur une grille, ou treillis (voir figure 1.7). La figure 1.7 illustre le calcul d'un prix possible pour l'option à (t,k), à partir des prix à $(t+1,k+1)$, $(t+1,k)$ et $(t+1,k-1)$ en suivant la formule 1.3.4.

Cette méthode demande de connaître des conditions aux limites. Celles-ci sont dépendantes du modèle et du type d'option à évaluer. Lorsque celles-ci sont connues, on peut remonter itérativement la grille à partir de la date de maturité pour obtenir la valeur de l'option. On obtient alors un ensemble de valeurs à $t = 0$. La valeur réelle de l'option est calculée par interpolation en fonction de la valeur du sous-jacent à $t = 0$. Cette méthode converge en $O(\Delta t)$ et en $O((\Delta S)^2)$ pourvu qu'elle respecte la condition de stabilité $\frac{\Delta S}{\Delta t} \leq \frac{1}{2}$.

La méthode de Crank Nicholson présente un meilleur taux de convergence et est toujours stable. Elle repose sur l'équation suivante :

$$\begin{aligned} &-\bar{a}_k P(t-1,k-1) + (1-\bar{b}_k)P(t-1,k) - \bar{c}_k P(t-1,k+1) = \\ &\bar{a}_k P(t,k-1) + (1+\bar{b}_k)P(t,k) + \bar{c}_k P(t,k+1) \end{aligned} \tag{1.9}$$

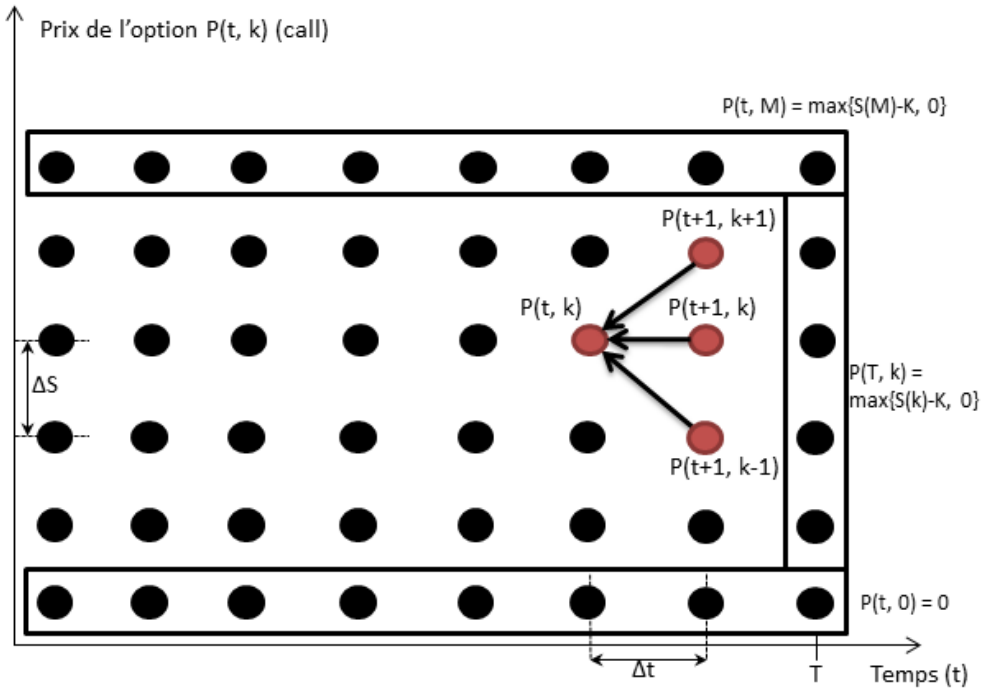

FIGURE 1.7 – Grille pour la méthode des différences finies explicites

Avec

$$\bar{a}_k = \frac{\Delta t}{4}(\sigma^2 k^2 - rk)$$
$$\bar{b}_k = -\frac{\Delta t}{2}(\sigma^2 k^2 + r)$$
$$\bar{c}_k = \frac{\Delta t}{4}\Delta(\sigma^2 k^2 + rk)$$

La méthode de Crank Nicholson demande de résoudre un système de M équations (où M est le nombre de niveaux de prix dans la grille) à une date t, puis de remonter itérativement jusqu'à $t = 0$. La résolution d'un tel système correspond à un calcul matriciel avec une matrice tridiagonale, i.e. du type :

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & 0 & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} \\ 0 & 0 & a_{4,3} & a_{4,4} \end{pmatrix}$$

La méthode de Crank Nicholson converge en $O((\Delta t)^2)$ et en $O((\Delta S)^2)$, sans la contrainte de stabilité présente avec la méthode implicite.

Un des défauts des méthodes des différences finies est qu'elles ne sont pas optimales pour des problèmes à grande dimension (typiquement, l'évaluation d'une option sur plusieurs sous-jacents). La méthode de Monte Carlo présente de meilleurs résultats dans ce cas d'utilisation (avec une convergence linéaire en fonction du nombre de dimensions).

Concernant l'implémentation de ces méthodes, on remarque que les méthodes de différences finies sont proches des modèles binomiaux et peuvent être implémentés de manière similaire. Les différences principales sont visibles sur les effets de bord et les contraintes de convergence. Ces méthodes se prêtent bien à une implémentation sur accélérateur, notamment la méthode explicite et la méthode de Crank-Nicholson. Les auteurs de [JLT11b] ont mis en place un *framework* d'évaluation d'option de styles divers (option européenne, américaine, asiatique) sur FPGA en simple et double précision flottante. Les auteurs de [CKW12], quand à eux, décrivent l'implémentation de la méthode explicite et de la méthode de Crank-Nicholson pour des options européennes exclusivement. Des implémentations sur GPU peuvent aussi être trouvées [DCJ10], pour des options basées sur plusieurs sous-jacents. Ces articles traitent toutefois uniquement le modèle de Black et Scholes.

1.3.5 Calcul numérique d'intégrales ("Quadrature method")

La plupart des modèles décrits précédemment peuvent être représentés sous la forme d'une intégrale, mais celle-ci n'a pas de solution analytique. Les méthodes usuelles d'approximation d'intégrale peuvent alors être utilisées pour parvenir à une estimation de la valeur de l'option. Si l'on reprend l'équation différentielle de Black et Scholes (1.7) et qu'on applique les changements de variables suivants :

$$x = \log\left(\frac{S(t)}{K}\right)$$
$$y = \log\left(\frac{S(t+\Delta t}{K}\right)$$

Il est possible d'obtenir le résultat suivant :

$$P(x,t) = A(x)\int_{-\infty}^{\infty} B(x,y)P(y,t+\Delta t)\,dy \tag{1.10}$$

Avec

$$A(x) = \frac{1}{\sqrt{2\sigma^2\pi\Delta t}}e^{(-((\frac{2r}{\sigma^2}-1)\frac{x}{1})-\sigma^2(\frac{2r}{\sigma^2}-1)^2\frac{\Delta t}{8}-r\Delta t)}$$
$$B(x,y) = e^{-(\frac{(x-y)^2}{2\sigma^2\Delta t}+(\frac{2r}{\sigma^2}-1)\frac{y}{2})}$$

Cette intégrale peut alors être approchée en la discrétisant sur un grand nombre d'intervalles identiques (n). Les méthodes les plus utilisées pour cela sont la méthode du trapèze et la méthode de Simpson. Pour approximer une intégrale $\int_a^b f(x)\,dx; a, b \in \mathbb{R}$ et en notant $h = \frac{b-a}{n}$, la méthode

du trapèze donne l'approximation suivante :

$$\int_a^b f(x)\,dx = h(\frac{f(a)}{2} + \sum_{k=1}^{n-1} f(a+kh) + \frac{f(b)}{2}) \tag{1.11}$$

La méthode de Simpson donne le résultat suivant :

$$\int_a^b f(x)\,dx = \frac{h}{2}(f(a) + \sum_{k=1}^{n/2-1} f(a+2hk) + \sum_{k=1}^{n/2} f(a+2hk-1) + f(b)) \tag{1.12}$$

Les travaux présentés dans [TTL12] utilisent la méthode de Simpson (au taux de convergence plus élevé) pour approcher l'intégrale (1.10), pour divers types d'options (européenne, à barrière et américaine) et pour des options avec deux sous-jacents (i.e. une double intégrale). Le coût en complexité pour une augmentation en dimension est toutefois important ; les auteurs déclarent qu'il faut $O(n^d(2d^2+4d+1))$ opérations flottantes pour calculer une option européenne avec cette méthode, avec d sous-jacents, sachant que ce nombre est plus important pour des options exotiques. Jin *et al.* [JLT11a] considèrent que le calcul numérique d'intégrales (Simpson) forme le meilleur compromis pour évaluer des options. Ils comparent toutefois plusieurs méthodes sans en décrire ni les implémentations, ni les différentes variantes des méthodes utilisées (par exemple, les méthodes de réduction de variance pour Monte Carlo), ce qui rend leur résultats difficiles à interpréter.

1.3.6 Conclusion sur les méthodes d'implémentations

L'implémentation des modèles d'évaluation d'options peut se faire en utilisant différentes méthodes de calcul. Ces méthodes ont chacune leurs avantages et inconvénients, en particulier en termes de précision et de complexité. Le choix de la méthode à utiliser dépend à la fois du modèle choisi et du composant ciblé. Toutes ces méthodes ont été étudiées et implémentées sur différentes cibles. Cependant, peu d'études ciblent différents modèles ou méthodes. Chaque implémentation d'un couple modèle méthode donne donc lieu à une nouvelle exploration d'architecture, ce qui freine fortement l'adoption d'accélérateurs de calcul en production.

1.4 Un besoin en accélération de calcul dans le domaine du calcul financier

Dans les sections précédentes, nous avons présenté le domaine d'application en termes de produits financiers et les problématiques de modélisation et d'implémentation rencontrées. Dans le domaine du calcul financier, l'expertise demandée aux personnes chargées de concevoir et d'implémenter des modèles de calcul est nécessairement focalisée sur leurs compétences en mathématique appliquée et en génie logiciel. Ils se doivent toutefois de rester maîtres de l'ensemble des solutions mises en place dans leur établissement financier. En effet, les particularités d'implémentation des modèles sont la plupart du temps tenues pour secrètes, afin d'éviter d'être prédictible sur le marché. Dans ce

cas de figure, l'ensemble des développements est réalisé en interne, du modèle mathématique au déploiement.

De plus, la concurrence forte du domaine se traduit par une évolution en complexité des modèles utilisés, pour mieux refléter la valeur des produits financiers sur le marché. Entre les recherches académiques réalisées et les optimisations propres à un établissement, de nouvelles versions des modèles sont déployées chaque année et coexistent avec les modèles précédents. Le cadre législatif évolue aussi pour limiter toute dérive du marché. Ce cadre est différent en fonction des pays (par exemple, Bâle III en Union Européenne), même si une base internationale commune existe. Cette course à l'innovation est donc aussi portée par les autorités de régulation, qui impose la mise en place d'outils de gestion de risque qui nécessite de réaliser des calculs relativement complexes. Ces changements chroniques imposent des mises à jour régulières des solutions implémentées. Ces mises à jour ne sont pas forcément triviales et peuvent nécessiter une refonte complète de l'implémentation. C'est pourquoi la complexité du développement sur plateforme hétérogène (à base de FPGA et/ou GPU) est un frein à leur adoption.

Cependant, la concurrence entre acteurs financiers, ainsi que les contraintes législatives du marché, se traduisent par une demande croissante en puissance de calcul. La réponse de l'industrie à cette charge de calcul est l'utilisation de fermes de serveurs, où les calculs sont parallélisés sur un grand nombre de GPP. Comme nous l'avons vu dans les sections précédentes, cette solution ne permet plus de suivre simplement l'augmentation de puissance requise. Le passage sur solution hétérogène devient donc nécessaire et on se retrouve face à une impasse.

Les principales problématiques du HPC en finance sont donc liées aux coûts en ressources pour obtenir la puissance de calcul requise d'une part et aux difficultés de programmation de solutions efficaces d'autre part. Réduire la consommation énergétique des solutions d'accélération a deux effets principaux. Le coût annuel des systèmes implémentés a un impact sur leur rentabilité. Réduire ce coût permet soit d'économiser une partie du budget alloué au calcul financier, soit de déployer plus de ressources pour un coût identique. Le deuxième avantage de la réduction de la consommation énergétique est de pouvoir fonctionner sur un budget en puissance moindre, ce qui relâche des contraintes sur l'implantation des locaux, ainsi que leur évolution possible. La proximité des serveurs et des salles de marché est un point crucial pour réduire le temps d'accès à l'information du marché. L'infrastructure urbaine ne peut supporter qu'une puissance limitée, qui contraint donc le nombre de serveurs qui peuvent être alimentés. Un exemple récent de cette contrainte a été observé durant les Jeux Olympiques de Londres 2012, où la puissance électrique de la City a été régulée pour s'accorder avec les activités de la ville de Londres [War08].

Les travaux présentés dans cette thèse ont été en partie menés en partenariat avec des acteurs du monde financier. Le but de ces travaux est de rendre accessible les accélérateurs matériels (GPU ou FPGA), qui sont vus par ces acteurs comme étant une technologie clé pour l'avenir du domaine. Cependant, ils n'ont pour l'instant ni les compétences, ni les technologies leur permettant d'atteindre des performances satisfaisantes en puissance de calcul et en latence pour pouvoir évaluer et utiliser ces solutions. L'objectif pour ces acteurs est de parvenir à une utilisation transparente

des accélérateurs à partir d'un environnement logiciel. Cet objectif n'est pas propre au monde de la finance, et il est déjà étudié de manière générique (HLS, environnements logiciels). Ces approches, bien qu'intéressantes, ne sont pas encore matures et ne répondent pas complètement au problème [CLN+11]. Nous avons donc exploré trois axes originaux et différents, qui se basent sur des approches distinctes, mais qui sont conçues spécifiquement pour le calcul financier.

Dans un premier temps, nous nous sommes concentrés sur une approche générique et dont la prise en main est aisée pour un développeur financier. Celle-ci repose sur l'intégration de noyaux d'accélération dans une librairie logicielle de calcul financier, QuantLib [Qua]. Nous y présentons la répartition des calculs entre couche logicielle et accélérateur, ainsi que le lien avec une décomposition logique d'un modèle financier. L'utilisation d'une librairie existante permet en effet de bénéficier d'une description déjà existante du domaine, qu'il faut traduire pour une implémentation hétérogène. Cette traduction peut être réalisée de manière itérative, pour intégrer sur accélérateur des parties de plus en plus conséquentes du calcul.

Dans une seconde partie, nous présentons une implémentation d'un algorithme financier relativement lourd à calculer à l'aide d'OpenCL en tant qu'outil de HLS. Le flot OpenCL est séduisant pour sa capacité à cibler plusieurs accélérateurs matériels (GPU et FPGA dans le cas présent) au prix de modifications mineures du code, généralement une simple modification du paramétrage des noyaux. Nous avons exploré diverses architectures, sur GPU et FPGA et les avons comparées à l'état de l'art. Cela nous a permis d'estimer le rendement d'une approche spécialisée d'implémentation de modèles financiers en termes de performances et de coût en temps de développement associé. En opposition à la première partie, cette approche, du HLS vers le domaine financier, permet de déterminer les contraintes liées à l'utilisation d'accélérateur matériel, ainsi que les limites des spécialisations d'une implémentation répondant à un problème restreint.

Enfin, nous présentons une chaîne de compilation dont l'optique est d'ouvrir la programmation de noyaux OpenCL à un développeur financier. Les parties de l'application à accélérer sont extraites du programme complet, pourvu qu'elles obéissent à certaines contraintes que nous décrivons. Ces contraintes sont spécifiques au domaine financier et à la catégorie de méthodes de calcul supportées par l'outil. Le fait de positionner cet outil à l'étape de compilation permet au développeur de s'affranchir de la gestion et de la caractérisation de la plateforme d'accélération. Une plateforme spécifique intégrant une puce FPGA a été étudiée pour en extraire des caractéristiques de latence de communication et de performance d'opérateurs élémentaires. Ceci fournit des métriques pour estimer les performances qu'il est possible d'espérer lors de l'implémentation d'un modèle sur cette cible. Ces étapes sont gérées en amont du flot, ôtant ainsi au développeur une tâche qui est hors de son domaine de compétence.

1.5 Conclusion

Le contexte de ces travaux a été présenté dans ce chapitre. Nous avons défini le domaine d'application, à savoir le calcul d'options financières et avons défini plusieurs modèles d'évaluation

de celles-ci. Nous avons ensuite introduit les plateformes de calculs susceptibles d'être utilisées pour l'implémentation des modèles de calcul d'options, dans un contexte de calcul haute performance. Nous avons présenté l'intérêt de solutions matérielles hétérogènes pour diminuer les temps de calcul (à un budget énergétique identique ou inférieur), ainsi qu'OpenCL, standard de programmation pour une telle infrastructure.

Pour résoudre un problème financier concret, comme par exemple l'évaluation d'un type d'option particulier, pour une précision fixée, le choix d'un modèle n'est pas suffisant. Il faut aussi considérer son implémentation sur matériel, ce qui demande de choisir une méthode d'implémentation adaptée parmi l'état de l'art. Les méthodes d'implémentations les plus communes ont été décrite dans ce chapitre et incluent la méthode de Monte Carlo, l'utilisation d'arbres binomiaux ou trinomiaux, plusieurs méthodes des différence finies ainsi que le calcul numérique d'intégral.

Les besoins de l'industrie en termes de puissance de calcul se confrontent aux limitations des couches matérielles : les GPP traditionnellement utilisés atteignent leurs limites, tandis que la complexité en programmation des solutions à base de GPU et FPGA est pénalisante et ne leur permet pas d'atteindre des temps de mise sur le marché compétitifs. Nous proposons donc une approche spécifique au domaine financier pour ouvrir l'accès en programmabilité des plateformes hétérogènes. L'objectif est de fournir une solution permettant de développer rapidement des solutions de calcul d'options, qui atteigne des niveaux de performances élevés en termes d'efficacité énergétique et d'accélération de calcul.

Chapitre 2

Intégration de noyaux matériels dans la librairie financière QuantLib

DANS le chapitre précédent, nous avons présenté la problématique au cœur de l'accélération de calcul d'options financières. Nous avons décrit tous les éléments inclus lors du calcul d'une option, de sa modélisation à son implémentation sur plateforme matérielle. Nous en avons conclu que faciliter l'accès en programmabilité de plateformes hétérogènes présente des avantages certains pour le domaine de la finance quantitative. Dans ce chapitre, nous proposons une première approche de cette idée, en partant du domaine d'application. Nous décrivons une couche logicielle permettant l'intégration de noyaux de calcul accélérés à une librairie de calcul financier *Open Source*, QuantLib. L'approche a pour objectif de conserver les abstractions spécifiques au domaine et déjà présentes dans QuantLib et d'intégrer à celle-ci une accélération matérielle, de manière transparente pour l'utilisateur.

Dans un premier temps, nous décrivons la librairie financière utilisée. Nous présentons ensuite une couche d'abstraction pour l'intégration logicielle d'accélérateurs, pour enfin l'appliquer sur l'utilisation de QuantLib.

2.1 Librairie QuantLib - la finance quantitative en Open Source

QuantLib (*Quantitative finance Library*, [Qua]) est une librairie logicielle Open Source (licence BSD) pour le calcul financier (modélisation, trading et gestion de risque). QuantLib inclut notamment des outils de modélisation d'options, de simulation de courbes de taux, de courbes de volatilité, ainsi que des méthodes de calcul telles que décrites dans le chapitre 1 (méthodes de Monte Carlo, méthodes à base de calcul d'arbres, méthode des différences finies). QuantLib est initialement une librairie C++, avec une représentation sous forme de classe des éléments qu'elle implémente. Elle présente néanmoins des interfaces dans d'autres langages, comme par exemple Python (via l'utilisation de SWIG [swi] pour l'interfaçage).

L'ensemble de la librairie est relativement bien documentée. Elle présente un manuel de référence, des tutoriels de prise en main et des exemples dans le répertoire source. Un livre est aussi en cours de rédaction par Luigi Ballabio, l'un des contributeurs majeurs de la librairie (*"Implementing QuantLib"*, les *drafts* sont disponibles sur le blog de l'auteur [Bal]). QuantLib est connue dans le domaine, même si son taux d'utilisation n'est pas aisé à évaluer. Elle est utilisée en industrie, notamment par StatPro Italia, qui emploie le contributeur principal de l'outil. Même quand elle n'est pas déployée dans sa totatité en production, QuantLib est toutefois clé en tant que *benchmark* de référence et exemple d'implémentation logicielle. La disponibilité de l'outil assure également la reproductibilité de résultats expérimentaux.

La structure de la librairie et l'organisation stricte des éléments spécifiques du domaine en classes présente de nombreux avantages. Notamment, cela facilite l'ajout d'éléments nouveaux, que ce soit des modèles de produits dérivés, une extension des moteurs de calculs ou une simple optimisation d'un élément de la librairie. QuantLib intègre aussi des *Design Pattern* (patrons de conception) qui sont adaptés pour le calcul numérique (tel que le *Curiously Recurring Template Pattern* - CRTP - pour gérer l'héritage de classe), ou adaptés au comportement de données financières, comme par exemple la gestion "paresseuse" de produits sous-jacents côtés en bourse. L'évaluation d'un produit sous-jacent par un modèle paresseux n'est effectuée que lorsque sa valeur est effectivement demandée par un utilisateur. Ainsi, une classe utilisatrice appelle une méthode interne à la classe du produit sous-jacent pour obtenir sa valeur actuelle. La surcouche QuantLib sur le C++ standard inclue aussi une gestion d'erreur étendue (inspirée du fonctionnement de Boost [boo]), ainsi que des types de donnée supplémentaires et un calendrier avancé, critique pour évaluer des produits réels.

La contrepartie de cette architecture est sa complexité. La première utilisation de QuantLib en dehors d'un simple tutoriel demande d'extriquer du code les interactions entre de nombreux objets, ceux-ci ayant un sens du point de vue financier, mais n'étant pas optimaux du point de vue de la conception logicielle. La multiplicité des niveaux d'abstraction ajoute un niveau de complexité lors de l'extension d'une librairie. Déterminer à quel niveau d'abstraction se placer demande de se familiariser avec son fonctionnement usuel.

Dans le cas qui nous intéresse, QuantLib inclut des *frameworks* de calcul numérique, pensés pour le domaine de calcul financier. Ces *frameworks* forment des collections de classes qu'il est possible

d'utiliser pour mettre en place un moteur de calcul d'options. De manière générale, QuantLib conserve la division entre la définition d'un produit financier (aussi nommé instrument financier) et sa méthode d'évaluation. Ces deux éléments sont représentés par deux classes, une classe dite *Instrument* et une classe dite *Pricing Engine* qui correspond au moteur de calcul proprement dit. Cette architecture suit un patron de conception logicielle dit *Strategy Pattern*. L'intérêt principal de cette division est la possibilité de changer la méthode d'évaluation durant l'exécution d'un programme, en fonction des besoins de l'utilisateur. Les données transférées entre ces deux classes sont elles-mêmes encapsulées dans des classes *results* (résultats) et *arguments*. Ceci permet d'assurer un niveau d'abstraction suffisant pour qu'un *Intrument* soit compatible avec plusieurs *Pricing Engine*. Ces deux classes de stockage de données masquent les spécificités de l'*Instrument* (par exemple, le critère d'arrêt pour une option à barrière). En pratique, le flot d'exécution de ces deux classes est très intriqué et une telle séparation est difficile à envisager sur accélérateur.

On s'intéresse ici plus particulièrement au *tree framework*, qui modélise des arbres (binomiaux et trinomiaux), ainsi que des treillis. Un diagramme séquentiel représentant les liens entre les différents éléments est présenté dans la figure 2.1. La création d'un moteur de calcul d'options à l'aide de ce *framework* passe par l'utilisation de la classe *TreeLattice* (notée *Lattice* dans le diagramme), qui implémente un treillis basé sur un arbre, en conjonction avec un produit sous-jacent discrétisé (héritant de la classe *DiscretizedAsset*). QuantLib propose par défaut plusieurs sous-classes d'arbres, en fonction des paramètres de parcours des branches, tous héritant d'une classe mère *Tree* et implémentée en suivant le patron dit CRTP, pour réduire les temps de calcul. Un *DiscretizedAsset* contient des méthodes pour modifier sa valeur, en tenant compte de ses spécificités (dépendant de sa modélisation : modèle de Black et Scholes, modèle à volatilité stochastique, *etc.*). Une fois le *DiscretizedAsset* initialisé à l'aide des structures de données d'entrée requises (taux d'intérêt, volatilité, *Quote* pour sa valeur initiale sur le marché, *etc.*), le treillis *TreeLattice* peut ensuite initialiser la matrice (*Array* dans le code source) contenant ses données internes et dérouler le calcul d'une ou plusieurs étapes temporelles (voir figure 2.1, extraite du draft d'Implementing QuantLib [Bal]). La matrice en elle-même ne stocke qu'un pas temporel à tout instant et le déroulage jusqu'au pas temporel demandé s'effectue par itérations successives d'un pas de discrétisation temporel (opération de *stepback()*). L'association de ce treillis et de ce produit sous-jacent discrétisé permet donc de modéliser l'évolution du produit sous-jacent au cours du temps. Le calcul est contrôlé par un objet *Option* (non représenté sur le diagramme séquentiel 2.1), qui utilise ce moteur de calcul de sous-jacent pour son évaluation.

À notre connaissance, une seule tentative d'accélération de QuantLib a été réalisée. Les travaux de Grauer-Gray *et al.* décrivent l'implémentation de plusieurs portions spécifiques de la librairie sur GPU. Ces portions sont extraites après des séries de tests de perfomances à l'aide d'un logiciel de profilage pour déterminer quelles sont les opérations dont le portage sur matériel est susceptible de conduire à un facteur d'accélération intéressant. Ils comparent les performances de plusieures chaînes logicielles (NVidia Cuda, OpenCL, HMPP et OpenAcc), ainsi que plusieurs gammes de GPU. Les mêmes portions ont été accélérées sur GPP en utilisant OpenMP pour atteindre un facteur de

parallélisation de 8 (utilisation de plusieurs cœurs de GPP). Afin de garantir une comparaison équitable, les portions sélectionnées ont été réécrites pour chacune des cibles envisagées. Les portions de la librairie qu'ils ont accélérées présentent des facteurs d'accélération importants (environ deux ordres de grandeur) même par rapport à la réécriture OpenMP. Toutefois, ils parviennent à ces performances après une mise à plat de tous les niveaux d'abstraction (*code flattening*), qui consiste en l'excision de toute abstraction haut-niveau du code. À l'exception des exemples qu'ils décrivent (solution analytique du modèle de Black et Scholes pour des options européenne, méthode de Monte Carlo sur un profil d'option particulier, *etc.*), leur approche présente peu d'intérêt, si ce n'est en tant que (prometteuse) preuve de concept.

Cette approche ne coïncide pas avec l'esprit de QuantLib et s'accompagne de conséquences indésirables : manque d'adaptabilité (dans le sens du rajout ultérieur de fonctionnalités), opacité pour l'utilisateur, implémentation de l'ensemble de QuantLib difficilement envisageable. Elle n'est également pas utilisable dans notre cas, vu qu'elle n'est pas généralisable et qu'elle présente une mauvaise utilisation de ressources.

Dans le cadre de cette thèse, des discussions ont eu lieu avec L. Ballabio, afin d'étudier la faisabilité et l'intérêt d'une accélération plus générique de QuantLib. Suite à cette discussion, nous nous sommes dirigés vers une approche *Bottom-Up* de développement d'une API pour l'accélération de portions de QuantLib. Malgré son intérêt pour ce type de travaux, QuantLib n'a pas vocation à être porté sur accélérateur par ses développeurs principaux. Ils ne se dirigent donc pas vers ce type d'approche. Dans la suite de ce chapitre, nous présentons nos résultats suite à une exploration des possibilités offertes par QuantLib dans le cadre de nos travaux. Cette exploration nous a mené à proposer une première méthodologie d'intégration d'accélérateurs matériels pour la finance qui sera présentée dans la section suivante.

2.2 Couche d'abstraction pour l'intégration logicielle d'accélérateurs matériels

2.2.1 Approche d'intégration à grain fin

Nous présentons ici une méthodologie d'intégration d'accélérateurs matériels au sein d'une librairie de calcul financier. Cette méthodologie repose sur une couche d'abstraction du matériel pour masquer son intégration à un développeur logiciel. Une API est utilisée pour tous les accès matériels et implémentée par un spécialiste pour chaque accélérateur. Cette approche présente plusieurs avantages. Notamment, elle permet d'abstraire la gestion du matériel après une phase d'initialisation et de détection des accélérateurs existants. Cette distinction entre une phase de déploiement de plateformes d'accélération et l'étape de calcul en elle-même rend leur utilisation plus transparente, tant en termes de gestion de matériel (connexion et déconnexion d'accélérateur, choix d'une cible disponible lors d'un calcul) qu'en termes de communications entre les portions logicielles et les portions accélérées d'une implémentation.

Une des difficultés majeures de la transition d'une architecture purement logicielle vers une architecture hétérogène est en effet la mise en place de liens de communications entre les différents éléments de la nouvelle architecture. Cette étape est prise en charge par la couche d'abstraction que nous présentons ici. Pour illustrer la méthodologie, nous présentons une utilisation de cette couche d'abstraction reposant sur OpenCL. Les fonctionnalités décrites dans le standard réduisent le travail que nous avons à réaliser pour cette étape. Pour bénéficier au mieux des avantages de la librairie QuantLib, nous avons choisi une approche d'intégration à la librairie à un grain relativement fin, malgré l'impact sur les performances. En effet, plus le grain est fin, plus la communication entre éléments physiques est explicite et plus il est difficile de saturer les accélérateurs. Une telle saturation conduit en effet à une utilisation optimale d'un accélérateur, en bénéficiant d'optimisations usuelles, comme le remplissage de pipeline avec des données tirées de plusieurs options calculées en parallèle, des accès mémoires concurrents plus efficaces (résolution de conflits mémoires plus aisée et accès groupés). Cependant, cette approche *bottom-up* de construction d'un *framework* d'intégration d'accélérateur permet d'aboutir à une solution relativement tôt dans le cycle de développement, pour ensuite itérer et affiner cette solution sur une base fonctionnelle. La flexibilité offerte par la finesse du grain préserve également les fonctionnalités haut-niveau de QuantLib. Les accélérateurs issus de cette approche sont de plus aisément réutilisables et limite le redéveloppement de fonctions.

2.2.2 Choix d'implémentation

L'architecture développée répond à une série de critères qui sont nécessaires pour obtenir certaines fonctionnalités requises. Tout d'abord, l'utilisation de la couche d'abstraction proposée (API) ne doit pas demander d'efforts importants de la part des utilisateurs pour la prendre en main et l'instancier. Autrement, cette démarche irait à l'encontre même d'une démarche de simplification de l'intégration d'accélérateurs matériels. Cette couche d'abstraction doit donc gérer les accélérateurs matériels de manière efficace, voire, dans le meilleur cas de figure, de manière transparente. L'implémentation que nous décrivons ci-après 2.2.3 repose sur l'utilisation d'OpenCL, qui aide à supporter les couches basses de l'architecture sans action des utilisateurs (détection d'accélérateur disponible, gestion des liens de communications). OpenCL permet aussi de cibler plusieurs types d'accélérateurs matériels avec un effort en temps de développement minimal.

De plus, une telle couche d'abstraction se doit d'être extensible aisément, tant pour l'ajout de nouvelles fonctionnalités que pour l'intégration d'accélérateurs matériels non supportés par OpenCL (par exemple, intégration du flot NVidia CUDA et des GPU associés). Ces contraintes doivent bien sûr être respectées sans surcoût conséquent en termes de temps d'exécution, ainsi qu'en termes de temps de développement pour un utilisateur. Deux utilisations distinctes de cette API sont à distinguer : le développement et l'intégration de noyaux matériels d'une part et leur utilisation dans une application financière d'autre part.

2.2.3 Implémentation

Nous avons choisi de développer l'API en C++. QuantLib, bien que comportant des interfaces de programmation dans plusieurs langages, est une librairie C++. L'intégration de notre API en est donc facilitée et n'est pas pénalisée par l'impact négatif usuel de l'échange de données entre des applications écrites dans deux langages différents. Enfin, bien qu'étant majoritairement développé en C, OpenCL propose lui aussi des interfaces de programmation C++. De plus, cet API repose sur la librairie C++ *Boost*, notamment pour l'utilisation de pointeurs intelligents (*smart pointers*), qui facilitent la gestion de la mémoire (durée de vie et partage de pointeurs).

Comme détaillé dans le diagramme UML 2.2, deux classes principales forment la structure de l'API, à savoir les classes héritant de PlatformManager, HWProcessor et spécifiques au *back-end* OpenCL. Celles-ci définissent un gestionnaire générique de plateforme matérielle (un accélérateur matériel et une manière d'accéder à ses ressources) et un cœur de calcul à instancier sur cette plateforme.

Un objet de la classe PlatformManager est instancié de manière statique pour gérer tous les accélérateurs présents sur le sytème. Les cœurs de calculs ont ainsi accès au même PlatformManager, ce qui lui permet de gérer tout conflit d'accès et d'instanciation sur ressources matérielles. Les classes PlatformManager et HWProcessor doivent être héritées par des classes filles implémentant les différentes méthodes virtuelles déclarées. Pour éviter le surcoût usuel dû à l'appel de méthodes déclarées dans une classe parente, le *pattern* dit CRTP est employé pour l'implémentation de classes filles de HWProcessor.

```
1  template <class Impl>
2  class HWProcessor : public QuantLib::CuriouslyRecurringTemplate<Impl>
3  {
4      /* (...) */
5      void execute();
6  };
7
8  template <class Impl>
9  void HWProcessor<Impl>::execute(){
10         this->impl().execute();
11 }
12
13 class CLProcessor : public HWProcessor<CLProcessor>
14 {
15     //Effective implementation of the method
16     void execute(const unsigned int globalRange, const unsigned int localRange);
17 };
18 %\label{code:crtp}
```

Listing 2.1 – Exemple d'implémentation du *pattern* CRTP

Comme illustré dans la portion de code 2.1, cette méthode permet de résoudre au moment de la compilation les conflits d'appel de méthodes entre la classe fille et la classe mère, pour des temps

d'exécution réduits.

La classe HWProcessor comporte quatre méthodes principales de gestion d'un cœur de calcul : send(), retrieve (), run() et execute(). Ces méthodes doivent bien sûr être implémentées par une classe fille spécialisée pour la gestion d'un type de plateformes. Les méthodes send() et retrieve () se chargent du transfert de données entre cœurs de calculs et programmes hôtes, tandis que la méthode execute() a pour but de lancer l'exécution du noyau. run() a pour objectif de regrouper les trois opérations précédentes pour les lancer consécutivement (send(), execute() puis retrieve ()). Cette méthode fournit un accès unique pour l'exécution d'un cœur de calcul et peut être utilisée pour automatiser quelques fonctionnalités intéressantes, comme par exemple l'omission de l'étape de calculs si les données d'entrées n'ont pas été rafraichies. Les objets de la classe HWProcessor présentent un indice de priorité, pour fournir au gestionnaire de plateforme un moyen d'ordonnancer les calculs en fonction de l'importance de chaque cœur implémenté sur plateforme. En l'état, toute plateforme ajoutée au système est gérée de manière identique, sans considération de performance.

Les deux classes génériques de l'API sont spécialisées en CLPlatformManager et CLProcessor, reposant sur l'API C++ d'OpenCL pour les lier au matériel. En plus des méthodes de la classe mère, CLProcessor inclut des éléments supplémentaires pour une gestion des noyaux de calcul OpenCL et des zones mémoires globales. La création d'éléments mémoires permet d'envoyer des données à un CLProcessor et de recevoir des données en provenance de celui-ci, via les méthodes adéquates. L'utilisation de *templates* lors de leur création permet de s'interfacer aisément avec les types de données spécifiques à QuantLib (Real, etc.). En plus d'une fonction de chargement sur matériel (loadProcessor()), CLProcessor inclut une méthode de déclaration de noyau (declareProcessor (...)) qui facilite la gestion de code spécifique à un programme OpenCL.

CLPlatformManager masque une majeure partie du code spécifique à OpenCL, comme la création de contexte et la gestion des files d'exécution des noyaux.

Ces deux classes fournissent aussi une gestion basique d'erreur, avec un retour de message d'erreurs pour la correction de bugs, via la levée d'exception lors de comportements non usuels.

2.3 Application à QuantLib et cas d'étude

Nous venons de présenter une couche d'abstraction qui permet de dissocier la gestion d'accélérateurs matériels du développement d'applications. Cette API est indépendante de la librairie logicielle qui l'utilise et peut donc être adaptée pour d'autres applications que l'accélération de QuantLib. Le but est maintenant de l'utiliser dans le cadre d'applications financières, basées sur QuantLib, sans surcoût important, tant en termes de performances d'applications portées sur accélérateur que de temps de développement. Une première approche consiste à utiliser directement l'API pour accéder à un *CLProcessor* sur matériel, depuis le code principal de l'application. Le développeur va donc utiliser QuantLib pour les calculs non accélérés et l'API pour l'accélération, en gérant lui-même les échanges entre ces deux outils. Nous proposons ainsi une approche moins coûteuse pour l'utilisateur et qui limite la répétition et le paramétrage de code via son encapsulation

dans les objets prédéfinis par QuantLib. L'approche consiste à intégrer l'application accélérée au sein de la librairie QuantLib.

2.3.1 Utilisation de l'API dans le flot QuantLib

Nous détaillons ci-après l'approche proposée pour l'utilisation de l'API au sein de l'environnement QuantLib pour la résolution d'un problème financier (tel que défini dans la section 1.2.3).

La première étape reste identique à toute approche de résolution de problème financier, à savoir le choix de la modélisation des instruments financiers, ainsi que les méthodes d'implémentations sélectionnées. Ces choix sont à réaliser en fonction de critères de performance à fixer, en termes de précision, de temps de calcul et de budget énergétique.

Une fois la méthode d'implémentation connue, il est possible d'isoler la portion du calcul que l'utilisateur souhaite porter sur accélérateur. Cette portion isolée de l'application doit présenter une complexité algorithmique suffisante pour justifier de la perte en performance liée à la communication entre hôte et accélérateur. Elle doit donc nécessairement présenter un niveau de parallélisme suffisant pour qu'elle soit adaptée à l'architecture matérielle sur laquelle elle va être portée. De plus, l'extraction de parties d'applications pour leur accélération doit respecter les frontières logiques mises en place par QuantLib. Quel que soit la séparation logique faites par QuantLib entre les différentes classes impliquées et leur droits sur les données manipulées, ce sont ces dernières qui permettent de distinguer où il est judicieux d'effectuer le partitionnement.

Lorsque la portion à accélérer est extraite de la méthode de calcul choisie, le portage effectif de celle-ci sur accélérateur peut être réalisé. En pratique, le développement de noyau d'accélération s'effectue à partir du flot de développement fourni par les constructeurs (que ce soit le flot CUDA pour les GPU NVIDIA, les différentes implémentations d'OpenCL par Altera, AMD, etc.). De manière générale, les degrés de liberté sur la distribution du calcul en parallèle (SIMD, MIMD, etc.) sont dépendants du niveau d'abstraction de la portion de code choisie et de l'interface attendue par le reste de la librairie QuantLib. Dans le cas de l'utilisation de plateformes OpenCL par exemple, la taille des *work-groups* à déployer (correspondant à une exécution SIMD), ainsi que le nombre de *work-groups* voire de noyaux différents à placer en file d'exécution (exécution MIMD possible) sont liés aux paramètres de la méthode implémentée.

Ce noyau d'accélération est ensuite encapsulé dans une classe fille correspondant aux objets QuantLib manipulés. En plus de servir d'interface avec le reste de QuantLib, ces classes filles gèrent l'ordonnancement des calculs et les effets de bord dus à l'utilisation de QuantLib (c'est-à-dire le report des fonctionnalités non gérées sur la classe mère). Cette encapsulation hérite de plus de *CLProcessor* pour chaque noyau développé, pour permettre la gestion des échanges de données entre les couches logicielles et matérielles, ainsi que l'initialisation du noyau et des zones mémoires globales correspondantes lors de la création d'un objet.

Une fois un premier cycle d'exécution et d'évaluation de performances, il est possible de réitérer pour atteindre les critères de performances choisis (si ce n'est pas déjà le cas), en reconsidérant la profondeur de la librairie QuantLib à accélérer, pour permettre une meilleure adaptation de

l'algorithme à la cible matérielle sélectionnée. Une telle approche *bottom-up* d'intégration logicielle sur accélérateur matériel repose donc sur l'intégration successive d'éléments logique plus grands, de méthodes spécifiques de classes jusqu'à une classe QuantLib en elle-même.

En pratique, en reposant sur un flot OpenCL pour cibler des accélérateurs, l'installation chez un utilisateur de la librairie QuantLib étendue pour en accélérer une partie inclut une librairie dynamique correspondant à l'API, une version modifiée de QuantLib, ainsi que les noyaux (sous la forme de code C99 pour compilation juste-à-temps sur GPU et de fichiers binaires pour des cibles FPGA).

2.3.2 Application sur le *framework* QuantLib de calcul d'arbres

Nous présentons ci-après une application du flot de conception présenté dans la section précédente. Nous nous intéressons au *framework* de calcul d'arbre, dans le cas du calcul d'options à l'aide d'un arbre binomial (tel que décrit dans la section 2.1).

Si l'on observe le diagramme séquentiel correspondant à l'évaluation de la valeur d'un sous-jacent par un arbre 2.1, on remarque que la méthode *stepback()* est un candidat intéressant pour le portage sur accélérateur. Cette méthode est en effet aisément parallélisable, car appliquée à l'ensemble des nœuds de l'arbre à un instant donné pour chacun de ses appels. De plus, elle est naturellement isolée du reste des fonctionnalités de la classe qui l'intègre (*TreeLattice1D*, correspondant à un arbre implémenté sous la forme d'un treillis), ce qui simplifie son extraction du flot logiciel. C'est donc un bon candidat pour une première itération de portage sur accélérateur. En termes d'implémentation, le noyau conçu réalise une opération élémentaire sur un unique nœud de l'arbre et est placé en file d'exécution autant de fois qu'il reste de nœuds à calculer au pas temporel considéré.

L'intégration à la librairie QuantLib passe ensuite par la création d'une classe héritant de *Lattice1D*. La classe fille *CLLattice1D* inclut ainsi un *CLProcessor*, gérant le noyau remplaçant la fonctionnalité de *stepback()*. Seulement deux méthodes sont à modifier : le constructeur pour initier ce *CLProcessor* et la méthode *stepback()* en elle-même, pour lancer l'exécution du noyau sur accélérateur. Pour des considérations d'interfaçage et d'utilisation directe des données en provenance de *CLLattice1D*, nous avons choisi d'étendre également la classe *CLProcessor* afin de surcharger le constructeur.

Pour un utilisateur final, l'utilisation de *CLLattice1D* est identique à une utilisation usuelle de QuantLib sans besoin d'action particulière de sa part, après installation de la librairie étendue.

Cette implémentation présente des temps de calculs peu intéressants, de par les échanges de données importants entre hôte et accélérateur entre deux itération de *stepback()*. Une optimisation possible de l'ensemble passe par la modification de *DiscretizedAsset*, pour permettre à la matrice stockant les résultats temporaires d'être entièrement présente sur accélérateur, sans copie nécessaire sur le GPP hôte. Dans le cadre du calcul d'options et en fonction du type d'option considéré, il faut donc créer une classe fille de *DiscretizedVanillaOption*. Dans un premier temps, il est possible d'implémenter un noyau ne gérant que les options américaines, la charge des autres options étant reléguée à la classe mère. Un tel noyau a la charge d'implémenter les méthodes *preAdjustValuesImpl()*

et *postAdjustValuesImpl()*, pour permettre la réalisation d'une opération de *rollback()* complète sur accélérateur. De même que pour *CLLattice1D*, *CLDiscretizedVanillaOption* ne modifie que le constructeur et les méthodes accélérées, le reste du code restant identique à la classe mère.

2.4 Conclusion

L'intégration d'accélérateurs à une librairie de calcul financier déjà existante présente des avantages certains : les spécificités du domaine considéré sont déjà présentes dans la librairie hôte, ce qui résout à première vue une grande partie des besoins d'un développeur financier. Nous avons présenté ici une couche logicielle permettant d'intégrer des noyaux matériels dans une librairie logicielle, de manière transparente pour un utilisateur de la librairie - une fois la plateforme matérielle installée sur la machine hôte. OpenCL est employé en tant que couche bas-niveau pour gérer les cibles matérielles et leur comportement durant l'exécution.

Nous avons toutefois montré que cette démarche nécessite de réaliser des compromis coûteux. L'architecture de QuantLib, pensée pour être une librairie purement logicielle, repose sur des communications fréquentes de données entre différentes classes, ce qui implique donc de n'accélérer que des opérations élémentaires sur matériel (sous peine de temps de communications coûteux entre hôte et accélérateur). Déporter de plus gros grains d'un modèle sur accélérateur demande de porter également le comportement de chaque classe de QuantLib impliquée, au prix des gains espérés en temps de calcul.

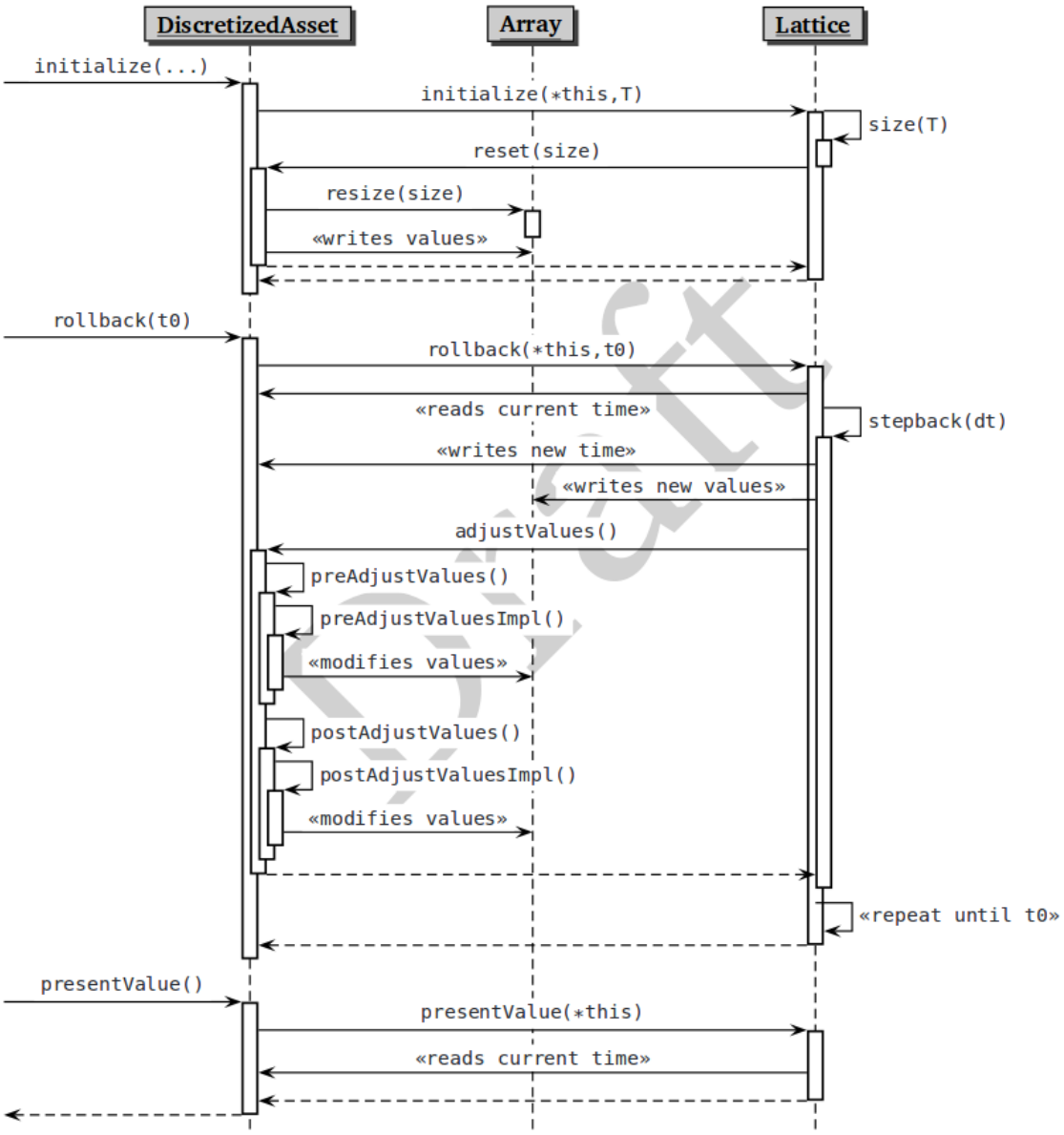

FIGURE 2.1 – Diagramme séquentiel : treillis pour l'évaluation d'un sous-jacent à l'aide d'arbres (binomiaux ou trinomiaux) [Bal]

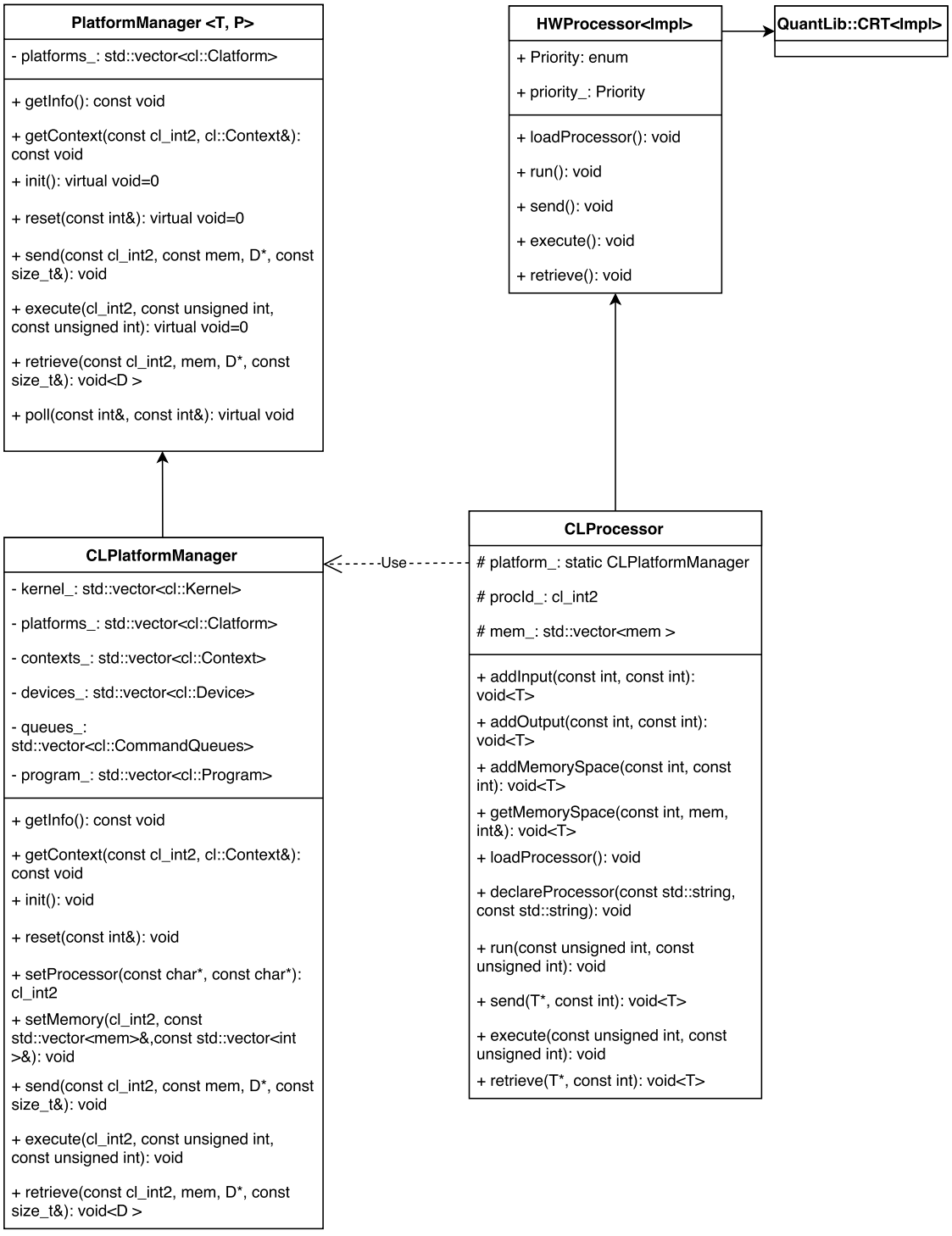

FIGURE 2.2 – Diagramme UML adapté de l'API

Chapitre 3

Approche spécialisée d'implémentation d'un modèle financier basé sur le flot OpenCL pour FPGA

UNE première approche d'intégration d'un accélérateur matériel dans un flot logiciel standard a montré l'impact des temps de communication entre hôte et plateforme sur les performances. Cet impact implique que pour obtenir une accélération globale, le gain en temps doit être supérieur au surcoût de la communication. Ceci n'est possible que dans le cas de l'accélération d'une portion conséquente d'un algorithme à accélérer.

Dans ce chapitre, nous reprenons l'exemple de calcul d'option américaine détaillé en 2. Dans un cas d'utilisation réel, un évaluateur d'options américaines est calibré avec les données du marché. Notamment, cette calibration passe par le calcul de la volatilité implicite d'options américaines avec différents jeux de paramètres d'entrées. Calculer la volatilité implicite est un processus itératif, qui implique d'évaluer des options avec certaines valeurs, puis d'affiner ces valeurs jusqu'à atteindre un niveau de précision acceptable sur la valeur de volatilité trouvé. Cette forme d'algorithme est souvent complexe à accélérer, en particulier à cause des dépendances entre les itérations.

Nous commençons par présenter en détail l'algorithme utilisé pour calculer les options américaines et la volatilité implicite. Nous réalisons une étude architecturale complète, en nous appuyant sur le flot de programmation OpenCL. Les performances obtenues sur plusieurs accélérateurs pour deux des architectures retenues sont ensuite présentées et évaluées, en termes de vitesse de calcul et de consommation énergétique. Ces résultats sont enfin utilisés pour proposer une architecture efficace pour le calcul de volatilité implicite.

3.1 Volatilité implicite et méthode de calcul d'options américaines

3.1.1 Modélisation d'une option américaine

Les options dites américaines sont des options relativement communes : la plupart des options échangées sur le marché sont américaines [Hul14]. La particularité d'une option américaine est qu'elle peut être exercée à tout instant, jusqu'à sa date d'échéance. La valeur finale d'une option américaine dépend donc de la valeur que prend le sous-jacent durant l'ensemble de la vie de l'option. Cette dépendance temporelle rend l'évaluation de ce type d'option non triviale : contrairement aux options européennes, elles ne peuvent pas être calculées analytiquement. Des méthodes existent toutefois pour obtenir une approximation de leur valeur.

Il n'existe pas de consensus sur le modèle optimal à utiliser pour évaluer les options américaines. Des travaux précédents [JLT11a] ont cependant estimé que les méthodes basées sur le calcul intégral forment le meilleur compromis en termes de précision et de temps de convergence. Ces mêmes travaux ont montré que l'utilisation d'arbres binomiaux fournissaient le meilleur compromis dans le cas où le temps de calcul était clé. L'utilisation d'accélérateurs pourrait réduire l'impact de ce défaut et rendre ces solutions plus attrayantes.

Le modèle CRR (Cox-Ross-Rubinstein, [CRR79]) est une version discrétisée du modèle de Black et Scholes. Il repose sur la construction d'un arbre binomial, qui permet de représenter la discrétisation temporelle. Il s'évalue donc par la méthode des arbres binomiaux présentée au chapitre 1. Cet arbre binomial est illustré figure 3.1. Ce modèle a été sélectionné pour ces travaux.

Pour simuler l'évolution du sous-jacent au cours du temps, le prix du sous-jacent peut soit augmenter soit diminuer d'un montant fixé à chaque étape temporelle, en suivant respectivement une probabilité p et $q = 1 - p$. Cet arbre est recombinant : si la valeur du sous-jacent augmente après un pas temporel pour diminuer au pas suivant, elle garde la même valeur. Une conséquence directe du caractère recombinant de cet arbre est que chaque pas temporel s'accompagne de la création d'un unique nœud supplémentaire. Soit Δt le pas temporel, à $t = n\Delta t$, il y a donc $n + 1$ valeurs possibles pour le sous-jacent. A la date finale T d'expiration de l'option, le sous-jacent a une valeur parmi N possibles (avec $T = N\Delta t$). A un instant $t < T$, les valeurs que le sous-jacent peut prendre sont notées $S_{t,k}$ avec $k \in 1, \ldots, n + 1$. La valeur $V_{t,k}$ de l'option associée au sous-jacent $S_{t,k}$ est calculée comme étant le maximum entre le prix d'exercice à t et sa valeur potentielle avant l'expiration de l'option. Pour évaluer les valeurs du sous-jacent et de l'option à chaque nœud (i.e. pour chaque coordonnée (t, k) dans l'arbre), l'arbre est calculé à l'envers, en commençant par les feuilles et finissant par calculer la valeur de l'option $S_{0,0}$ à $t = 0$.

$S_{0,0}$ représente la valeur maximale de l'option dans la période de temps considérée. Les valeurs aux feuilles ($S_{T,k}$, pour $k \in 0, ..., N - 1$) correspondent aux prix d'options européennes et peuvent donc être calculées analytiquement.

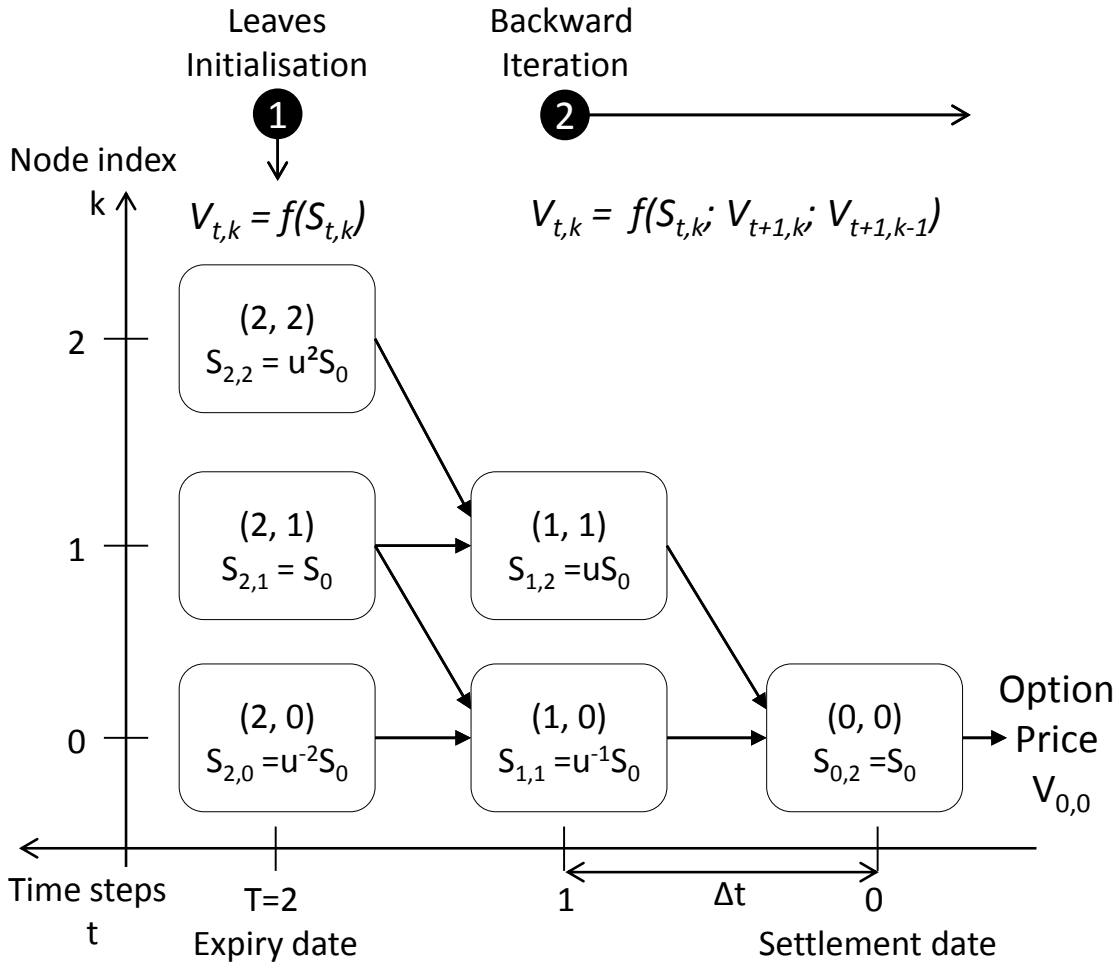

FIGURE 3.1 – Arbre binomial appliqué à l'évaluation d'une option américaine

Chaque nœud est évalué en utilisant les formules de récurrence suivantes (pour un *call*) [Hul14] :

$$\begin{aligned} S_{t,k} &= dS_{t+1,k} \\ V_{t,k} &= max(S_{t,k} - K; rpV_{t+1,k} + rqV_{t+1,k-1}) \end{aligned} \tag{3.1}$$

Avec d, K, r, p, q sont des paramètres de l'option, où :

- $d = e^{(-\sigma\Delta t)}$, avec σ la volatilité de l'option,
- K le prix d'exercice, ou *strike* (le prix d'achat de l'option),
- r est le taux sans risque.

3.1.2 Volatilité implicite

La volatilité d'un sous-jacent est définie comme "l'incertitude sur la rentabilité d'un titre" [Hul14]. Ce paramètre est nécessaire pour modéliser l'évolution d'un sous-jacent, mais n'est toutefois pas directement observable. Une étape de calibration des modèles utilisés pour évaluer des options

correspond donc à l'estimation de la volatilité du sous-jacent. La calibration des modèles d'évaluation de produits financiers est une étape critique pour la précision de ceux-ci. Cette calibration compense une hypothèse approximative réalisée lors de l'utilisation du modèle de Black et Scholes. En effet, ce modèle suppose que les distributions de probabilité de la valeur d'un actif sont log-normales. En pratique, on obtient plutôt des distributions à queues 'plates', plus étalées sur les valeurs peu probables. L'évaluation des courbes de volatilité revient à corriger cette hypothèse à partir des prix effectifs du marché. Les traders utilisent en pratique des surfaces de volatilité implicite, rajoutant comme dimension supplémentaire la durée de vie de l'option, dont la volatilité dépend aussi. Dans la suite du document, le terme implicite pourra être omis par souci de concision. L'évaluation des surfaces de volatilité pour un portefeuille d'options peut donc entraîner une charge de calcul conséquente, qui doit être répétée relativement fréquemment pour réduire les risques sur les prises de position sur ce portefeuille.

Une manière aisée de réaliser cette calibration est d'utiliser les valeurs historiques du prix du sous-jacent. Cette méthode a toutefois des limites en termes de précision et le calcul de volatilité implicite lui est préféré. La volatilité implicite est calculée à partir des prix d'exercice d'options échangées sur le marché. Dans le cas d'options américaines, le modèle de calcul d'options n'est pas réversible et une méthode itérative de recherche de racines est couramment utilisée. Plus précisément, la volatilité implicite σ est la valeur de volatilité pour laquelle $V_{marché} - V(\sigma) = 0$, où $V_{marché}$ est le prix d'exercice d'une option effectivement échangée en bourse et $V_(\sigma)$ est la valeur retournée par le modèle d'évaluation d'option, ayant pour paramètre d'entrée σ.

L'implémentation retenue repose sur l'algorithme de Newton-Raphson pour obtenir une approximation de la volatilité implicite. A partir d'une supposition viable de la volatilité, notée σ_0, on calcule un prix d'option V_n, qui est ensuite comparé au prix du marché. L'approximation sur la volatilité est ensuite mise à jour, selon l'équation (3.2).

$$\sigma_n = \sigma_{n-1} + V_{market} - V_n/\nu \tag{3.2}$$

Avec ν représentant la grecque Vega ($\nu = \delta V/\delta\sigma$). ν est elle aussi mise à jour de manière itérative :

$$\nu = \frac{V(\sigma_n + \delta\sigma/2) - V(\sigma_n - \delta\sigma/2)}{\delta\sigma} \tag{3.3}$$

On peut évaluer grossièrement la charge de calcul liée au tracé d'une surface de volatilité. Pour une surface de volatilité, on cherche généralement à obtenir environ 2000 points, à partir desquels le reste de la surface est interpolé [DHS07]. Chaque point nécessite au maximum 100 itérations pour le niveau de précision souhaité. Comme ν doit être calculé à chaque itération, celle-ci passe par l'évaluation de 3 options. Cela nous donne une borne supérieure de 6.10^5 calculs d'option par surface de volatilité. Comme on le verra dans les sections suivantes, le calcul d'une option est déjà coûteux, obtenir une courbe de volatilité n'est donc pas trivial. Pour un cas réaliste, il faut calculer ces surfaces pour différentes durées de vie d'options qui vont de l'ordre de quelques mois à quelques années, pour pouvoir prendre position sur les marchés. La complexité de ce calcul en fait un cas

d'étude intéressant et utile pour l'utilisation d'accélérateurs dans le domaine financier.

La majorité des publications se concentrent sur l'accélération de l'évaluation d'options, avec pour performance clé le facteur d'accélération obtenu (en comparaison avec une référence logicielle). Bien que l'application financière est éloignée de celles illustrées dans ce manuscrit, les travaux de Nasar et al. [NU12] méritent d'être mentionnés : ils implémentent explicitement une solution de calcul de volatilité implicite, pour des options dites *Constant Maturity Swap spread (CMS spread)*. A notre connaissance, il n'y a pas d'autres travaux s'intéressant spécifiquement à l'accélération du calcul de volatilité, et ce malgré l'intérêt montré par les partenaires rencontrés durant ces travaux. Il est vraisemblable que les solutions utilisées soient conservées en interne, sans publication.

3.2 État de l'art

3.2.1 Choix de méthode d'implémentation et critères de performance

Nous venons de présenter le modèle de Cox-Ross-Rubinstein (aussi nommé modèle binomial), ainsi que son utilisation pour le calcul de volatilité implicite d'une option américaine. L'implémentation de ce modèle peut être effectuée suivant plusieurs méthodologies. Le choix de la méthodologie dépend des contraintes de l'application et doit pouvoir représenter à la fois les spécificités du domaine financier et permettre de traduire le problème en une implémentation réelle sur matériel (que ce soit un GPP ou une architecture hétérogène). Nous avons choisi de suivre le formalisme défini par de Schryver et al. dans leur travaux [dSSK+11]. Comme décrit dans le chapitre 1, ils distinguent trois niveaux dans un système de calcul financier : le problème à résoudre, sa modélisation ainsi que la solution de calcul, à savoir une méthode d'implémentation et l'architecture matérielle qui la supporte. Ils se servent de cette séparation pour créer un environnement de test automatique. Les critères de performance étudiés sont plus exhaustifs que ceux usuellement retrouvés dans la littérature. Ils incluent notamment le nombre d'éléments traités par seconde, l'énergie consommée par élément traité et la précision de ces éléments (en RMSE). L'unification de l'environnement de test permet de comparer entre eux des travaux qui resteraient indépendants sans cet effort. Dans notre cas, le problème que nous considérons est l'estimation d'une valeur de volatilité qui rend le modèle d'évaluation d'option fidèle à la réalité. La modélisation mathématique que nous employons est donc le calcul de volatilité implicite, qui repose sur une recherche itérative de racine (méthode de Newton-Raphson) et sur le modèle binomial.

La méthode d'implémentation choisie doit être adaptée à l'architecture matérielle, ainsi qu'au modèle sélectionné. Une étude comparative des méthodes adéquates pour le portage de modèles financiers sur FPGA a été réalisée par Jin *et al.* [JLT11a]. De manière générale, selon leurs critères, la méthode offrant le meilleur compromis se base sur du calcul intégral. Ils ajoutent que les méthodes à base d'arbres (binomial comme trinomial) sont plus intéressantes dans le cas d'une forte contrainte temporelle, alors que du point de vue de la précision, la méthode de différence finie explicite est à préférer.

Il existe dans la littérature différentes solutions d'implémentation qui s'approche du problème étudié. Certaines de ces approches sont détaillées ci-après pour comparaison avec notre approche.

3.2.2 Méthode de Monte Carlo

La méthode de Monte Carlo est particulièrement adaptée à la résolution de problème comportant plusieurs dimensions. Dans le cas de l'évaluation d'options, elle montre tout son intérêt pour des options sur plusieurs produits sous-jacents corrélés entre eux. De même, les modèles à volatilité stochastique (tels que le modèle d'Heston) sont eux aussi de bons candidats pour implémentation selon la méthode de Monte Carlo. Le fait que la méthode de Monte Carlo soit essentiellement une simulation temporelle de l'évolution d'un sous-jacent est aussi bénéfique lors de l'implémentation de produits exotiques, comme par exemple des options comportant des barrières.

Cette méthode se découpe en trois étapes :

— Une étape de génération de chemins aléatoires forme le cœur de la simulation du sous-jacent et de l'option considérée. Cette étape est répétée un grand nombre de fois (sur plusieurs chemins indépendants).

— Un générateur de nombres aléatoires fournit une entrée de la simulation. Ces nombres aléatoires suivent généralement une loi normale.

— Une étape de réduction finalise le calcul en moyennant les résultats des différents chemins.

On remarque immédiatement l'adéquation de cette méthode avec une architecture massivement parallèle. Le degré de parallélisme de la méthode de Monte Carlo se retrouve à plusieurs échelles : au niveau des chemins calculés, au calcul d'un pas temporel et au niveau de la méthode de simulation elle-même. Ce parallélisme en est même qualifié d'embarrassant comparativement aux autres méthodes d'implémentation.

Le portage des méthodes de Monte Carlo sur accélérateur a donc naturellement été très étudié et ce malgré les limites de cette méthode. Comme mentionné au chapitre 1, son taux de convergence est relativement faible et doit être compensé par une adaptation des paramètres (nombre de chemins simulés et pas temporel) qui implique une plus grande charge de travail en comparaison d'autres méthodes.

Une approche tirant parti de la puissance d'un supercalculateur hétérogène a été proposée par Sridharan *et al.* [SCH+12]. Elle s'appuie sur l'architecture du NOVO-G [GLS11]. Ce calculateur intégrait au moment de la publication 400 FPGA, associés à des quad-cœurs Xeon E5620 et des GPU NVidia GTX-480. Les FPGA utilisés pour cette implémentation sont des Stratix IV E530 (Altera). Ces travaux ciblent exclusivement les FPGA comme accélérateurs. L'architecture de l'implémentation proposée repose sur deux éléments principaux, répliqués autant que possible sur chaque FPGA utilisé. Ceux-ci sont un cœur de calcul de Monte Carlo, ainsi qu'un ordonnanceur associé. Cet ordonnanceur a pour rôle de gérer les accès à la RAM du FPGA où sont stockés les paramètres des options et les critères d'arrêt d'évaluation d'options lorsque celles-ci comportent des barrières. Les cœurs de calcul sont en effet paramétrables et peuvent supporter différentes configurations d'option et de modélisation du produit sous-jacent. Pour étudier les performances de

cette implémentation, elle a notamment été portée sur 16 FPGA et comparée à une implémentation sur 16 *threads*, ce qui rend plus équitable cette comparaison. Cette implémentation atteint des facteurs d'accélération de 40 à 250, en fonction des paramètres des options considérées (pour des options sur sous-jacents corrélés).

Une exploration poussée d'architecture sur FPGA a été décrite par Marxen *et al.* [MKK+11]. Leur objectif est de parvenir à une adéquation optimale entre la modélisation du problème qu'ils considèrent (évaluation d'options à barrière sur des sous-jacent suivant le modèle d'Heston) et la réalisation matérielle (méthode de Monte Carlo optimisée pour la cible matérielle considérée). Leurs principaux critères de sélection sont le temps de simulation (à précision fixée à 0,001 pour la valeur finale de l'option) et l'énergie consommée par l'architecture. Pour y parvenir, ils testent diverses optimisations et sélectionne le meilleur compromis. Le résultat de cette étude est présenté dans un autre article [dSSK+11], dans lequel leur solution sur FPGA est comparée à une implémentation sur GPU. Leur architecture est majoritairement réalisée sur FPGA. Toutefois, l'étape finale de réduction est déportée sur le GPP hôte. Les données sont représentées dans un format flottant simple précision (les opérateurs étant générés à l'aide des IP Core de Xilinx). Les chemins de données du cœur de calcul ne sont pas conçus directement, mais sont générés à l'aide de VisualHDL [vis], ce qui fournit une approche de plus haut-niveau pour cette partie critique de l'implémentation. Cette architecture a été déployée sur une machine portable (GPP de faible puissance, Intel Core 2 Duo T7250@2.0 GHz et 2 GB de RAM associée) associé à une carte ML507 intégrant un Virtex V. Elle a été comparée à une implémentation sur serveur contenant un GPP et un GPU Tesla. Le résultat en temps est moins intéressant pour l'approche FPGA (2 fois plus long), pour une consommation énergétique réduite de 60%.

Une dernière approche utilisant une machine hétérogène embarquée (FPGA Zynq de Xilinx) est présenté par Varela *et al.* [VBT+15]. Ils présentent un modèle original d'évaluation d'option (méthode dite *Reverse Longstaff-Schwartz*). Ce modèle s'appuie sur du *backtracking* : les chemins sont simulés jusqu'à obtenir les valeurs finales, qui sont stockées pour servir ensuite de point de départ pour l'étape de *backtracking*. La modélisation du sous-jacent suit le modèle de Black et Scholes. L'accent est principalement mis sur la précision et la consommation énergétique. Tout comme pour l'approche précédente, l'étape de réduction, qui n'est pas critique, est réalisée sur le GPP. L'implémentation se compose en pratique de deux architectures différentes, une par sens de calcul. Le passage de l'une à l'autre durant le calcul est faite par reconfiguration dynamique (de l'ordre de la seconde). Les données sont stockées dans la RAM entre ces deux étapes. Les opérateurs utilisés sont flottant en simple précision. L'implémentation est comparée à une version logicielle, exécutée sur un Intel i5-2450M (2,50 GHz). L'architecture proposée semble présenter un meilleur temps de calcul et une consommation énergétique drastiquement réduite. En effet, la version Zynq exécute le calcul en 17 ms, pour une consommation de 47 mJ/option, contre 270 ms et 12,7 J/option pour la version logicielle. Cependant, ces chiffres sont à nuancer par l'utilisation de la reconfiguration qui est exclue des mesures de performance. Même s'il est possible de contrôler ce surcoût (par exemple, en utilisant un FPGA plus gros ou en calculant un très grand nombre

d'options), cette omission ne permet pas de conclure sur l'intérêt pratique de cette méthode.

Les trois solutions décrites s'attachent à l'implémentation de Monte Carlo sur FPGA. Ces solutions présentent d'après leurs auteurs, des résultats en temps de calcul et en consommation, qui en font des candidates certaines pour une utilisation réelle. Cependant, elles souffrent de plusieurs problèmes qui les rendent difficilement adoptables. Tout d'abord, comme pour beaucoup d'implémentations, la comparaison est réalisée par rapport à une référence logicielle de provenance inconnue. Même si les résultats de cette référence sont sans doute corrects, il est difficile de conclure quand à son efficacité. Un autre problème réside dans le fait que ces solutions sont conçues pour un problème très précis et ne sont que difficilement modifiables. Bien que des efforts aient été faits pour atteindre une certaine généricité, en particulier avec utilisation de cœurs de Monte Carlo paramétrables ou d'outils de haut niveau, ils ne suffisent pas pour donner accès à la solution à des développeurs non experts en électronique numérique. Finalement, le modèle de Monte Carlo est tellement peu adapté à une implémentation logicielle sur GPP qu'il est relativement simple d'obtenir un facteur d'accélération sur des FPGA. Il aurait été intéressant de pouvoir comparer ces résultats à d'autres modèles. On risque en effet d'avoir des implémentations FPGA de la méthode de Monte Carlo moins efficaces que les implémentations GPP d'autres méthodes. La comparaison de Jin et al. [JLT11a] va d'ailleurs dans ce sens. Leur utilisation de Monte Carlo est cependant justifiée par la multi-dimensionnalité des options considérées.

3.2.3 Calcul intégral

Les méthodes d'évaluation d'option à base de calcul intégral sont intéressantes de part leur taux de convergence élevé, ce qui permet d'obtenir un résultat relativement précis (pour un temps de calcul fixé). Tse et al. [TTL12] ont présenté une étude comparative de l'utilisation de calcul intégral (dit *Quadrature Methods*) sur FPGA, GPU et GPP. Leur étude s'attache notamment à la consommation énergétique de chaque architecture, en plus du temps de calcul et de la complexité algorithmique de la méthode employée. Leur approche isole des portions de calcul qui sont identiques pour différents types d'options pour gagner en généricité : il leur est possible de calculer plusieurs types d'options à l'aide de leur *framework*. Leur architecture repose sur un accélérateur, que ce soit un GPU ou un FPGA, au sein duquel est implémenté le cœur de calcul intégral. La majeure partie des opérations de contrôle sont laissées à la charge du GPP hôte. L'implémentation sur FPGA est réalisée à haut niveau, à l'aide de l'outil HyperStreams (programmation en Handel-C). Leur étude se concentre sur des options sur sous-jacents corrélés. Ce point est relativement surprenant, étant donné que la méthode de Monte Carlo est généralement la plus à même de traiter ces cas d'utilisation, le taux de convergence des méthodes de calcul à base de calcul intégral chutant exponentiellement avec la dimension (convergence en $O(N^{2d})$, avec N le nombre de pas de discrétisation et d le facteur de dimension).

En calcul flottant simple précision, ils obtiennent les facteurs d'accélération respectif de x4,59, x1,75 et x8,37 sur cibles Xilinx Virtex 4 xc4vlx160, NVidia GeForce 8600GT, NVidia Tesla C1060 par rapport à une cible logicielle sur GPP (Xeon double cœur W3504). Une implémentation en

virgule fixe sur FPGA n'est pas envisageable ici, étant donné l'intervalle important de valeurs entre les intégrales partielles calculées et les intégrales de taille complète obtenues en fin d'évaluation d'option. La référence pour ces facteurs d'accélération est une solution multi-threadée sur Intel Xeon W3505. Les performances en double précision sont approximativement deux fois moins bonnes pour chaque implémentation. La cible Virtex 4 surpasse les deux cibles GPU en terme d'efficacité énergétique moyenne, avec des ratios de consommation respectifs diminués de x26, x1, x1,94. Pour des dimensions supérieures, les GPU gagnent en intérêt tandis que les ratios de performance sur FPGA s'amoindrissent. L'utilisation de GPU ne se justifie donc que pour des cibles GPU haut de gamme (Tesla ou équivalent) et dans des conditions où la consommation énergétique n'est pas un facteur limitant.

3.2.4 Méthode des différences finies

Les méthodes des différences finies ne représentent pas un compromis très intéressant pour le calcul financier. Elles convergent plus lentement que les autres méthodes présentées ici et ne sont pas les plus rapides à implémenter. Elles sont toutefois utilisées lorsque le modèle se décrit aisément sous la forme d'équations différentielles partielles.

L'accélération de ces méthodes a été étudiée sur différentes cibles. Une première étude est présentée par Chatziparaskevas *et al.* [CKW12]. Elle s'appuie sur une modélisation de Black et Scholes pour le produit sous-jacent et implémente les méthodes implicite et de Crank-Nicholson. Elle fournit un certain niveau de généricité par l'utilisation d'un petit jeu d'instructions dédié. Des cœurs programmables implémentant ce jeu d'instruction sont connectés sur un anneau. Cette structure permet une communication suffisante pour le calcul du treillis sans surcharger les ressources de communication. L'architecture est déployé sur une cible FPGA Xilinx Virtex 5, et a atteint un facteur d'accélération de 8 par rapport à une référence logicielle.

Une approche moins flexible est présentée par Jin *et al.* [JLT11b] pour la méthode explicite. Un pipeline est généré à partir d'un fichier de configuration représentant les calculs nécessaires pour un type d'option. La génération d'un flot de donnée au format flottant est assurée par FloPoCo [dDP11]. L'architecture a été générée pour des options européenne, asiatique et américaine, avec une précision simple et double. Le facteur d'accélération constaté sur Virtex-6 XC6VLX550T par rapport à une référence logicielle atteint 24.

Les GPU ont également été exploités comme cible d'accélération. Les travaux de Dang *et al.* [DCJ10] présentent une telle approche pour des options américaines reposant sur plusieurs sous-jacents. La méthode de Crank-Nicholson a été sélectionnée et optimisée pour le portage sur GPU. Ils utilisent la tri-dimensionnalité du GPU pour gérer jusqu'à trois sous-jacents. Leur architecture, portée sur des NVidia Tesla S1060/S1070, présente un facteur d'accélération pouvant atteindre x18, dans les meilleures conditions observées (taille maximale du treillis).

Les méthodes des différences finies peuvent être efficacement implémentées sur des architectures parallèles. Des optimisations sont nécessaires pour limiter l'impact des conditions aux limites. Bien que présentant un réel intérêt théorique, ces implémentations n'offrent pas un compromis idéal

dans la pratique, quels que soient les critères d'optimisation recherchés.

3.2.5 Méthodes binomiale et trinomiale

Ce chapitre définit une approche de conception basée sur OpenCL, à travers l'exemple de l'accélération des méthodes basées sur des arbres. Ce ne sont cependant pas les premiers travaux réalisés sur ces méthodes. D'autres approches présentant des résultats prometteurs existent.

Ces approches peuvent être divisées en deux catégories [TT14] :

— une approche scalaire qui utilise un cœur de calcul par option et le réplique pour calculer plusieurs options en parallèle ;

— une approche vectorielle, qui utilise plusieurs cœurs de calcul pour une unique option, ce qui réduit la latence pour le calcul d'une option, au prix d'un ordonnancement plus complexe.

Tavakkoli et al. [TT14] présentent une architecture vectorielle décrite comme systolique. L'architecture est conçue pour calculer un pas temporel de discrétisation en un cycle (*cf.* figure 3.1). Elle se base sur des cœurs de calcul conçus pour évaluer un nœud de l'arbre, qui sont répliqués autant de fois que nécessaire pour calculer le dernier pas temporel. Le calcul est ensuite déroulé jusqu'à atteindre la racine de l'arbre, chaque cœur de calcul adaptant son comportement (passant ou opérant) en fonction de sa position dans l'arbre et de l'étape temporelle. Pour réduire le chemin critique de leur implémentation, les auteurs ont choisi d'évaluer la même étape temporelle de plusieurs options dans chaque cœur de calcul. Cette opération leur permet de placer plusieurs étages de registre selon l'approche dite *C-Slow* [WMPW03]. Ils ont porté leur architecture sur cible Virtex 7, ainsi que sur cible Virtex 4 pour une meilleure comparaison avec l'état de l'art. Deux formats de données sont utilisés : un format flottant simple précision pour le calcul initial de constantes et un format en virgule fixe (16,16) pour le reste du flot de donnée. FloPoCo est utilisé pour générer les opérateurs flottants et de conversion flottant/fixe. Leur implémentation sur Virtex 7 xc7vx980t calcule jusqu'à 30.10^9 nœuds par seconde ($10.7.10^9$ pour la Virtex 4 xc4vsx55). Ils ne mentionnent pas la consommation énergétique de leur architecture.

Des travaux précédents réalisés par la même équipe avaient aboutis à une architecture différente, s'appuyant sur une description à plus haut niveau [JTLC08]. Ils utilisent HyperStreams et le langage haut-niveau Handel-C pour l'implémentation matérielle. L'architecture réalisée est relativement standard : une mémoire est utilisée pour stocker les paramètres et une LUT sert au stockage des prix temporaires de sous-jacent. Ils réalisent une comparaison avec un GPU et des implémentations sur FPGA au format fixe et flottant simple et double précision. Ils atteignent une performance de $1.16.10^9$ nœuds par seconde, sur un FPGA Virtex 4 xc4vsx55.

Une dernière implémentation considérée comme une référence a été proposée par Wynnyk *et al.* [WMI09]. Ces derniers présentent une architecture vectorielle avec un facteur de réplication de 4. L'objectif de cette implémentation est diminuer au maximum la latence pour l'évaluation d'une option. L'implémentation s'appuie sur des *IP Cores* d'Altera pour les opérateurs de calcul. Le format de donnée utilisé est à virgule flottante double précision. Sur un FPGA Altera Stratix III ep3sc260, les auteurs parviennent à atteindre un débit de $0,6.10^9$ nœuds par seconde.

Il n'est pas aisé de se comparer aux deux dernières implémentations décrites. En effet, les travaux présentés sont portés sur des technologies relativement anciennes. Ils sont également précurseurs dans le domaine. Malgré l'intérêt de ces travaux, les débits atteints ne sont pas suffisants pour réellement envisager une utilisation en production. Les travaux de Tavakkoli *et al.* sont plus récents (ils se comparent d'ailleurs à une première version de nos travaux). Même si les résultats présentés semblent impressionnants, ils s'appuient sur une description en HDL une fois de plus très peu accessible par des développeurs non experts dans la conception de circuit. Leur architecture est peu flexible et requiert un FPGA suffisamment grand pour contenir autant de cœurs de calcul que de pas temporels. Ceci a un impact direct sur la précision de l'option évaluée (dans leur cas, au centième près). Cette contrainte de taille du FPGA est également limitante pour un passage à l'échelle de leur solution. Enfin, la consommation énergétique n'a pas été évaluée (même si elle dépend plus de la cible que de l'architecture sur FPGA).

3.3 Implémentation du calcul d'options américaines

Afin d'accélérer le calcul des courbes de volatilité, il est requis d'évaluer efficacement la valeur des options considérées. La première étape nécessaire pour parvenir à ce résultat est bien sûr d'utiliser un cœur d'évaluation d'option qui soit efficace. L'objectif du modèle n'est pas d'être calibré, mais bien d'évaluer des options. L'étape de calibration est bien sûr moins fréquente que l'évaluation directe d'options. Elle ne doit pas être optimisée au détriment du calcul d'options lui-même.

Nous proposons deux architectures d'évaluation d'options reposant sur le flot OpenCL, portées toutes deux sur cible GPU et sur cible FPGA. Les résultats sont ensuite comparés à l'état de l'art.

3.3.1 Une première architecture en flot de données

3.3.1.1 Principe de l'architecture

Contrairement à une méthodologie de développement HDL classique, l'ordonnancement de tâches sur des ressources matérielles est déléguée à l'environnement OpenCL, défini dans le chapitre 1. Le développeur définit dans le programme hôte les *work-items* à exécuter. La charge de travail est ensuite répartie sur les ressources de la plateforme (i.e. les noyaux présents), sans qu'il n'y ait besoin de développer une couche de contrôle autour du calcul qu'il souhaite réaliser. Les tâches à exécuter sont placées dans une ou plusieurs files (*queue* en OpenCL). La plateforme se comporte mieux pour une charge de travail importante, qui permet de masquer les temps de communication entre le processeur hôte et l'accélérateur matériel, en maximisant le remplissage du *pipeline*.

Pour cette raison, la première architecture que nous avons choisi de développer repose sur une représentation en flot de données de l'algorithme, supportée par des noyaux indépendants les uns des autres et formant une application hautement parallélisable. Ces noyaux sont ensuite organisés logiciellement en un *pipeline* qui permet de calculer l'arbre complet.

3.3.1.2 Implémentation

Dans cette architecture, chaque nœud de l'arbre est représenté par un noyau (figure 3.2). Ce noyau est chargé dans la file d'exécution un nombre de fois suffisant pour calculer un arbre complet, soit $N(N+1)/2$ *work-items*, avec N le nombre de pas de discrétisation en temps. Le calcul de l'arbre est organisé en étages, qui correspondent aux colonnes (i.e. à chaque pas temporel). Les étages sont calculés un à un, avec un point de synchronisation lors du passage d'un étage à l'autre. Cette implémentation est adaptée à la structure matérielle de l'accélérateur. Elle s'appuie sur un grand nombre de *work-items*, ce qui maximise l'utilisation des ressources matérielles au cours du calcul.

Ce type d'accélérateur s'intègre dans des applications calculant un volume conséquent d'options. Le fonctionnement par étage est une architecture bien adaptée à la création de *pipeline* : l'option circule d'étage en étage et à chaque itération une option différente est calculée dans chaque étage de l'arbre. Le pipeline complet stocke donc $N+1$ options sur l'accélérateur, avec une option en sortie de l'arbre et N options à chaque étage. Le calcul d'une option est réalisé en exécutant ce réseau de noyaux N fois, jusqu'à ce que le résultat final soit disponible en sortie du *pipeline*. Ces itérations sont contrôlées par le programme hôte, qui ordonnance les transferts de mémoire entre l'hôte et l'accélérateur, ainsi que l'exécution séquentielle des noyaux par groupe de taille N. Pour chaque groupe à calculer, l'hôte exécute quatre instructions :

1. il initialise les données nécessaires pour remplir les premières adresses de la mémoire tampon d'entrée (correspondant à une nouvelle option) ;
2. il écrit ces données dans la mémoire globale de l'accélérateur ;
3. il place en file d'exécution les N noyaux à calculer ;
4. une fois l'exécution achevée, il lit un résultat en mémoire globale.

Pour illustrer le fonctionnement de cette architecture, prenons l'exemple de l'évaluation de 2000 options, avec un nombre de pas de discrétisation de 1024. L'arbre correspondant pour l'évaluation d'une option est composé d'environ 5.10^5 nœuds ($\frac{1024\times1025}{2}$). L'ensemble de l'arbre doit être calculé pour chaque option, ce qui revient à environ 1.10^9 nœuds calculés (et donc autant de noyaux). Le résultat de chaque *work-group* est stocké en mémoire globale. Ces tampons mémoires contiennent l'ensemble des données en sortie de chaque nœud de l'arbre, c'est-à-dire $S_{t,k}$, $V_{t,k}$, ainsi que des indices de calcul. Les données dépendant de l'option, mais constantes lors de son évaluation, sont stockées dans une autre mémoire tampon globale, non illustrée dans la figure 3.2. L'utilisation de mémoire globale est nécessaire pour permettre un stockage des données persistant entre deux groupes de noyaux calculés. Cela permet à chaque noyau d'accéder à n'importe quelle donnée stockée en mémoire. Pour éviter tout conflit d'accès à une donnée, deux mémoires tampons sont utilisées en ping-pong (les données de l'une sont lues tandis que les autres sont écrites). Ces mémoires tampons sont échangées entre chaque *work-group* exécuté pour permettre au flot de donnée de circuler dans le réseau de noyaux. La figure 3.2 illustre ce flot de donnée, appliqué à l'arbre précédemment décrit dans la section 3.1.1.

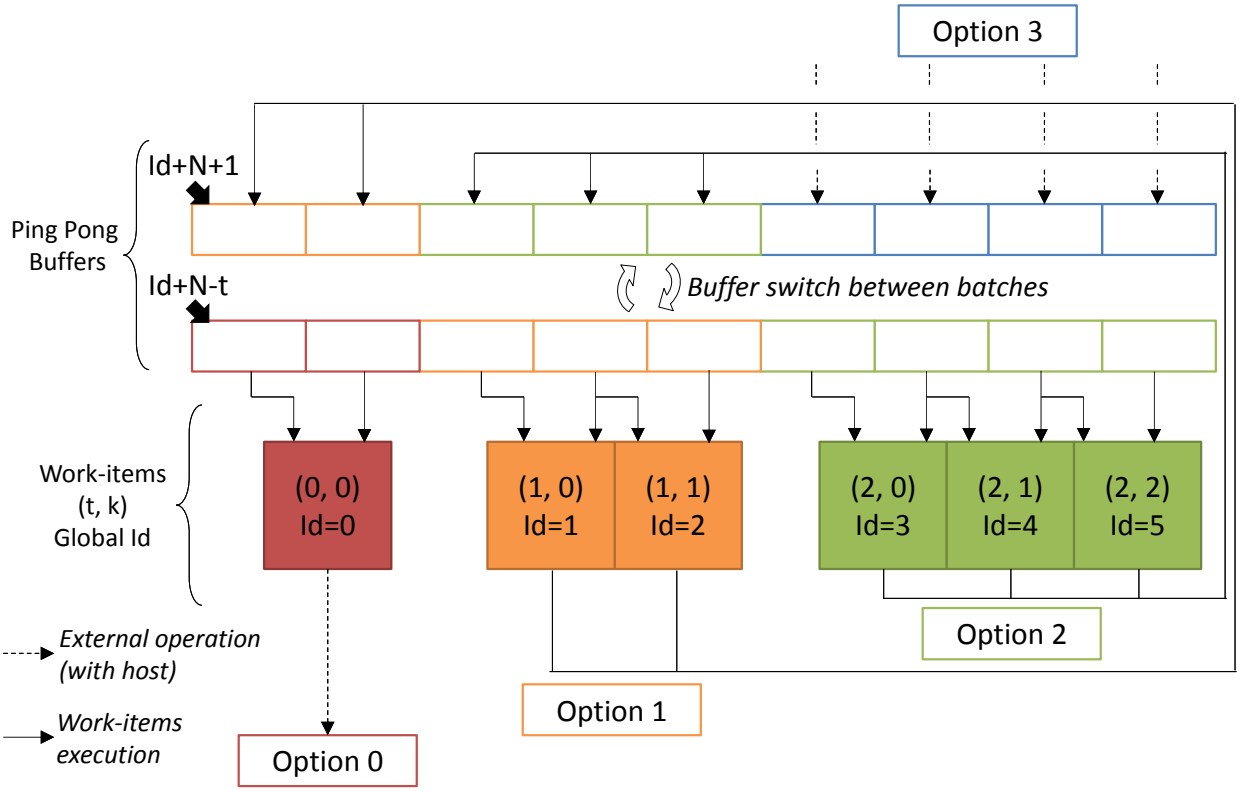

FIGURE 3.2 – Calcul d'un arbre binomial illustré figure 3.1 selon une architecture en flot de données

Pour mieux illustrer l'exécution successive de groupes de noyaux, l'arbre est représenté ici sous forme étendue, avec le dernier nœud positionné à gauche de la figure (à $(t, k) = (0, 0)$) et les feuilles de l'arbre représentées à droite (de $(t, k) = (2, 0)$ à $(t, k) = (2, 2)$). L'option 0, la première à être rentrée dans le *pipeline*, est traitée par le *work-item* $(0, 0)$ et est préparée pour être lue par l'hôte avant que le prochain groupe de noyaux ne soit placé en file d'exécution. Trois autres options sont présentes dans le pipeline ; les options 1 et 2 sont traitées par les *work-items* $(1, 0)$, $(1, 1)$ pour l'option 1 et par les *work-items* $(2, 0), (2, 1)$ et $(2, 2)$ pour l'option 2. L'option 3 est en cours d'écriture dans la mémoire tampon. Deux addresses mémoires sont lues par chaque *work-item* pour calculer la valeur d'un nœud (en accord avec la formule de récurrence (3.1)). L'indice Id indiqué pour chaque *work-item* représente leur indice global respectif. Le premier *work-item* rencontré a pour indice global 0 et correspond à la position $(2, 2)$ dans l'arbre. Les valeurs lues sont stockées en mémoire privée dans le *work-item*, zone mémoire correspondant généralement à des bancs de registres.

Les *work-items* ne peuvent être identifiés de manière fiable que par leur indice global. L'adresse de leurs entrées est alors $(Id+N-t)$. Le calcul des pas temporels par les *work-items* est relativement coûteux en ressources ; ils sont donc stockés dans une mémoire tampon constante, ce qui permet à chaque *work-item* de déterminer leur adresse de lecture au début de chaque groupe $(Id + N - t)$.

En contrepartie, les adresses d'écriture sont moins complexes et ne dépendent que de l'indice global des indices ($Id + N + 1$).

3.3.1.3 Limites de cette architecture

Cette architecture passe le facteur d'échelle et est implémentée aisément. Elle présente toutefois plusieurs limitations.

Premièrement, l'adresse d'une donnée d'entrée d'un *work-item* est dépendante du pas temporel t correspondant à la position du *work-item* dans l'arbre binomial. Toutefois, les *work-items* au sein d'un *work-group* ne correspondent pas nécessairement à des nœuds positionnés sur le même pas temporel. En effet, la taille des *work-groups* est fixée par noyau synthétisé : tout *work-group* placé en file d'exécution a la même taille. Dans le cas illustré précédemment, l'idéal serait d'avoir un *work-group* par étage, pour éviter l'utilisation de mémoire globale. Cependant, les étages sont tous de tailles différentes, ce qui impliquerait de doubler le nombre de *work-items* pour atteindre cet idéal. Cette solution n'est bien sûr pas envisageable.

Un deuxième inconvénient réside dans la taille du groupe de noyaux. L'hôte doit itérer sur l'ensemble des groupes de noyaux pour continuellement remplir le pipeline de l'accélérateur, ce qui introduit un surcoût en termes de temps de calcul. Les opérations d'écriture et lecture de mémoire, ainsi que l'exécution des *work-items*, sont entrelacés et synchronisés par l'hôte, mais elles ont tout de même un coût en temps de calcul. De plus, la structure des *work-items* n'est pas conservée entre les exécutions de groupes de noyaux : les mémoires privées des noyaux sont vidées et les mémoires locales (vues dans le chapitre 1) ne sont pas utilisées. Cette utilisation non optimale des zones mémoires implique de recopier des données précédemment calculées de la mémoire globale vers les *work-items* au début de l'exécution de chaque groupe de noyaux. Comme présenté dans la section 3.5, ceci limite fortement les performances de cette architecture.

3.3.2 Une seconde architecture : optimisation des accès mémoire

Les limites de l'architecture précédente peuvent être résolues pourvu que l'on parvienne à réorganiser les *work-items* en *work-groups* qui sont mieux adaptés à l'évaluation d'un arbre binomial. Plutôt que de se concentrer sur le parallélisme inhérent à une organisation de l'algorithme sous la forme d'un flot de données, un niveau de parallélisation à plus gros grain est plus adapté à l'utilisation de *work-groups* comme élément de base d'une architecture. Nous présentons dans cette section une architecture basée sur un parallélisme de tâches, avec un *work-group* assigné à l'évaluation d'une option (i.e. un arbre binomial complet). Pour réduire le volume de données copié durant une évaluation, à chaque *work-item* est assigné une rangée d'un arbre, où une rangée est définie comme tous les nœuds (t, k) de l'arbre pour un k donné. Un *work-item* est donc responsable du calcul des nœuds (t, k), de $t = T$, jusqu'à $t = N - (k + 1)$. Tout t inférieur à $N - (k + 1)$ correspond à une évaluation d'un hypothétique nœud en dehors de l'arbre et n'est donc pas utile.

Un nombre de *work-items* égal au nombre de pas temporels sont nécessaires au début de

l'évaluation pour initialiser les feuilles de l'arbre. A chaque pas temporel, et donc après chaque itération complète sur k du *work-group*, un *work-item* de moins doit être calculé, jusqu'à ce qu'il n'en reste plus qu'un. Pour une telle répartition du calcul, les paramètres de l'option évaluée, ainsi que la valeur temporaire $S_{t,k}$, peuvent être conservée en mémoire privée, puisqu'on reste dans le même *work-group*. $S_{t,k} = dS_{t-1,k}$ est mis à jour par le *work-item* k en début de chaque itération t. Toutes les données partagées entre *work-items* sont stockées en mémoire locale (i.e. $V_{t,k}$ pour $k \in \{0...N-1\}$). Cette optimisation est possible car tous les *work-items* impliqués dans l'évaluation d'une option sont localisés dans un unique *work-group*. Moins de mémoire locale est disponible sur une plateforme OpenCL que de mémoire globale. La mémoire globale est en effet généralement une mémoire externe (DDR), tandis que la mémoire locale consiste généralement en des blocs RAMs. C'est le cas pour les architectures Altera basées sur FPGA. De ce fait, nous avons choisi de délaisser les mémoires tampons organisées en ping-pong pour une unique mémoire tampon locale. Les conflits mémoire en écriture et en lecture sont résolus à l'aide de points de synchronisation locaux (des *barrières*), ainsi qu'à l'aide de copies temporaires en mémoire privée de certaines données (*cf.* figure 3.3).

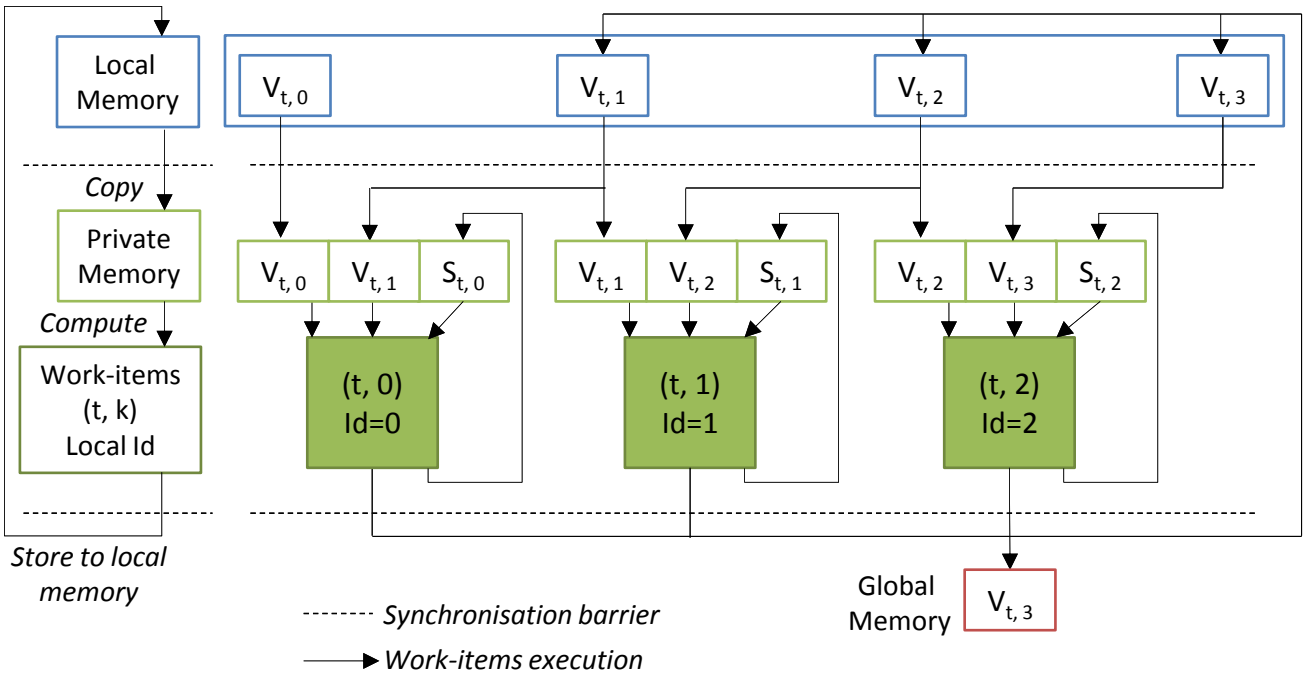

FIGURE 3.3 – Calcul d'un arbre binomial selon une architecture avec optimisation des accès mémoire

3.4 Implémentation du calcul de volatilité implicite

Comme détaillé dans la section 3.1.2, la volatilité implicite est la valeur de volatilité telle que le modèle binomial d'évaluation d'option fournit une valeur d'option égale à la valeur donnée par le

marché (tous les autres paramètres étant connus). Pour la calculer, il faut donc évaluer une valeur d'option de manière itérative, la comparer à une valeur de référence puis adapter les paramètres. On répète ces étapes jusqu'à approcher suffisamment la valeur de référence. Le nombre de répétition est limité en cas de non-convergence.

La méthode de Newton-Raphson, sélectionnée pour calculer la volatilité, est implémentée en tant que deux noyaux disjoints. L'échange de données entre les deux noyaux est assuré à l'aide de *channels* (FIFO) et de la mémoire globale (figure 3.4). Une première approche reposant uniquement sur la mémoire globale a été implémentée, mais offrait des performances très dégradées.

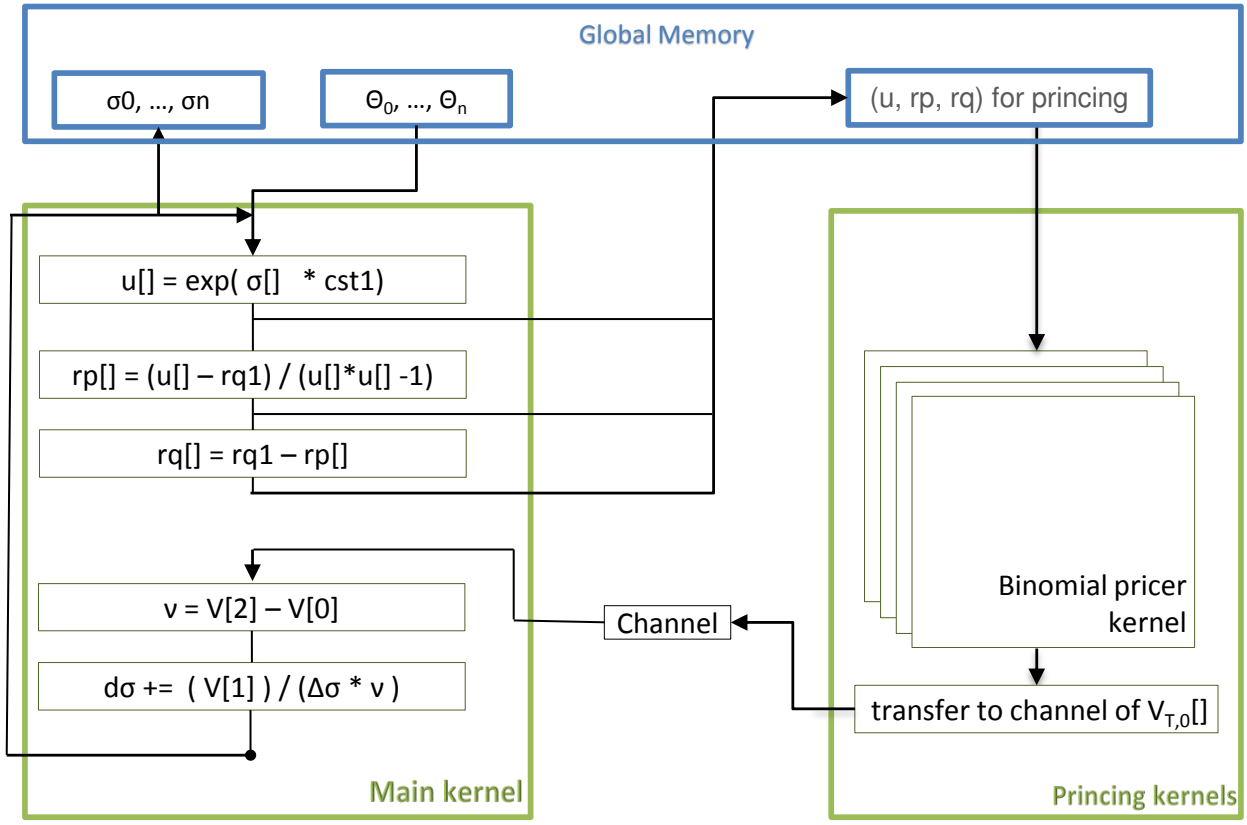

FIGURE 3.4 – Calcul de volatilité implicite sur la base du noyau de calcul d'options américaines

Cette division présente plusieurs avantages. Les *channels* forment un medium de communication entre deux noyaux qui est restreint à la structure interne de la plateforme (i.e. peu coûteux en termes de latence par rapport à l'utilisation de mémoire globale). Il est important de noter que les *channels* sont une construction spécifique à la version d'OpenCL supportée par Altera. Un équivalent existe dans le standard OpenCL (les *pipes*), avec les mêmes fonctionnalités, mais uniquement à partir de la version 2. Le lien entre les deux n'est pas opérationnel dans la version actuelle de l'environnement Altera. De plus, les GPU de NVidia ne supportent pas cette version d'OpenCL. Les travaux présentés sur l'implémentation du calcul de volatilité implicite ne sont donc déployables que sur cible Altera.

A priori, l'utilisation de *channels* pour tout échange entre noyaux paraît séduisante. Elle a cependant des limites. En effet, un rebouclage sur un noyau à l'aide de *channels* mène à des conditions de blocage du calcul. Après un nombre aléatoire d'itérations (généralement moins de trois), l'exécution de la plateforme est stoppée, la disponibilité des données stockées dans les *channels* étant mal signalée. Une architecture naturelle utiliserait les *channels* pour tout échange de données entre les deux noyaux. Cette limite rend cette solution irréalisable. L'utilisation de mémoire globale permet de briser la boucle "*channel* - noyau - *channel* - noyau ". Elle a également comme avantage de faciliter l'initialisation des noyaux à partir de l'hôte, celui-ci ne pouvant pas accéder directement à un *channel* interne. La mémoire globale fournit aussi un accès commun à tout noyau pour certaines données, évitant ainsi des copies consommant des ressources supplémentaires.

Une autre solution consiste bien sûr en l'utilisation d'un unique noyau. Cette division permet toutefois d'utiliser le noyau décrit dans la section 3.3.2 sans le modifier (à l'exception d'une modification des entrées et sorties). De plus, elle permet de paralléliser les noyaux d'évaluation d'option séparément du reste du calcul de volatilité implicite. Le calcul de volatilité implicite et l'évaluation d'options reposent sur une implémentation commune du modèle binomial, ce qui assure que les résultats des deux algorithmes restent en accord. Enfin, toute amélioration du modèle binomial peut aisément être intégrée au calcul de volatilité implicite.

Les deux noyaux (*main kernel* et *pricing kernel* sur la figure 3.4) forment une boucle de rétroaction, du fait de la nature itérative de la méthode de Newton-Raphson. Les trois paramètres (u, rp, rq) dépendant de la volatilité sont mis à jour dans le noyau principal (*main kernel*), puis transférés au noyau d'évaluation d'option en passant par la mémoire globale (*pricing kernel*). Une fois l'option calculée, sa valeur est stockée dans un *channel* jusqu'à sa lecture par le noyau principal. La synchronisation entre les deux parties du calcul est réalisée grâce au *channel*, qui stoppe l'exécution du noyau principal tant que tous les évaluateurs d'option n'ont pas achevé leurs calculs. Le transfert de données du noyau principal vers l'évaluation d'option n'est pas synchronisé. La phase d'initialisation du noyau évaluant les options américaines est en effet bien plus longue que le temps nécessaire pour réaliser une itération du noyau principal. Cette initialisation est effectuée à chaque itération, ce qui assure que l'évaluateur d'options n'est jamais en manque de données d'entrées. L'initialisation correspond à la mise à jour des tableaux de paramètres $u[]$, $rp[]$ et $rq[]$ (tels qu'illustrés sur la figure 3.4), avec la version la plus récente des valeurs de volatilité correspondantes $\sigma[]$.

Ces tableaux de paramètres $u[]$, $rp[]$ et $rq[]$ contiennent trois éléments. En effet, pour une unique itération de la méthode de Newton-Raphson, trois options sont évaluées. Deux de ces jeux de paramètres sont en effet nécessaires pour mettre à jour Vega (*cf.* 3.1.2), tandis que le dernier jeu de paramètres est utilisé pour évaluer l'approximation actuelle de la volatilité implicite.

La majorité de la charge de travail est traitée par le noyau évaluant les options ; le noyau principal se contente de mettre à jour les paramètres dépendant de la dernière approximation de la volatilité et de réaliser la comparaison à la valeur de référence. La différence de charge de travail se reflète dans les ressources utilisées par chaque noyau, ainsi que dans leur ordonnancement. Le

noyau principal est placé en file d'exécution comme un unique *work-item* par l'hôte, tandis que N *work-items* sont appelés pour le noyau d'évaluation d'options. De même, les efforts de réplication matérielle ont été concentrés sur le noyau d'évaluation d'options durant la phase d'optimisation (*cf.* section 3.5).

Le calcul de volatilité implicite est un défi pour une plateforme OpenCL ; d'autant plus sur FPGA. La nature itérative de la méthode de Newton-Raphson, qui s'accompagne d'une forte dépendance de données, rend toute parallélisation de l'algorithme difficile. L'implémentation proposée ici présente un niveau de parallélisme acceptable pour une implémentation sur FPGA, avec trois cœurs d'évaluation d'options concurrents pour un noyau principal, eux-même pouvant contenir un déroulage de boucle pour s'adapter au mieux aux ressources présentes sur le FPGA cible.

3.5 Étude de performance

3.5.1 Méthodologie d'expérimentation

Afin de valider les architectures présentées, nous avons mis en place un environnement de travail, intégré dans une machine de calcul Dell Precision T7400. En tant qu'hôte, nous avons utilisé un processeur Intel Xeon X5450, avec 4 cœurs cadencés à 3 GHz [Int]. Il a accès à 16 Go de RAM DDR2. Cette technologie est relativement ancienne, ce qui est un facteur limitant mineur pour l'exécution de référence sur GPP. Le lien de communication entre cet hôte et les différents accélérateurs est une liaison PCIe gen3 (985 Mo/s par voie et jusqu'à 16 voies par accélérateur). On associe à cet hôte deux cartes pour les accélérateurs. La première est une carte graphique ASUS, basée sur un GPU NVidia GeForce GTX660 TI [NVIa] contenant 5 SMX (Unités de Calcul indépendantes, *cf.* section 1.2.2). Cette carte contient également 2 Go de mémoire GDDR5 (mémoire globale OpenCL). Chaque SMX intègre une mémoire cache de 48 ko (mémoire locale OpenCL). Le lien PCIe utilise les 16 voies accessibles. La deuxième carte est une DE5-NET [Tec], fournie par Terasic et se base sur un FPGA Altera Stratix V 5SGXEA7N2F45C2 [Alt15]. Cette carte contient également 2 Go de mémoire DDR3 (mémoire globale OpenCL). La mémoire locale est de la mémoire interne au FPGA, ce qui offre plus de flexibilité que pour le GPU. Ce FPGA contient 234,720 ALMs, 938,880 registres, 256 DSPs et 52,428,800 éléments mémoires (avec 2560 blocs de RAM M20K).

D'un point de vue logiciel, l'hôte accueille une distribution GNU/Linux basée sur un noyau 3.2. Tous les programmes détaillés sont écrits en C ou C++. La carte NVidia supporte une version 1.1 d'OpenCL, fournie par le constructeur. La carte Altera supporte la version 14.1 de Quartus, qui implémente une version modifiée d'OpenCL 1.2.

Les travaux ont précédemment été déployés sur un FPGA Stratix IV, avec une version 13.1 de Quartus et une version beta d'OpenCL 1.1. Cependant, la double précision flottante n'était pas supportée, et nous rencontrions des difficultés lors de la synchronisation de calculs sur cible. Les résultats pour cette carte ne sont pas présentés, d'autant que seule la première architecture pouvait être déployée.

3.5.2 Résultats pour l'évaluation d'options américaines

3.5.2.1 Critères de performance

La performance minimale que l'évaluateur doit atteindre correspond à un cas d'utilisation pour le tracé de courbes de volatilité, soit un objectif de 2000 options calculées par seconde, avec un budget énergétique de 20 W. Cet objectif est en accord avec les contraintes exprimées par un partenaire du domaine financier, présent lors de l'élaboration du cahier des charges. Il permet à la fois une bonne réactivité pour le calcul des courbes de volatilité et une performance suffisante pour l'évaluation d'options. Cet objectif d'une courbe en une seconde est ambitieux et plus performant que les solutions actuelles. Il permet d'envisager un calcul plus fréquent des courbes de volatilité, ce qui limite les risques encourus par un trader, en conservant le modèle proche de la réalité. De plus, ce critère de performance permet un débit plus que suffisant pour l'évaluation d'options en dehors du cadre de la calibration du modèle. Le budget de 20 W, quand à lui, facilite l'intégration dans les stations de travail.

La précision des résultats est assurée par la représentation des données en format flottant double précision, ainsi qu'en choisissant un pas de discrétisation de $T = 800$; ce pas de discrétisation est empiriquement vérifié comme étant suffisant pour ce cas d'utilisation. Il provient également de discussions avec des acteurs du domaine. Ces paramètres forment un compromis convenable entre vitesse d'exécution, précision et ressources matérielles utilisées (notamment en termes de ressources mémoire et de DSP utilisées). Des performances supérieures peuvent être atteintes selon un processus d'optimisation long, en utilisant des types de données parfaitement adaptés à l'application. Cette aproche est usuelle pour la conception d'architectures sur FPGA. Nous avons choisi de délaisser cette tâche. L'environnement OpenCL supporte nativement la double précision flottante. Les gains que nous aurions pu obtenir en virgule fixe n'auraient pas été suffisants pour justifier les temps de développement associés, qui auraient diminué l'intérêt d'une appproche de conception haut-niveau *via* OpenCL.

3.5.2.2 Optimisation de l'utilisation des ressources matérielles

Le tableau 3.1 résume les ressources utilisées pour chacune des deux architectures sur le Stratix V.

Ces résultats sont fournis par l'outil de synthèse Quartus II (*"Fitter Summary"*) après l'exécution du compilateur OpenCL d'Altera. La consommation énergétique des noyaux quand à elle provient de l'outil d'estimation de puissance de Quartus (Quartus Power Estimation - *quartus_pow*). Nous nous concentrons sur l'utilisation d'ALM (*Adaptive Logic Modules*), de registres, d'éléments mémoire et de DSP.

Il est possible de paralléliser chaque noyau plusieurs fois, à l'aide d'options du compilateur OpenCL d'Altera. Dans un premier temps, des directives de compilation peuvent être utilisées pour soit dupliquer des pipelines matériels complets, soit vectoriser l'exécution d'un noyau. Lorsque le pipeline est dupliqué, les calculs sont réalisés de manière indépendente. La vectorisation quand à

elle correspond à une division en SIMD des tâches. Selon nos propres observations, la vectorisation consomme généralement moins de ressources. La vectorisation est toutefois plus contraignante : celle-ci doit être un multiple de deux (*i.e.* vectorisation par 2, 4, 6, etc.) et le facteur de vectorisation doit être un diviseur de la taille totale du *work-group* (le nombre d'itérations étant un entier). En plus de la vectorisation ou la duplication de noyau, il est possible de dérouler des boucles internes, à l'aide de directives de précompilation (*#pragma*). Le déroulage de boucle consomme moins de mémoire qu'une duplication complète, tout en fournissant un autre degré de liberté pour augmenter le débit et optimiser l'utilisation de ressources. Toutefois, un déroulage trop important des boucles peut avoir un impact très négatif sur les temps d'accès à la mémoire locale.

La sélection des paramètres de parallélisation optimaux est un problème complexe et long à réaliser. La synthèse d'une architecture prend aux alentours de 10 heures pour chaque changement de paramètres. Une étude exhaustive n'est donc pas envisageable. Nous avons constaté durant la phase d'optimisation de l'architecture qu'un noyau non parallélisé n'utilisait pas toutes les ressources disponibles, nous avons donc cherché à maximiser le parallélisme, en privilégiant si possible la duplication. Dans le cas considéré, le noyau en flot de donnée présente un facteur 2 de vectorisation, pour un facteur de duplication de 3 (facteur de parallélisation de 6). Ces paramètres permettent d'utiliser au mieux les ressources du circuit FPGA ciblé. Le noyau de la seconde architecture décrite est dupliqué deux fois, pour un facteur de vectorisation de 4 (facteur de parallélisation de 8), pour un format de données en double précision. En simple précision, ce même noyau peut être dupliqué trois fois, pour un facteur de vectorisation de 4 (facteur de parallélisation de 12) et ce, sans diminution significative de la fréquence d'horloge. En effet, le parallélisme important de l'architecture, ainsi que la nature du calcul effectué forcent l'utilisation de ressources génériques (ALM) pour certaines opérations flottantes. Dans des conditions optimales, celles-ci seraient calculées par des DSP, qui sont la ressource limitante sur le FPGA utilisé. Les options de parallélisation pour ces deux noyaux sont le fruit d'itérations de cycles de compilation jusqu'à obtenir le meilleur taux d'utilisation de ressources. Ces résultats sont l'optimal observé pour chaque noyau sur cette cible. Pour des raisons de clarté, nous ne présentons pas les résultats pour d'autres combinaisons, dont l'intérêt est moindre.

3.5.2.3 Performances des noyaux

Le tableau 3.3 décrit les performances de chaque noyau pour une implémentation sur GPU et FPGA de l'évaluation d'options binomiales, une référence logicielle et les résultats de la littérature pour comparaison. Tous les résultats présentés sont échantillonnés après saturation des plateformes d'accélération, ce qui correspond aux meilleures conditions d'utilisation. Après saturation, le temps de calcul est une fonction linéaire du nombre d'options à évaluer. On considère en effet qu'il n'y a pas de perte de données d'entrée au vu de l'espace disponible en mémoire globale (2 Go sur carte Terasic) et de l'empreinte mémoire occupée pour une option en attente d'évaluation (quelques octets). Cette saturation est observable aux alentours de 10^5 options. Ce scénario est facilement atteignable pour un cas d'utilisation réel. Le seul noyau présentant un taux de saturation plus élevé

est l'implémentation du noyau optimisé sur cible GTX660 (10^6 options pour une précision simple comme double). Dans ce cas, pour garder un débit optimal, il est nécessaire de considérer des cas d'utilisation avec un plus grand volume de données à traiter, ce qui se traduit par une augmentation de la latence. Cet équilibre s'effectue au détriment de l'efficacité de l'accélérateur pour des charges de travail plus faibles. Notre implémentation est conçue pour répondre aux besoins d'un unique trader et non pas pour servir de ressource partagée (i.e. en tant que service). Les résultats présentés dans ce cas précis sont à nuancer si on ne parvient pas à garantir la saturation.

L'utilisation de blocs RAM (M20K) diffère entre les deux noyaux. La version en flot de données utilise la RAM pour agréger les accès à la mémoire globale et stocke ces entrées et sorties dans des petites FIFO, tandis que les RAM forment la mémoire locale du noyau optimisé. La puissance consommée par ces noyaux (15 W et 17 W) sont des limites supérieures de la puissance effectivement consommée et sont acceptables en comparaison du budget énergétique disponible. Cette puissance consommée n'inclut toutefois ni la consommation des mémoires globales (moins de 1.2 W pour notre cas [kin13]), ni la consommation de composants autres que la puce FPGA sur la carte. Celle-ci est néanmoins responsable de la majorité de la puissance consommée ; ces résultats représentent une bonne approximation de la puissance totale consommée par la carte DE5-NET.

En revanche, les résultats obtenus pour le noyau en flot de données n'atteignent pas les niveaux espérés, que ce soit sur FPGA ou sur GPU. Ceci s'explique par la copie de données de la mémoire globale vers l'hôte : une des deux mémoires tampons en ping - pong est entièrement lue entre deux groupes de noyaux exécutés (soit environ 19 MB pour $N = 1024$ et 15 MB pour $N = 800$). Cette opération de lecture stoppe le noyau pour éviter d'écraser les résultats, ce qui pénalise le temps d'exécution.

Comme attendu, le noyau optimisé présente des résultats plus prometteurs. Un peu moins de 2000 options peuvent être calculées en une seconde sur le FPGA (1930 options/s). La limitation du facteur de parallélisation qu'impose le nombre de DSP du FPGA utilisé laisse envisager des résultats encore plus intéressants sur un FPGA plus adapté. L'écart en performance entre l'implémentation sur FPGA et celle sur GPU s'explique par le grand nombre de multiprocesseurs en *streaming* présents sur le GPU. Ceci implique une parallélisation plus importante. Le GTX660 est composé de 960 processeurs en *streaming* (i.e., des cœurs de calcul CUDA), avec une *Arithmetic Logic Unit* (ALU) double précision par tranche de 8 processeurs (120 DP-ALU), pour une fréquence d'horloge de 980 MHz. Ces ressources sont plus importantes que celles présentes sur la carte FPGA utilisée. Le revers de la médaille est que le GPU nécessite une charge de travail plus importante pour atteindre des performances optimales (10 fois plus d'options à évaluer). De plus, l'écart induit en latence par un tel nombre d'options ne s'accompagne pas d'un gain en accélération important, étant donnée que le nombre d'options calculées par seconde sur GTX660 et sur FPGA ne sont éloignés que d'un facteur 5.5. Si on ajoute à cela la consommation électrique accrue du GPU, les résultats qui semblent sans appel en première lecture sont en réalité très mitigés. Il est difficile de conclure de manière générale sur le choix entre les deux accélérateurs. La solution sur GPU est intéressante si les performances brutes sont considérées, mais elle requiert plus d'espace, de systèmes

de refroidissement et de puissance électrique qu'une solution sur FPGA. Avec un budget énergétique de l'ordre de quelques dizaines de Watts, on peut envisager d'intégrer une solution d'accélération à base de FPGA au sein de serveurs existant, sans revoir toute l'architecture [PCC+14]. Le GPGPU peut nécessiter une refonte complète d'une salle de serveurs ou d'une infrastructure informatique.

Le tableau 3.3 résume les performances obtenues par des travaux proches de ceux présentés ici. Les meilleurs résultats sont obtenus par les travaux décrits par Tavakkoli et al. [TT14] (*cf.* section 3.2.5). Cependant, même si nos travaux ne peuvent pas atteindre les performances brutes d'une implémentation HDL optimisée en l'état actuel des technologies, les résultats présentés sont à nuancer. La cible de ces travaux est un FPGA de plus haute capacité, le Xilinx Virtex 7 (xc7vx980t), sur lequel est implémenté une architecture conçue par une approche HDL traditionnelle. La puissance consommée n'est pas fournie dans l'article, mais est supposée raisonnablement faible. La consommation du Virtex 7 est comparable à celle d'un Stratix V, pour une performance brute plus importante. Toutefois, d'après notre étude de cas, cette approche présente plusieurs inconvénients. L'objectif de l'implémentation que nous proposons ici est d'être aisément modifiée, intégrée dans une application existante et accessible pour un développeur qui n'est pas expert dans la conception de circuit. L'architecture proposée suit les lignes directrices de la norme OpenCL, notamment en termes de duplication de noyau, de déroulage de boucle et d'utilisation de *channels*. Les travaux de Tavakkoli et al. [TT14] est efficace et serait certainement une solution séduisante si l'on souhaitait uniquement obtenir une implémentation dédiée pour un algorithme spécifique, sans le modifier dans les années qui viennent. Malheureusement, le temps de développement nécessaire pour développer une telle architecture, ainsi que l'adéquation entre l'algorithme et l'architecture font que toute modification, même mineure risque de s'accompagner de redéveloppement majeur de l'ensemble du système. L'approche OpenCL atteint les niveaux de performance requis tout en étant suffisamment flexibles pour supporter des changements de l'algorithme sous-jacent. Une autre différence clé entre les deux approches est visible en étudiant les cibles matérielles utilisées. Les travaux de [TT14] ciblent un FPGA à haute capacité contenant plus de 3000 DSP. Notre cible quand à elle contient seulement 256 blocs de DSP. Les résultats en débit pour l'architecture optimisée en double précision atteignent $0.6.10^9$ nœuds par seconde. Cependant, la même architecture déployée sur un Stratix IV contenant 1024 blocs de DSP doublait ce débit pour une fréquence plus faible. L'utilisation d'un FPGA plus adapté en termes de ressources intégrées permettra d'atteindre des débits plus élevés en ne modifiant que le paramétrage du parallélisme. De plus, notre implémentation fonctionne en double précision, ce qui est une contrainte forte des partenaires du domaine financier. La même implémentation en simple précision double le débit. L'approche de Tavakkoli *et al.* utilise un format à virgule fixe, qui impose des temps de développement plus longs et qui doit encore faire ses preuves auprès de partenaires du domaine. Finalement, les cibles matérielles sont trop différentes pour obtenir une comparaison équitable.

TABLEAU 3.1 – Ressources consommées pour la synthèse de noyaux d'évaluation d'options

	Stratix V 5SGXEA7N2F45C2		
Type de noyau	Flot de données	Optimisé	
Précision	Double	Simple	Double
ALM	135,689 (58%)	154,487 (68 %)	152,275 (65%)
Registres	262,622 (28%)	226,659 (24%)	217,469 (23%)
Éléments Mémoires (bits)	20,996,740 (40 %)	15,648,147 (30 %)	15,002,238 (29 %)
dont M20K	1,718 (67%)	1,303 (51 %)	1,164 (45%)
DSP (18-bit)	240 (94%)	256 (100 %)	256 (100%)
Fréquence d'Horloge	198 MHz	156 MHz	160 MHz
Puissance Consommée	17W	15 W	15 W

TABLEAU 3.2 – Performances obtenues pour l'évaluation d'options binomiales

	Flot de donnée		Optimisé			
Plateforme	FPGA	GPU	FPGA	FPGA	GPU	GPU
Précision	Double	Double	Double	Simple	Simple	Double
options/s	25	53	1930	2790	83,300	11,100
RMSE	0	0	0	$\sim 10^{-4}$	$\sim 10^{-4}$	0
options/J	1.7	0.4	128	186	595	79.3
nœuds/s	13 M	30 M	0.62 G	0.90 G	27 G	3.6 G

3.5.3 Performance de l'implémentation du calcul de volatilité implicite

Comme décrit dans la section 3.4, le calcul de la volatilité implicite repose sur la méthode de Newton-Raphson. Étant données les performances des noyaux d'évaluation d'options, le noyau optimisé est utilisé pour évaluer les prix d'options itérativement.

Comme pour l'étude de performance de noyau d'évaluation d'options, certains paramètres sont fixés de manière empirique. Le nombre de pas de discrétisation temporelle est fixé à $N = 800$ et le nombre d'itérations de la méthode de Newton Raphson à $N_{NR} = 100$. N_{NR} est le nombre maximal d'itérations nécessaire pour obtenir une valeur de volatilité implicite avec une précision acceptable (10^{-5}). Dans un cas usuel, la valeur de la volatilité implicite est obtenue en moins d'itérations. Fixer $N_{NR} = 100$ permet de garantir une borne inférieure pour les performances de la solution proposée.

L'architecture décrite dans la section 3.4 est implémentée et compilée pour la cible FPGA. Comme précisé précédemment, l'implémentation sur GPU de l'architecture proposée n'est pas encore possible, pour des raisons de compatibilité de versions d'OpenCL. Les résultats sont résumés dans le tableau 3.4. Les bénéfices de l'approche de conception OpenCL se sont fait ressentir lors des cycles de développement, très courts par rapport à un cycle de développement HDL classique.

TABLEAU 3.3 – Performances de référence pour l'évaluation d'options binomiales

	Logiciel de référence		[Jin et al. 2008]	[Wynnyk et al. 2009]	[Tavakkoli et al. 2014]
Plateforme	Single Core Xeon X5450		Virtex 4 xc4vsx55	Stratix III EP3SE260	Virtex 7 xc7vx980t
Précision	Simple	Double	Double	Double	Fixe 16.16
options/s	550	720	385	1152	93,000
RMSE	$\sim 10^{-4}$	0	0	0	$\sim 10^{-4}$
options/J	4.6	6	N/A	N/A	N/A
nœuds/s	0.18 G	0.23 G	202 M	576 M	29.95 G

La description de l'architecture finale de l'application consiste en environ 200 lignes de code, ce qui est bien moins que l'équivalent en lignes de code HDL et plus simple à déboguer qu'une approche bas niveau.

Comme pour l'application d'évaluation d'options, une recherche du facteur de parallélisation optimal a été effectuée. La configuration retenue permet de placer trois noyaux (facteur de duplication de 3) évaluant des options et un noyau de calcul de volatilité. Lors des premiers tests, la comparaison des performances entre ces deux noyaux a montré une prédominance de l'évaluation d'options dans le temps de calcul global de l'application. Il paraît donc logique de concentrer les efforts de parallélisation sur ce noyau. La puissance consommée par cette architecture est d'environ 17 W, pour une fréquence d'horloge de 179 MHz.

Les résultats obtenus sur l'accélérateur sont identiques à la référence logicielle, sans perte de précision (pour un format de donnée flottant double précision). Le temps de calcul sur accélérateur (communication entre l'hôte et l'accélérateur comprise) est deux fois moins long que notre référence logicielle (soit un facteur d'accélération de 2). Les performances obtenues sont en majeure partie limitées par le noyau d'évaluation d'options et, au vu des contraintes matérielles et des résultats précédents, ce ratio est proche de la valeur optimale pour l'approche proposée. Dans le pire cas, 100 itérations sont nécessaires pour calculer une valeur de volatilité implicite avec une précision acceptable. Une itération utilise 2000 options et notre noyau d'évaluation d'option peut calculer environ 2000 options/s. Cela signifie que le tracé d'une courbe de volatilité (en négligeant l'étape d'interpolation) nécessite environ 100 s. Dans cette configuration, la référence logicielle nécessite 200 s pour calculer les valeurs nécessaires au tracé d'une courbe.

La ressource limitant cette implémentation est une fois encore le nombre de blocs DSP intégrés à la matrice du FPGA. Le choix d'une cible matérielle plus adaptée, présentant plus de DSP (soit dans la même gamme de FPGA - Stratix V, soit dans une gamme supérieure) devrait entraîner un facteur d'accélération plus important.

TABLEAU 3.4 – Ressources consommées pour le calcul de volatilité implicite

	Stratix V 5SGXEA7N2F45C2		
Noyau	Architecture complète (4 noyaux)	Noyau principal	Noyau d'éval. d'options
Éléments Logiques	123,227 (52%)	24,210 (10%)	23,584 (10%)
Registres	245,925 (26%)	62311 (7%)	58,804 (6%)
Éléments mémoire (bits)	29,305,105 (56%)	11,245,723 (21%)	7,978,259 (15%)
dont M20K	2,147 (84 %)	738	540
DSP (18-bit)	248 (97 %)	82 (32%)	83 (32%)

Discussion

Les résultats présentés dans ce chapitre montrent ce qu'OpenCL apporte à une démarche d'optimisation d'application via leur accélération sur FPGA. Les résultats obtenus pour l'évaluation d'options affichent un bon facteur d'accélération, pour un temps de développement et d'intégration minimal lorsqu'ils sont comparés à une approche HDL traditionnelle de conception et de prototypage sur FPGA. Pour prendre l'exemple de l'application d'évaluation d'options, un mois s'est écoulé entre la conception initiale de l'architecture et son intégration dans un environnement d'expérimentation. L'application développée est de plus portable d'une plateforme à une autre : elle peut être utilisée sur GPU ou GPP, pourvu que l'environnement OpenCL de ces plateformes soit compatible et que la consommation énergétique de la solution ne soit pas un facteur limitant.

Implémenter le calcul de volatilité implicite sur une architecture parallèle est un véritable défi du fait de sa nature séquentielle. La dépendance de donnée entre le processus d'évaluation d'option et l'approximation itérative de la volatilité diminue grandement le potentiel de parallélisation qu'il est possible d'atteindre. Toutefois, notre implémentation présente des résultats deux fois plus rapides qu'une solution implémentée sur GPP. L'accélération est majoritairement limitée par le nombre de blocs DSP présents sur la carte utilisée et peut donc être accrue en ciblant des FPGA de plus grande capacité.

La majorité du temps de développement pour cette solution a été consacrée à une étude générale de l'architecture, ainsi que le débogage des *channels* pour le transfert de données, qui limitaient les performances initiales. Si le modèle d'évaluation d'option évolue, ce temps de développement pourra être capitalisé et l'architecture restera en grande partie inchangée. On atteint ici une des limites de l'approche OpenCL sur FPGA. Même si les technologies utilisées sont abordables pour un expert logiciel, une recherche architecturale proche de celle réalisée pour une implémentation HDL a tout de même été réalisée. Cela nécessite une bonne connaissance des spécificités de la plateforme, ce que l'on souhaitait justement éviter. Ce constat est renforcé lors du débogage de l'application. Pour comprendre quelles portions du code posent problème, il faut comprendre comment elles se traduisent sur la plateforme matérielle.

Malgré ce point propre à l'utilisation d'OpenCL sur FPGA, l'approche proposée ici permet tout de même d'atteindre les objectifs fixés, en termes de puissance consommée, de temps de calcul et

de précision. L'intégration et la comparaison avec le logiciel sont également grandement facilitées par l'utilisation d'un environnement qui s'exécute sur l'hôte.

3.6 Conclusion

Nous avons présenté une étude d'architecture sur un cas concret, à savoir l'évaluation d'options américaines, sur cible FPGA. Deux optiques sont détaillées : une première implémentation repose sur un calcul en flot de donnée, tandis que la seconde se concentre sur l'optimisation locale des accès mémoires. L'implémentation a été réalisée à l'aide d'un flot haut-niveau (Compilateur Altera OpenCL) et nous avons obtenu des performances conséquentes pour la deuxième approche, avec un budget énergétique faible. Cette architecture a ensuite été étendue à un problème plus complexe - le calcul de volatilité implicite.

Les temps de développement relativement courts pour une telle architecture valident cette approche haut-niveau pour le développement d'accélérateur de calcul sur matériel. La solution d'accélération développée est toutefois très spécialisée ; sa modification pour étendre ses fonctionnalités ou la mettre à jour n'est donc pas triviale. Un équilibre entre la stratégie de développement détaillée ici et l'intégration au sein d'une librairie logicielle (telle que détaillée chapitre 2) est trouvé dans le chapitre 4. Celui-ci repose sur une chaîne de précompilation de code spécifique au domaine pour être en adéquation avec la plateforme cible.

Chapitre 4

Chaîne de compilation spécialisée pour le calcul d'options

NOUS avons présenté dans le chapitre 1 les spécificités du domaine financier et en quoi leur implémentation logicielle s'oppose à une implémentation performante sur matériel. L'implémentation d'une solution optimisée pour le calcul d'option américaine (et de volatilité implicite) confirme l'utilité d'une appproche haut-niveau, ainsi que la possibilité d'atteindre des performances acceptables de la sorte. Toutefois, l'optimisation d'une solution spécifique présente ses propres inconvénients. Cette spécialisation demande à l'utilisateur une connaissance poussée des outils utilisés ainsi que de l'accélérateur ciblé. De plus, il n'est pas assuré que l'application développée puisse évoluer et être améliorée pour répondre à des modifications des besoins de l'utilisateur. Nous présentons dans ce chapitre les éléments d'une solution spécifique au domaine financier et ouvrant l'accès en programmabilité à des solutions d'accélération sur FPGA, via l'utilisation d'OpenCL.

Nous caractérisons dans un premier temps une plateforme matérielle spécifique, ce qui fournit une base sur laquelle nous proposons un flot de programmation liant le domaine applicatif au flot de compilation, synthèse et exécution OpenCL d'Altera. Nous définissons un état de l'art sur les langages de programmation spécifiques à un domaine (DSL : *Domain Specific Language*) et nous appliquons les conclusions de l'état de l'art pour déterminer comment appliquer les spécificités du calcul d'options financières pour leur implémentation sur plateforme matérielle. Nous décrivons finalement en détail l'architecture d'un outil de compilation traduisant un programme financier écrit dans un langage de haut-niveau (C) en une représentation intermédiaire qui est utilisée en entrée du flot de compilation d'Altera pour OpenCL.

4.1 Contexte : chaîne de compilation classique et OpenCL

4.1.1 Compilation : de la représentation humaine à la représentation machine

Le développement logiciel implique généralement la représentation d'un programme en utilisant un langage adapté au calcul souhaité. Ce langage est souvent conçu pour proposer un compromis entre utilisabilité et performance (temps de calcul, précision, consommation énergétique). L'utilisabilité permet de simplifier le processus de développement, en proposant au développeur des mécanismes proches de ceux utilisés dans son domaine d'application, afin de rendre l'écriture la plus naturelle possible. Elle passe généralement par l'utilisation de niveaux d'abstraction très hauts, permettant de masquer la complexité de la machine. La performance permet d'optimiser l'exécution du programme par la machine. Elle est souvent en opposition avec l'utilisabilité, car elle implique une représentation adaptée à la machine, et non au domaine d'application.

Afin d'être effectivement exécuté, un programme écrit dans un langage de programmation doit ensuite être compilé. L'étape de compilation consiste en la transformation d'un programme représenté dans un langage de haut-niveau d'abstraction, en une représentation compréhensible par la machine cible. Un compilateur est généralement divisé en plusieurs composants, chacun en charge d'une fonctionnalité distincte. De manière simplifiée, on retrouve généralement trois composants principaux :

— un *front-end* qui analyse le code de l'utilisateur, en vérifie la syntaxe et le traduit en un format de représentation interne au compilateur,
— un étage d'optimisations successives, indépendantes du langage initial et de la cible matérielle. Cet étage est organisé en série de passes optimisées pour une tâche particulière. La dépendance entre ces passes et leur caractère optionnel ou obligatoire est à la charge du compilateur, leur utilisation dépendant du niveau d'optimisation paramétré par l'utilisateur.
— un *back-end*, qui traduit la réprésentation optimisée du programme en un code compréhensible par la cible matérielle

Cette subdivision permet d'obtenir une architecture de compilation flexible, pouvant accommoder plusieurs langages de programmation et plusieurs cibles matérielles. Les cibles sont traditionnellement des processeurs, pour lesquels il est possible de réaliser des optimisations spécifiques dans le *back-end*, en fonction du type de processeurs ou des versions d'un même processeur. Mais le *back-end* peut aussi être utilisé pour générer du code de haut-niveau, ou directement des architectures matérielles.

4.1.2 Synthèse de haut niveau et représentation interne d'un programme

Comme nous l'avons évoqué précédemment, le développement sur FPGA souffre de la complexité du flot de conception et du niveau d'abstraction très bas auquel il est réalisé. La plupart des développeurs souhaitent décrire la fonctionnalité de leur programme, sans forcément contrôler toute l'architecture qui l'accompagne. C'est pour permettre cette approche que la synthèse de haut niveau (*High Level Synthesis*, HLS) est développée et prend de plus en plus d'ampleur.

Lorsqu'on utilise un outil de HLS, on décrit la fonctionnalité souhaitée dans un langage logiciel classique. Le compilateur va ensuite générer une architecture matérielle en lieu et place du code assembleur. Cette génération peut être complètement automatique, ou bien guidée par l'utilisateur, à l'aide de directives et de commandes.

LLVM est un *framework* de compilation qui suit l'architecture *front-end*/optimisation/*back-end* présentée précedemment. Il se prête particulièrement bien à la HLS, en particulier grâce à sa représentation intermédiaire très flexible. Le format de cette représentation (appelée IR pour *Intermediate Representation*) est un format de code assembleur lisible. Un programme sous cette forme est regroupé sous une structure appelée *module*, contenant des *functions* elles-mêmes subdivisées en *basic blocks* (séquence d'instruction présentant un unique point d'entrée et un unique point de sortie). L'IR utilisée par LLVM a la particularité d'être une représentation de la forme *Static Single Assignment* (SSA). Ceci signifie que chaque variable est assignée une et une seule fois. Les variables du code d'origine sont représentées par différentes versions pour respecter cette forme. Cette représentation permet de simplifier certaines optimisations, en particulier sur l'ordre dans lequel les opérations ont lieu.

Le compilateur OpenCL d'Altera (AOC) est basé sur LLVM 3.0 pour la génération de noyaux portés sur FPGA. AOC utilise des versions internes des outils LLVM pour créer un flot de compilation du code OpenCL de l'utilisateur et l'IR qu'il génère est donc spécifique. Un exemple de Représentation Intermédiaire généré par AOC en amont de son flot d'optimisation est donné ci-après 4.1. Une description complète de l'IR est disponible pour chaque version de l'outil.

```
1
2  ; ModuleID = '[path to board support package]/vectorLocAdd.cl'
3  target datalayout = "e-p:64:64:64-n8:16:32:48:64-a0:0:64-s0:0:64-S128-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64
       :64:64-f16:16:16-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64:64:64-v128:128:128-v256:256:256-v512
       :512:512-v1024:1024:1024-v24:32:32-v48:64:64-v96:128:128-v192:256:256"
4  target triple = "fpga64"
5
6  @vectorAdd.local.xl = internal addrspace(2) global float 0.000000e+00, align 4
7  @vectorAdd.local.yl = internal addrspace(2) global float 0.000000e+00, align 4
8  @vectorAdd.local.zl = internal addrspace(2) global float 0.000000e+00, align 4
9
10 define void @vectorAdd(float addrspace(1)* align 1024 %x, float addrspace(1)* align 1024 %y, float addrspace
       (1)* noalias align 1024 %z) nounwind {
11 entry:
12   %x.addr = alloca float addrspace(1)*, align 8
13   %y.addr = alloca float addrspace(1)*, align 8
14   %z.addr = alloca float addrspace(1)*, align 8
15   %index = alloca i32, align 4
16   store float addrspace(1)* %x, float addrspace(1)** %x.addr, align 8, !tbaa !5
17   store float addrspace(1)* %y, float addrspace(1)** %y.addr, align 8, !tbaa !5
18   store float addrspace(1)* %z, float addrspace(1)** %z.addr, align 8, !tbaa !5
19   %0 = bitcast i32* %index to i8*
20   call void @llvm.lifetime.start.p0i8(i64 -1, i8* %0)
```

```
21    %call = call i64 @___acl_wide_get_global_id(i32 0) nounwind readnone
22    %conv = trunc i64 %call to i32
23    store i32 %conv, i32* %index, align 4, !tbaa !8
24    %1 = load i32* %index, align 4, !tbaa !8
25    %idxprom = sext i32 %1 to i64
26    %2 = load float addrspace(1)** %x.addr, align 8, !tbaa !5
27    %arrayidx = getelementptr inbounds float addrspace(1)* %2, i64 %idxprom
28    %3 = load float addrspace(1)* %arrayidx, !tbaa !9
29    store float %3, float addrspace(2)* @vectorAdd.local.xl, align 4, !tbaa !9
30    %4 = load i32* %index, align 4, !tbaa !8
31    %idxprom1 = sext i32 %4 to i64
32    %5 = load float addrspace(1)** %y.addr, align 8, !tbaa !5
33    %arrayidx2 = getelementptr inbounds float addrspace(1)* %5, i64 %idxprom1
34    %6 = load float addrspace(1)* %arrayidx2, !tbaa !9
35    store float %6, float addrspace(2)* @vectorAdd.local.yl, align 4, !tbaa !9
36    %7 = load float addrspace(2)* @vectorAdd.local.xl, align 4, !tbaa !9
37    %8 = load float addrspace(2)* @vectorAdd.local.yl, align 4, !tbaa !9
38    %add = fadd float %7, %8
39    store float %add, float addrspace(2)* @vectorAdd.local.zl, align 4, !tbaa !9
40    %9 = load float addrspace(2)* @vectorAdd.local.zl, align 4, !tbaa !9
41    %10 = load i32* %index, align 4, !tbaa !8
42    %idxprom3 = sext i32 %10 to i64
43    %11 = load float addrspace(1)** %z.addr, align 8, !tbaa !5
44    %arrayidx4 = getelementptr inbounds float addrspace(1)* %11, i64 %idxprom3
45    store float %9, float addrspace(1)* %arrayidx4, !tbaa !9
46    ret void
47 }
48
49 declare void @llvm.lifetime.start.p0i8(i64, i8* nocapture) nounwind
50
51 declare i64 @___acl_wide_get_global_id(i32) nounwind readnone
52
53 !opencl.kernels = !{!0}
54 !altera.opencl.ir.version = !{!1}
55 !acl.opencl.kernels = !{!2}
56
57 !0 = metadata !{void (float addrspace(1)*, float addrspace(1)*, float addrspace(1)*)* @vectorAdd}
58 !1 = metadata !{metadata !"0.0"}
59 !2 = metadata !{metadata !"name", metadata !"vectorAdd", metadata !"attrs", metadata !3}
60 !3 = metadata !{metadata !"local_mem_size", metadata !4, metadata !"pointer_mem_location", metadata !4}
61 !4 = metadata !{}
62 !5 = metadata !{metadata !"any pointer", metadata !6}
63 !6 = metadata !{metadata !"omnipotent char", metadata !7}
64 !7 = metadata !{metadata !"Simple C/C++ TBAA", null}
65 !8 = metadata !{metadata !"int", metadata !6}
66 !9 = metadata !{metadata !"float", metadata !6}
```

Listing 4.1 – IR spécifique à AOC pour un noyau simple (addition)

LLVM permet d'annoter le code généré grâce à des métadonnées, indiquées par un point d'exclamation en fin de code. Celles-ci sont utilisées dans ce cas pour fournir des informations propres à OpenCL. Les noyaux (un seul dans l'exemple ci-dessus, vectorAdd) ainsi que leurs entrées sont définies, de même que d'autres métadonnées liées à l'outil (version, cible, etc.). A titre d'exemple, on peut examiner plus en détail la série d'instructions décrivant l'initialisation et l'utilisation de mémoire locale. L'allocation d'une zone mémoire locale demande la création d'une variable globale correspondant à l'adresse 0 de la mémoire, avec les caractéristiques des données à stocker :

```
1 @vectorAdd.local.xl = internal addrspace(2) global float 0.000000e+00, align 4
```

Le stockage d'un élément dans cette mémoire passe par l'instruction *store*, une fois la donnée récupérée (ici, en provenance de la mémoire globale) :

```
1 store float %3, float addrspace(2)* @vectorAdd.local.xl, align 4, !tbaa !9
```

Une donnée stockée en mémoire locale peut ensuite être lue et utilisée via l'instruction *load* :

```
1 %7 = load float addrspace(2)* @vectorAdd.local.xl, align 4, !tbaa !9
```

Dans cet exemple, les mémoire locales ne contiennent qu'un élément, ce qui simplifie leur adressage et les opérations de lecture et d'écriture de manière générale.

4.1.3 Utilisation du flot AOC pour la réalisation d'un DSL

Dans ce chapitre, nous proposons un flot de conception permettant l'écriture d'un DSL qui peut ensuite être accéléré sur FPGA sans intervention du développeur. Nous avons déjà identifié OpenCL comme une technologie intéressante pour cet objectif. Pour permettre ce DSL, il faut cependant pouvoir générer du code compatible pour OpenCL à partir d'un autre langage, ce qui peut se révéler fastidieux. La génération de code est en effet un processus complexe, et le résultat est souvent compliqué à valider ou à comprendre par un humain.

AOC repose sur LLVM pour générer une représentation du programme qui peut être traitée par leurs outils de synthèse. Même si les passes spécifiques à la génération de noyaux pour FPGA ne sont disponibles que sous forme de binaires, il est possible de développer nos propres passes d'optimisation, qui pourront être appliquées en amont. Ceci permet donc d'envisager de modifier le format d'entrée du programme pour le flot AOC, en générant un IR correct à partir d'un langage à définir. Cette approche permet de se dispenser de la génération de code C de noyau OpenCL fonctionnel. La génération de code C à partir d'IR est une opération possible, mais elle présente en effet certaines difficultés. Il faut en effet s'assurer que le code ainsi généré suive une syntaxe fonctionnelle et lisible par un utilisateur (sans quoi ces efforts présentent un intérêt bien moindre).

De plus, se placer au niveau de l'IR d'un programme donne l'accès à un format de données suffisamment neutre pour se positionner précisément entre tout type de *front-end* et de *back-end*. Il est envisageable de réutiliser la majeure partie d'un tel outil pour cibler d'autres plateformes matérielles, ou plus prosaïquement une nouvelle version d'AOC. L'IR précédente 4.1 est extraite du

flot de compilation d'AOC avant les majeures passes d'optimisation de ce flot et donc à un bon point d'entrée pour une surcouche externe.

Nous réalisons dans les deux sections qui suivent la caractérisation d'une plateforme matérielle, cible finale du *back-end* considérée, ainsi qu'un état de l'art sur les DSL. Ces sections ont pour objectif de délimiter l'espace de liberté existant :

— pour optimiser un programme en cohérence avec les propriétés de la plateforme cible,
— et pour développer un *front-end* spécifique au domaine financier et viable pour notre cas d'utilisation.

4.2 Caractérisation d'une plateforme matérielle et du flot OpenCL d'Altera

4.2.1 Méthodologie d'analyse

Lorsque l'on cherche à compiler un programme sur une plateforme cible, il est nécessaire de connaître cette plateforme avant d'envisager une approche performante. Cette connaissance permet de réaliser une étape d'optimisation du programme de l'utilisateur pour l'adapter à l'accélérateur. Les ressources présentes sur la carte et leur taux d'utilisation pour la synthèse d'opérations élémentaires sont en effet nécessaire pour évaluer au mieux les capacités de parallélisation d'un programme. De plus, l'estimation du débit de chaque étape du calcul (de l'échange de données entre le GPP hôte et l'accélérateur à la latence d'un calcul donné) permet de fournir une estimation grossière de performance à l'utilisateur avant l'étape de synthèse.

Le but de cette section est de présenter l'étude menée pour caractériser les résultats de synthèse du compilateur OpenCL d'Altera sur une plateforme matérielle cible. En s'appuyant sur les informations mises à disposition, une analyse de l'architecture matérielle générée a été réalisée. Cette analyse est complétée par la réalisation d'un banc d'étude ciblé, permettant d'évaluer les capacités de l'outil sur certaines opérations et sur les accès mémoires.

Le banc de test mis en place a pour objectif de caractériser deux principaux éléments de la plateforme, critiques pour parvenir à des niveaux performants d'utilisation de ressources pour un temps de calcul minimal. D'une part, il permet d'évaluer l'implémentation d'opérateurs élémentaires sur carte, en fonction de divers paramètres. Les paramètres considérés sont :

— le format de données (flottant ou entier),
— leur précision (sur 32 ou 64 bits),
— leur facteur de vectorisation (par le formalisme OpenCL : double2, etc.),
— l'organisation des noyaux (déroulage de boucle, manuel ou automatisé).

D'autre part, il permet de déterminer le débit qu'il est possible d'atteindre en fonction de la zone mémoire sélectionnée, à la fois pour le stockage de données et la communication inter-noyaux. Les zones mémoires considérées sont la mémoire globale, la mémoire locale, ainsi que les *channels* (encapsulation de FIFO par OpenCL).

Ce banc de test est composé de tests unitaires, dont les noyaux sont synthétisés et exécutés sur la plateforme matérielle cible, décrits dans le tableau 4.1.

TABLEAU 4.1 – Caractéristiques des tests unitaires réalisés

caractéristiques étudiées	opérateurs unitaires		déroulage de boucle	transfert de données	
numéro du test	01	02	03	04	05
fréquence (MHz)	216	176	232	201	240
taille de *work-group*	10	10	1	1000	1000

Chaque jeu de noyaux porté sur accélérateur est accompagné d'un exécutable sur machine hôté et d'un script s'assurant de la bonne configuration de l'environnement. Ces jeux de tests atteignent des fréquences d'horloge comprises entre 176 MHz et 240 MHz, fréquences paratagées par tous les noyaux au sein d'un même jeu de test. L'utilisateur d'AOC n'a pas de contrôle direct sur la fréquence d'horloge des noyaux et il est difficile d'estimer quels éléments ont un impact sur cette caractéristique. Cette partie logicielle du banc utilise CTest (intégré à CMake) pour l'organisation de la série de tests et le stockage des résultats. Les premiers jeux de test (01 et 02) se concentrent sur la caractérisation d'opérateurs unitaires. Toutes les opérations de bases ne sont pas vérifiées dans cette série de tests : les opérations sélectionnées sont toutes utilisées dans le calcul de volatilité implicite décrit dans le chapitre 3 (multiplication, addition, opérateur puissance, exponentiel). Les formats de données comparés sont les formats en virgule fixe et en virgule flottante, sur 32 et 64 bits. De plus, les opérateurs vectoriels (déclaré en utilisant le type [type]2 dans le code du noyau) sont testés sur un additionneur et un opérateur puissance (sur des entrées double2 et entier 32-bit pour l'exposant de l'opérateur puissance).

Les noyaux regroupant les opérateurs selon leur type (tests unitaires 01 et 02) fournissent une référence d'utilisation de ressources sur la plateforme. Ils ne sont toutefois pas utiles pour déterminer la latence d'opérateurs identiques pour des précisions différentes, l'ensemble des calculs étant bien sûr effectué à chaque exécution d'un noyau. La latence d'opérateurs en fonction de la précision est testée via la réplication de noyaux réalisant une unique opération par exécution. Pour se placer dans un cas d'utilisation viable et permettre d'agréger les accès à la mémoire globale, la taille des *work-groups* considérés a été fixée à 1000. Les données de tests sont générées pseudo-aléatoirement et préalablement envoyées en mémoire globale ; ce temps de transfert (ainsi que le temps de réception des données après calcul) n'est pas comptabilisé lors de l'évaluation des opérateurs unitaires.

De plus, le déroulage de boucles internes au noyau (comparativement à un déroulage via l'utilisation de plusieurs noyaux complets répliqués) est aussi testé (test unitaire 03), que ce soit pour un déroulage via directives de préprocesseur ou manuellement. Selon la figure 4.1, on peut constater qu'il est préférable de laisser l'outil dérouler les boucles de lui-même, ce qui lui permet d'obtenir des performances légèrement supérieures à un déroulage manuel.

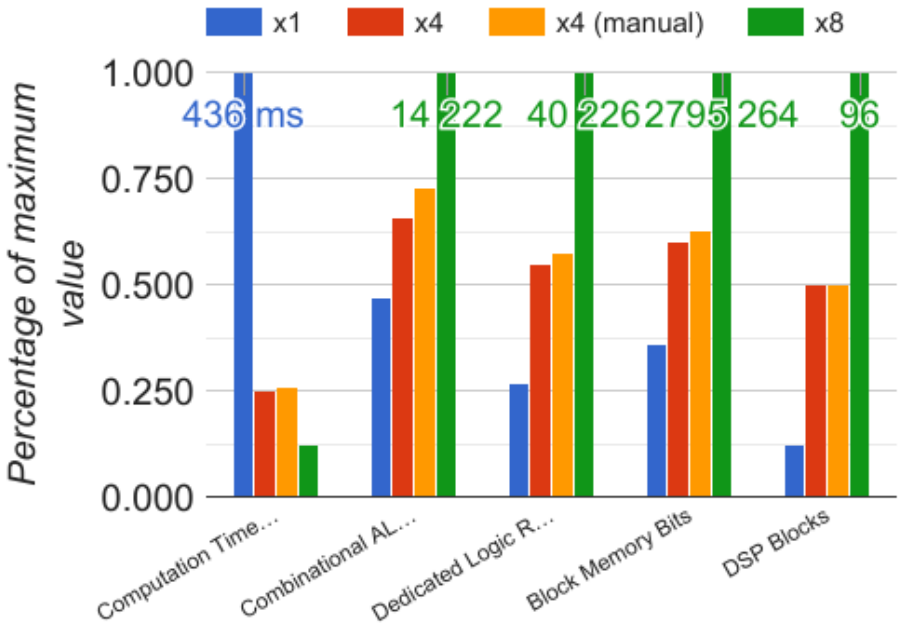

FIGURE 4.1 – Impact du déroulage de boucle l'utilisation de ressources

TABLEAU 4.2 – Organisation générale d'un noyau OpenCL sur FPGA Altera

acl work group dispatcher	ordonnancement des différentes instances d'un noyau
iterator	génération d'identifiants (ex. : *work-items*)
kernel finish detector	génération d'un signal de fin de calcul
local memory interconnect	accès pipeliné ou en *burst* à la mémoire locale
kernel datapath	cœur de calcul tel que défini par l'utilisateur
blocs optionnels	blocs de débogage et d'estimation de performance

4.2.2 Partie de contrôle générique définie par AOC

Dans le cas de l'utilisation d'OpenCL pour le FPGA, une architecture générale est mise en oeuvre qui permet d'intégrer ensuite les noyaux. Bien que peu décrite par Altera, cette architecture est visible en étudiant les fichiers générés après la compilation d'une architecture OpenCL. Elle s'appuie principalement sur un bus propriétaire d'Altera (Avalon) pour la communication entre la couche de communication vers l'hôte (en liaison PCIe) et la couche de gestion des différents noyaux. Chaque noyau est en effet encapsulé dans une couche de gestion, ce qui lui permet à la fois de respecter la norme OpenCL et les caractéristiques physiques du FPGA. Les différents éléments sont repris dans le tableau 4.2.

Le surcoût dû à ces couches de contrôle est minimal par rapport aux ressources disponibles pour les chemins de données des noyaux. Des essais avec différents noyaux de complexités différentes ont montré une taille comparable, quelque soit la taille du chemin de données ou la complexité des

TABLEAU 4.3 – Ressources utilisées par les surcouches sur un exemple de noyau de test

Élément	ALM utilisés	ALUT	Registres
wrapper	437,7	404	702
acl work group dispatcher	139,3	262	196
iterator	155,6	277	208
kernel finish detector	138,5	248	214
blocs de profilage (optionnels)	227,7	456	448

TABLEAU 4.4 – Ressources utilisées par la partition d'interface.

Élément	ALM utilisés	ALUT	Registres	Blocs mémoires (bits)
Partition d'interface	35038 / 234720 (15%)	55838	53395	1 419 856

boucles. Les seules couches dont l'utilisation de ressource sont relativement variables sont :

1. la couche optionnelle de profilage, qui n'est pas présente dans les noyaux hors débogage,
2. le réseau d'interconnexion de la mémoire locale, qui gère les conflits d'accès mémoire aux bloc de RAM et est donc généré au cas par cas pour chaque noyau.

Ces couches générées automatiquement doivent toutefois être prises en compte lors de la conception d'un noyau de calcul. Les valeurs présentées dans le tableau 4.3 proviennent d'un noyau de test de performance de *channels*, et se trouvent proches de la médiane des valeurs observées.

Une manière d'éviter ce cas de figure consiste en une recopie d'une donnée locale pour chaque *work-item* dans des registres, en mémoire privée.

A ces ressources propres à chaque instance physique de noyau s'ajoutent les ressources consommées par une partition figée du *design* sur FPGA, qui permet notamment de charger des noyaux différents sur carte sans reconfiguration de la liaison PCIe entre machine hôte et accélérateur. Cette partition d'interface gère aussi l'accès à la mémoire globale, ici une mémoire externe DDR3, qui est nécessaire lors de l'utilisation du FPGA dans le cas d'OpenCL. Elle intègre également la gestion des horloges. Les ressources consommées par cette partition sont détaillés dans le tableau 4.4. Ces ressources utilisées sont bien plus conséquentes que celles des couches de gestion de noyaux.

4.2.3 Caractérisation de l'utilisation des ressources par les mémoires et les opérateurs élémentaires

On distingue deux types de ressources dans le cas d'OpenCL sur FPGA : la mémoire (selon la nomenclature OpenCL) et les opérateurs. La correspondance entre les éléments de la norme OpenCL et les ressources du FPGA est étudiée dans cette partie.

4.2.3.1 Implémentation de la mémoire

La norme OpenCL définit 3 types de mémoire, avec chacune un champ d'utilisation de plus en plus restreint. La mémoire globale est partagée entre tous les éléments d'un programme. La mémoire locale est partagée entre tous les éléments d'un même *work-group*. La mémoire privée est spécifique à un *work-item*. Dans le cas de l'implémentation d'Altera, il existe un quatrième modèle, le *channel*, qui permet l'utilisation de FIFOs. Conceptuellement, plus la mémoire est restreinte, plus elle est performante (en termes de latence et de débit) mais précieuse en termes de ressource mémoire présente. L'architecture proposée ressemble donc aux architectures mémoires traditionnelles à base de cache.

Afin d'étudier l'implémentation des mémoires dans le cas d'AOC, différents noyaux faisant appel à un seul type de mémoire ont été implémentés, avec à chaque fois une taille connue de mémoire allouée. La synthèse de ces noyaux a ensuite été analysée, afin d'analyser la répartition des ressources mémoires. Il en ressort que la mémoire globale implique une très faible utilisation de RAM interne au FPGA, sans rapport avec la quantité de mémoire allouée. Ceci laisse penser que ce type de mémoire est implémenté dans la mémoire externe imposée dans le cas de l'utilisation d'OpenCL sur FPGA. Les quelques RAM utilisées servent de tampon pour optimiser le transfert des données.

Le noyau basé sur la mémoire locale requiert une quantité équivalente de RAM interne au FPGA. Celle-ci n'est pas intégrée dans la hiérarchie du noyau, et l'accès s'effectue par un réseau d'interconnexion généré en fonction des accès mémoires détectés dans le code source. Ce réseau d'interconnexion peut former un goulot d'étranglement. Ce goulot d'étranglement est observé dans le cas où plusieurs *work-items* font des accès répétés vers une donnée stockée en mémoire locale. Il y a aussi un risque de congestion. Les accès répétés à une donnée stockée en mémoire locale doivent donc être minimisés.

La mémoire privée génère une augmentation équivalente du nombre de registres utilisés dans le noyau. Ceux-ci sont dans la hiérarchie du noyau, et sont directement intégrés dans le chemin de données généré. Une manière d'éviter la congestion de la mémoire locale peut donc consister en une recopie d'une donnée locale pour chaque *work-item* dans des registres, en mémoire privée. Ces résultats sont confirmés expérimentalement dans la section 4.2.4.

4.2.3.2 Implémentation des opérateurs

Les opérateurs flottants sont comparés avec les opérateurs générés à l'aide de FloPoCo [CCA+11]. FloPoCo, développé par l'INRIA, est un outil de génération de chemins de données en précision flottante arbitraire. Il permet aussi de générer des opérateurs discrets, paramétrés pour être adaptés à des cibles particulières. Dans ce cas, il ne génère toutefois pas de signaux de contrôle. La figure 4.2 fournit des indications supplémentaires sur l'utilisation de ressources en fonction du format de données (simple et double flottants et entier 64 bits).

On constate que les opérateurs générés par AOC présentent, dans la majeure partie des cas,

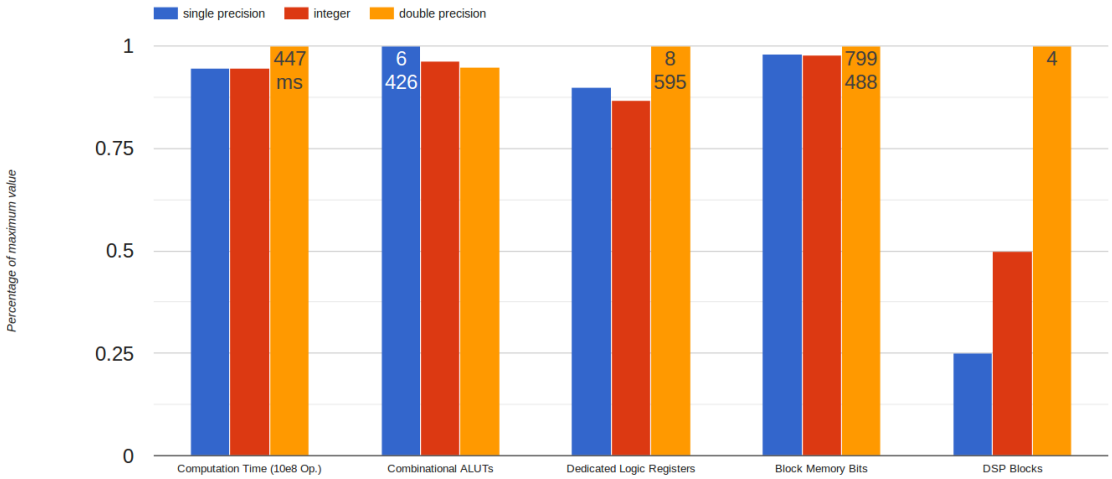

FIGURE 4.2 – Impact de la représentation des données sur l'utilisation de ressources (pour une multiplication)

TABLEAU 4.5 – Ressources utilisées pour la synthèse d'opérateurs double précision

	Additionneur		Puissance		Exponentiel	
	FloPoCo	AOC	FloPoCo	AOC	FloPoCo	AOC
ALUT	644	2172	5110	6304	1138	3966
Registres	1382	3590	10361	10999	2350	5921
Mémoire (bits)	36	100256	458321	486144	85280	71488
DSP 18x18	0	0	38	27	8	10

une utilisation de ressources supérieure aux opérateurs similaires générés par FloPoCo. La mémoire est donnée à titre indicatif, mais n'est pas significative ici. Elle est en effet générée pour le noyau, qui requiert des portions de mémoire pour le contrôle des entrées/sorties, alors que FloPoCo génère des opérateurs pour un chemin de données simple, sans contrôle. Les noyaux générés n'utilisent pas explicitement de mémoire privée, afin d'éviter de fausser les résultats sur l'utilisation des registres. Il est difficile d'évaluer la part du surcoût lié à l'opérateur lui-même et celle liée à l'intégration dans l'architecture OpenCL. Cependant, sauf cas particulier, les ressources consommées restent dans le même ordre de grandeur 4.5.

Les blocs de DSP sont la ressource la plus discriminante, avec une utilisation similaire des autres ressources matérielles (plus de 80% des ressources maximales consommées pour tout format de donnée). FloPoCo a tendance à utiliser les blocs DSP de manière aggressive lors de la création de ses opérateurs, ce qui explique une consommation importante parfois observée, en fonction du paramétrage de l'outil (ici, 38 blocs de DSP sont par exemple utilisés pour l'opérateur puissance, tandis qu'AOC parvient à générer un opérateur équivalent avec une consommation de 27 blocs seulement).

La vectorisation des données présente des avantages en termes de réutilisation de ressources, à

TABLEAU 4.6 – Surcoût en ressource d'opérateurs vectoriels doubles (facteur multiplicatif)

	ALUT	Registres	Mémoire (bits)	Bloc DSP
Additionneur	x1,1	x1,4	x1,0	-
Puissance	x1,7	x1,8	x1,9	x2

taux variable selon le type d'opérateur et le format de donnée comme illustré dans le tableau 4.6. Les seules ressources qui ne sont pas optimisées par cette approche sont les DSP.

4.2.4 Caractérisation du temps d'accès et du débit

En termes de temps de calcul, tous les opérateurs générés par AOC réussissent à fournir un résultat par cycle d'horloge, avec un surcoût temporel minime sur un jeu complet de données. Pour tous les tests, comme déjà vu dans le tableau 4.1, les noyaux atteignent une fréquence d'horloge de plus de 150 MHz (176 MHz pour le test 02 et plus de 200 MHz pour les autres).

Une fois vérifié le comportement d'opérateurs élémentaires de manière isolée, il est possible de caractériser les liens de communication entre *work-items* disponibles dans l'architecture standard générée par AOC. Les tests 04 et 05 mentionnés dans la section 4.2.1 permettent de déterminer les débits atteignables dans des conditions optimales de dimensionnement de données transférées. Ces tests ont pour objectif d'obtenir les débits atteignables en utilisant la mémoire globale, la mémoire locale ou des *channels* lors de transferts de données, tels qu'illustrés figure 4.3.

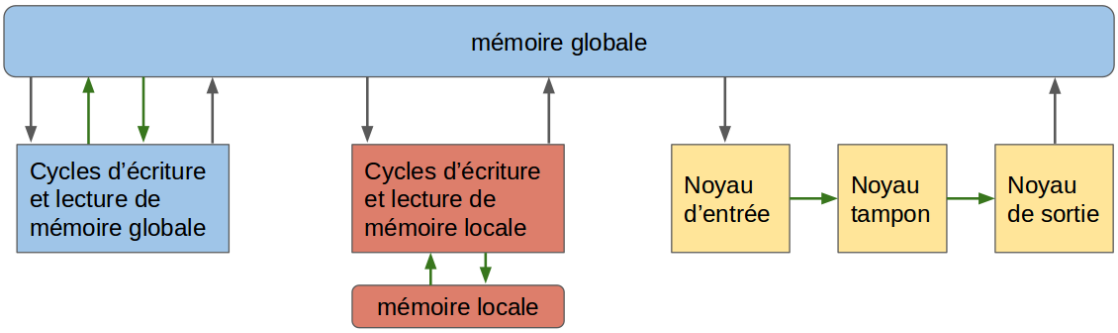

FIGURE 4.3 – Représentation schématique du banc de test de transfert de données

Les noyaux définis dans ces tests sont conçus pour éviter que le synthétiseur ne tronque des portions du flot de données par souci d'optimisation. Les données sont donc consommées dans des opérateurs simples. Comme ces opérateurs peuvent calculer un résultat par cycle d'horloge, ils ont une influence négligeable sur le débit mesuré. Les flèches grises sur la figure 4.3 correspondent à des transferts de données pour initialisation et finalisation de calcul et ne sont pas prises en compte pour le calcul de débit.

Pour l'évaluation de débit en provenance de la mémoire globale, les données sont transférées par packet de 1000 (toutes les données transférées sont sur 64 bits) pour qu'il soit possible de

réaliser un transfert groupé et se placer ainsi dans les meilleures conditions de transfert. De plus, nous addressons des plages de mémoires globales différentes à chaque transfert pour nous assurer qu'aucune optimisation imprévue du noyau ne soit réalisée.

Dans le cas de l'évaluation de la mémoire locale, nous nous plaçons dans un cas où le réseau d'interconnexion de la mémoire locale est simple, c'est-à-dire sans conflit d'accès à la RAM.

Pour l'évaluation des *channels*, nous implémentons trois noyaux consécutifs, de sorte à pouvoir mesurer un cycle d'écriture / lecture sur le noyau central.

Pour obtenir des performances moyennes fiables, ces tests ont été réalisé sur des *work-group* de taille 1000. Les mesures sont effectuées pour des itérations de cycle d'écriture / lecture de 1000 données. Les résultats sont synthétisés dans la courbe 4.4.

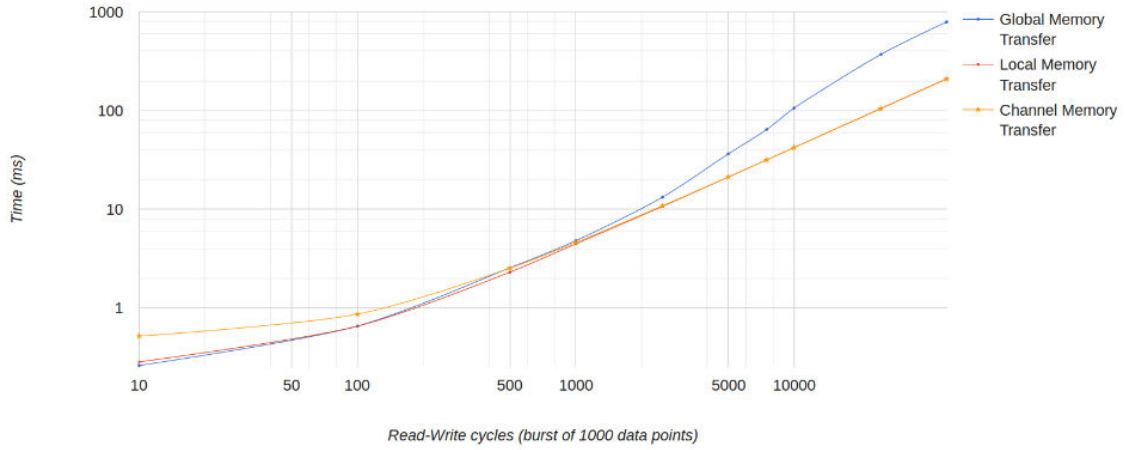

FIGURE 4.4 – Temps d'exécution d'un noyau par cycle de lecture / écriture de donnée, en fonction du type de transfert

4.2.5 Conclusion : caractérisation d'une configuration

Nous avons présenté ici le flot OpenCL d'Altera et son implémentation sur une plateforme matérielle (pour une version 14.1 de Quartus et une carte FPGA Terasic DE5NET). Cette étude a confirmé le fonctionnement de la chaîne de synthèse haut niveau AOC, ainsi que des performances des *designs* générés.

Pour des opérateurs élémentaires (addition, multiplication, ..., exponentielle), nous avons constaté une latence moyenne d'un résultat par cycle d'horloge, pour des débit suffisamment élevés pour compenser les temps de transferts de données. Nous avons caractérisé l'utilisation de ressource de ces éléments, ainsi que le surcoût des différentes couches de contrôle et de communications liées au standard OpenCL.

Nous avons défini les cas d'application de plusieurs types de transferts de données (mémoire globale, mémoire locale, *channel*), en étudiant leur débit respectif pour des tailles de paquets de données de 1000.

Ces résultats nous conforte dans l'utilisation d'AOC comme *back-end* pour un flot d'accélération d'applications financières.

4.3 État de l'art : langages dédiés à un domaine d'application (DSL)

L'utilisation de langages génériques présente des avantages certains pour un développeur. La connaissance d'un tel langage permet d'aborder un éventail exhaustif de domaines d'application, pour un certain prix en termes de complexité de programmation ainsi qu'en termes de performances. L'utilisation d'un langage conçu spécifiquement pour un domaine permet d'envisager un développement plus adapté aux contraintes du domaine en question. De plus, un langage dédié est adapté à l'utilisateur, généralement expert dans le domaine d'application mais pas nécessairement en programmation. Il existe différents types de langages dits Dédiés, ou *Domain Specific Language* (DSL), qui sont généralement classifiés en fonction de leur implémentation. Plusieurs études ont proposé de telles classifications. Dans ce travail, on peut toujours les résumer en deux classes :

— Des langages dits embarqués, soit sous la forme de librairies spécialisées développées dans un langage générique parent, soit sous la forme d'une surcouche masquant le langage parent ;
— Des langages dits *standalone*, conçus entièrement pour le domaine, sans s'appuyer sur un langage générique préexistant.

Concevoir et développer un DSL *standalone* est une tâche complexe. Cet acte requiert à la fois des compétences en techniques de compilation et des connaissances poussées dans le domaine considéré. Une partie de cette complexité peut être évitée en utilisant des DSL embarqués. Le concepteur du DSL peut alors se concentrer sur son domaine d'expertise et produire une solution sous la forme d'une librairie logicielle, sans se soucier de la compilation bas-niveau du code, ainsi que de son optimisation. Cette approche est usuelle, mais le langage hôte peut induire un surcoût en termes de complexité, qui se traduit alors en un manque à gagner sur les performance obtenues (*i.e.* temps d'exécution d'une application).

Nous présentons ci-après un état de l'art sur les DSL qui décrit plusieurs approches réussies d'implémentation de DSL.

4.3.1 Scala et les *frameworks* de génération de DSL

Certaines techniques d'implémentation de DSL se concentrent sur une approche *standalone* en s'appuyant sur des outils de génération de DSL. Ces techniques de méta-programmation peuvent dans certains cas cibler des architectures parallèles pour obtenir des niveaux de performance plus élevés.

Les travaux les plus récents à propos de la génération de DSL reposent en partie sur Scala. Scala est un langage de programmation maintenu par un groupe de recherche à l'EPFL (Lausanne), qui repose sur une machine virtuelle Java (JVM, *Java Virtual Machine*) et permet d'associer une

programmation orientée objet à des fonctions d'ordre supérieur, à savoir des fonctions retournant des fonctions et ayant en entrée des fonctions.

Scala est le point d'entrée de plusieurs projets (reposant les uns sur les autres) réduisant le coût de développement d'un DSL, notamment le Lightweight Modular Staging (LMS) *framework* et Delite.

4.3.1.1 Le *framework Lightweight Modular Staging (LMS)*

LMS propose une approche originale au paradoxe de programmation générale mais performante : un utilisateur de LMS développe un programme qui génère ensuite le programme répondant au problème, de manière optimisée. Cette approche est séduisante pour le développement de DSL embarqués ; d'autres outils de DSL décrits ci-après reposent d'ailleurs sur LMS.

Dans ce cas d'application, l'objectif de LMS est de fournir à un concepteur de DSL un environnement lui permettant de s'épargner certaines tâches de développement qui ne sont pas liées au domaine. Nous décrivons cette division du développement d'un DSL en couches spécifiques à un domaine de compétence ci-après. Le concepteur décrit son DSL sous la forme d'une Représentation Intermédiaire (IR) dans un format imposé par LMS. L'utilisateur du DSL emploie cette IR pour décrire son application. LMS utilise cette représentation et fournit l'environnement de génération de code et de compilation permettant d'exécuter cette application.

Ce développement de générateurs de programmes repose sur un mécanisme de compilation en deux étapes, dit *staging*. Le *staging* dans LMS est implémenté sur les types de données, d'une manière s'apparentant aux *templates* présents en C++. Cet outil est disponible pour le concepteur pour lui fournir une abstraction pour le développement de son DSL. Une manière simple d'appréhender cette étape de *staging* est de considérer un exemple [ORS+13], repris dans le listing 4.2 :

```
1 class Vector[T](val data: Array[T]) {
2   def foreach(f: T => Unit): Unit = {
3     var i = 0; while (i < data.length) { f(data(i)); i += 1 }
4 }}
```

Listing 4.2 – Exemple de *staging* [ORS+13]

Cet exemple présente l'implémentation Scala d'une structure haut-niveau de vecteur, reposant sur une matrice. Le type des valeurs T est générique. Cette structure peut donc être utilisée quelque soit le type des valeurs. La boucle foreach définie ici peut ne pas être optimale en termes de temps d'exécution. Pour résoudre cette difficulté, les auteurs de [ORS+13] propose de remplacer chaque opération de foreach vectoriel par une boucle **while** optimisée. Voici la représentation d'une opération qu'un concepteur pourrait fournir au générateur de code :

```
1 class Vector[T](val data: Rep[Array[T]]) {
2   def foreach(f: Rep[T] => Rep[Unit]): Rep[Unit] = {
3 var i = 0; while (i < data.length) { f(data(i)); i+=1 }}}
```

L'expression Rep[X] présente dans le premier étage (*i.e.* avant la génération de code) produit une expression de type X durant le second étage (*i.e.*, après la génération de code et donc juste avant l'étape finale de compilation). Durant le premier étage, chaque X est évalué et remplacé par une constante. Le *staging* intégré dans LMS s'accompagne de *lifting* pour les expressions étagées : les méthodes sont surchargées pour acccepter des types de données classiques (X) ainsi que des expressions étagées (Rep[X]).

Pour générer du code source, LMS crée dans un premier temps une IR (Représentation Intermédiaire) du programme. Cette IR est organisée en une *sea-of-node*, c'est-à-dire un graphe de dépendence, où les arcs du graphe représentent les dépendences de contrôle. Contrairement à une représentation classique où les instructions individuelles sont regroupées en blocs de base et connectés en un CFG (*Control Flow Graph*), l'organisation en *sea-of-node* n'ordonne pas les instructions séquentiellement. Le seul ordonnancement existant entre instructions est lié à leur dépendance en contrôle et données d'entrée. Cette IR est une fonctionnalité cruciale de LMS, puisqu'elle permet à un développeur de DSL de réutiliser les IR optimisées développées par d'autres et de rajouter ses propres IR au *framework* LMS. LMS est en effet conçu pour être utilisé de la sorte lors du développement d'un nouveau DSL [RO10]. Cela implique toutefois que le développeur de DSL a une connaissance approfondie du langage Scala, ainsi que du *framework* LMS. Ceci impose donc au concepteur une connaissance d'un langage (Scala), ce que l'on cherchait intialement à éviter.

Nithin et al. ont utilisé LMS dans leur Preuve de Concept pour un outil de synthèse matérielle spécifique à un domaine et à faible coût [GNR+13]. Leur méthodologie repose sur certains concepts généraux :

— Leur flot de compilation est divisé en plusieurs étages, chaque étage possédant sa propre IR. Ces différents étages consistent en des niveaux d'optimisation différents. Un étage peut par exemple se focaliser sur les spécificités du domaine visé, des optimisations génériques du code source ou bien encore des optimisations pour rendre le programme plus apte à être porté sur une cible matérielle spécifique.
— Leur outil repose sur des outils de HLS externes pour la dernière étape du flot de compilation, à savoir la synthèse sur cible FPGA. Les deux outils supportés dans [GNR+13] sont FloPoCo et LegUp, outil de HLS *open source* [CCA+11].
— Pour illustrer leur *framework*, le domaine spécifiquement ciblé est le calcul matriciel.

4.3.1.2 Delite

Delite est un *framework* de compilation, auquel s'ajoute un environnement d'exécution. L'ensemble forme une extension plus haut-niveau de LMS. L'objectif de Delite est d'aider à concevoir et développer des DSL qui souhaitent supporter des architectures hautement parallèles. Le *framework* de compilation Delite a été présenté pour la première fois en 2011, dans [BSL+11]. Il fonctionne de manière similaire à la preuve de concept développée par Nithin *et al.* à l'aide de LMS, avec trois niveaux principaux d'IR (spécifique au domaine, optimisations génériques profitant du parallélisme

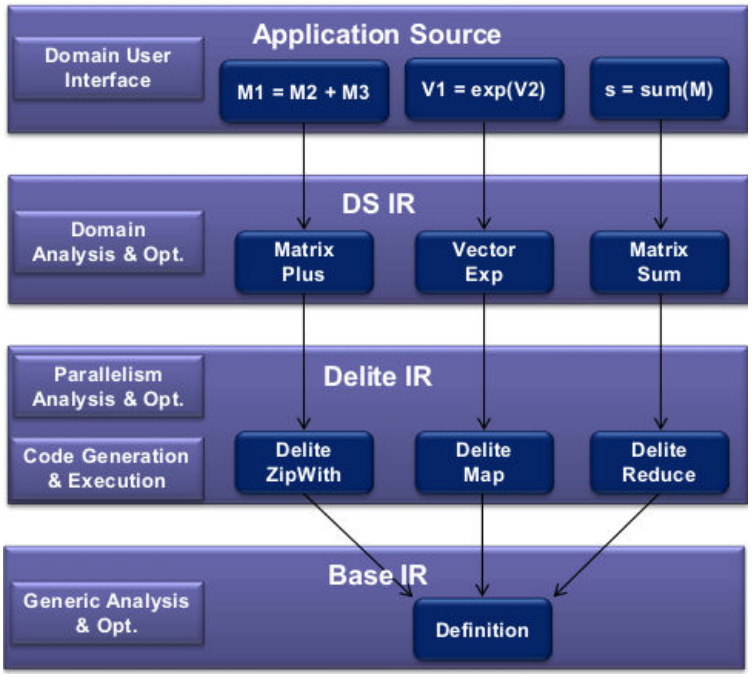

FIGURE 4.5 – Delite - Description des IR d'un DSL [BSL+11]

des cibles, IR basique). La figure 4.5 décrit plus en détail ces différentes descriptions du programme utilisateur en différentes IR, en fonction du niveau d'abstraction.

Pour résumer, Delite fournit un *framework* plus haut niveau que LMS pour des développeurs de DSL, grâce à l'ajout d'IR parallèles qui peuvent être utilisées telles quelles ou étendues par le développeur.

Les concepteurs de Delite décrivent le processus de développement d'un DSL *via* leur *framework* de la manière suivante :

— (action requise) définition des structures de données spécifiques au domaine,
— (action requise) conception d'IR spécifiques au domaine et donc uniques au DSL développé (en appelant les *ops* de Delite appropriées : MapReduce, ZipReduce, etc.),
— (action optionnelle) création de nouvelles *ops* pour étendre Delite, à partir des *ops* existantes (une *ops* correspondant à une opération d'optimisation appliquée au code utilisateur),
— (action optionnelle) développement d'une implémentation optimisée d'un générateur de code (pour cibler des cibles matérielles précédemment non supportées, en contournant le comportement usuel de Delite).

Un concepteur de DSL se contente donc de décrire des IR qui permettent d'identifier des motifs parallélisables spécifiques à son domaine. L'implémentation de ces motifs parallèles est laissée aux experts du calcul concurrent, c'est-à-dire soit les contributeurs au projet Delite, soit les développeurs qui souhaitent définir le code généré pour certaines *ops* sur certaines plateformes matérielles.

Le DSL ainsi créé est embarqué dans le langage Scala, qu'un utilisateur du DSL se doit donc de connaître. Une application développée dans ce DSL est compilée en bytecode JAVA (Scala

s'exécutant dans la Machine Virtuelle Java), qui est ensuite pris en entrée du compilateur de DSL (*i.e.* Delite avec les extensions spécifiques au domaine concerné). Ce compilateur construit l'IR du programme et l'optimise. Les artefacts de compilation en sortie du compilateur sont :

— un graphe d'exécution, dit DEG pour *Delite Execution Graph*, qui contient l'ensemble des *ops* Delite utilisées, leurs dépendances, ainsi que la plateforme ciblée,
— le code généré pour chaque *op*, enveloppé dans des fonctions ou des noyaux que l'environnement d'exécution peut appeler,
— un jeu de structure de données requises par l'application (spécifiques au domaine et définies par le développeur du DSL).

Ces artefacts, associés aux informations connues sur les architectures matérielles ciblées, sont ensuite utilisées durant l'exécution du programme pour l'ordonnancement et la synchronisation, ainsi que pour la génération d'exécutables pour chaque cible matérielle (e.g. CPU, GPU, etc.).

Nithin *et al.* ont appliqué cette approche à la synthèse matérielle [GLN+14]. Leurs travaux rajoutent plusieurs améliorations à Delite pour permettre la synthèse d'application pour cibler des FPGA. Premièrement, ils définissent une liste de *templates* de plateformes matérielles qui est adaptée pour les applications du domaine considéré. Ces *templates* définissent de plus les interfaces entre chaque noyau matériel et le reste de l'architecture (nombre de ports, protocole de communication, etc.). Dans leurs travaux, le *template* décrit contient aussi un processeur *softcore*, un contrôleur de mémoire externe et plusieurs connexions externes (DRAM, UART et interface JTAG).

Le processeur *softcore* leur permet d'implémenter les sections séquentielles de l'application en logiciel, tandis que les sections parallèles sont implémentées en matériel. La séparation du flot de données et du flot de contrôle est réalisée à partir du *Delite Execution Graph*, duquel est extrait un contrôleur, ainsi que les noyaux menant ensuite à l'implémentation d'unités matérielles de calcul. Dans la solution qu'ils présentent, le contrôleur est effectivement implémenté sur le processeur *softcore*, ce qui simplifie l'étape de synthèse. Le contrôleur gère l'ordonnancement des noyaux et libère la mémoire allouée dynamiquement.

L'utilisation de FPGA en cible matérielle est une autre contribution clé de leurs travaux. L'outil de HLS qu'ils emploient est Vivado HLS, développé par Xilinx [Xil14]. Plusieurs optimisations sont réalisées avant de laisser la main à l'outil de synthèse. Une analyse de l'utilisation des données est effectuée pour optimiser l'empreinte sur la mémoire globale. De plus, plusieurs versions des noyaux sont générées pour atteindre un compromis acceptable entre temps d'exécution et utilisation de ressources.

Les temps de calcul atteints par leur cas d'application sont longs, mais leur approche est prometteuse. Sur les différentes applications utilisées pour évaluer leur approche, les deux facteurs principaux de perte de performance sont l'utilisation prédominante de mémoire externe et l'utilisation d'opérateurs en précision flottante. Malgré le fait qu'ils obtiennent des temps similaires sur un GPP typique de serveur de calcul et sur cible Virtex 7 (et donc un facteur d'accélération de 1), il est néanmoins intéressant de constater que la solution portée sur FPGA consomme moins d'énergie et

libère le GPP pour d'autres utilisations.

4.3.2 Utilisation de DSL pour le traitement d'image en HPC

Les défis rencontrés pour le traitement d'image haute performance sont proches de ceux rencontrés en finance. Une demande existe pour des implémentations performantes, malgré la constante évolution du domaine et la diversité des différentes applications de traitement d'images. De plus, les applications de ce domaine sont fortement contraintes en termes de budget énergétique, du fait de la nature embarquée ou tout au moins mobile de la plupart des architectures matérielles ciblées. Il n'est donc pas étonnant de retrouver des DSL pour le HPC dans ce domaine, qui sont historiquement apparus plus tôt que ce qu'on peut trouver en finance.

4.3.2.1 Halide

Halide est un des nombreux DSL spécialisés pour le traitement vidéo et le traitement d'image, sur cible hétérogène. Comme pour l'extrême majorité des travaux de ce domaine, l'architecture ciblée consiste en un GPU accélérant les calculs, associé à un GPP hôte. Les cibles GPU sont en effet tout particulièrement adaptées pour le traitement d'image. Comme les auteurs le décrivent dans [RKBA+13], les algorithmes de traitement d'image (définis sous la forme de *pipelines* d'opérations sur des données d'entrée) allient les défis propres au calcul en treillis et ceux classiquement liés aux traitement en flot de données.

Le concept majeur d'Halide est la séparation de l'algorithme et de son implémentation (*i.e.* des diverses optimisations qui lui sont appliquées). Contrairement à une application classique du domaine, consistant en un programme finement optimisé manuellement et très complexe à appréhender, un programme Halide se résume à un simple algorithme, suivi d'une description de son implémentation. Cette description expose le parallélisme et la localisation des données de l'application. De la sorte, le code Halide est efficace, générique et aisé à maintenir.

Les caractéristiques d'Halide sont intrinsèquement liées à l'objectif de la librairie, à savoir la génération de *pipelines* de traitement d'images. Ces derniers peuvent être considérés comme des calculs de treillis sur des matrices de pixels.

Halide est embarqué dans les langages Python et C++ ; le choix de la version à sélectionner est laissé à la convenance de l'utilisateur. Etant donné la spécialisation d'Halide, les optimisations usuelles de boucles ne sont pas adaptées (fusion de boucles, etc.). Cela rend difficile d'optimiser le ratio du nombre d'opérations réalisé sur un point mémoire. En effet, des points voisins dans le treillis consommés par un opérateur sont susceptibles d'être générés à partir de données communes, de manière similaire au calcul d'arbre binomiaux décrits dans le chapitre 3.

Halide fournit à la place la possibilité pour l'utilisateur de parvenir à différents compromis en termes de localisation de données, de redondance de tâche (ou de stockage de données) et de parallélisme. Ce choix est rendu possible grâce à l'organisation d'un programme codé à l'aide d'Halide, qui est divisé en une description non optimisée d'un algorithme d'une part et son ordonnancement

d'autre part. En fonction des choix réalisés lors de l'ordonnancement de l'algorithme, l'utilisateur a la main libre sur ces compromis d'implémentation.

Les éléments principaux du DSL, tels que décrit dans [Web], sont les suivants :

— *Func* : des fonctions pures définies sur le domaine entier (comme par exemple les fonction spécifiques au domaine dites *blur, brighten*, etc.),
— *Var* : des types abstraits représentant le domaine,
— *Expr* : des expressions algébriques de *Func* et *Var*,
— *Image* : des images en entrées et sorties d'un programme Halide

Un programme Halide consiste donc en un pipeline de *Func*, avec pour entrée et sortie des *Image* et la spécification d'un ordonnancement pour définir l'ordre d'exécution au sein d'un *Func* et entre plusieurs *Func*.

Halide présente toutefois des contraintes fortes en termes de paradigme de programmation pour offrir un niveau de performance élevé. C'est un langage de programmation dit fonctionnel (qui ne présente donc pas d'effets de bord, ni de boucles). Cette décision est justifiée par la facilité d'analyse d'un code écrit dans un tel langage. En termes de *back-end*, un programme codé en Halide est compilé (et non pas interprété), cette étape reposant sur le *framework* de compilation LLVM [LLVb]. Halide propose de plus les services d'un *autotuner* qui infère des spécifications d'ordonnancement haute performance pour l'utilisateur.

Pour résumer, la structure du DSL Halide et les choix d'implémentation qui ont été effectués offrent les caractéristiques suivantes :

— il répond aux problématiques de localisation du stockage de données (tant d'un point de vue spatial que temporel), à l'aide de typage (*Var*), de définition d'entrées et sorties (*Image*) spécifiques au domaine et d'un ordonnancement explicite du calcul,
— il donne à l'utilisateur l'opportunité de gérer l'équilibre entre stockage de données intermédiaires et opérations redondantes,
— il repose sur une représentation d'une application divisée en un algorithme et une spécification d'ordonnancement,
— il repose sur un paradigme de programmation fonctionnelle, contrainte forte sur le programme de l'utilisateur qui simplifie grandement l'optimisation de celui-ci par la chaîne de compilation d'Halide.

4.3.2.2 HIPA^{cc}

HIPA^{cc} [MHT+12] est un autre DSL pour le traitement d'images, qui adopte une approche différente d'Halide. Il est embarqué dans le langage C++ (commun en traitement de signal) et inclut des classes pour la gestion d'images 2D et de filtres. Les noyaux de calculs codés en C++ sont compilés (à l'aide de LLVM [LLVb] et du *front-end* de compilation CLang [LLVa]) et les *Abstract Syntax Tree* (AST) générés sont analysés et optimisés. Les concepteurs d'HIPA^{cc} ont extrait du domaine du traitement d'image des concepts clés. Ceux-ci forment les classes majeures du langage. Celles-ci sont essentiellement :

— la classe *Image*, qui décrit l'élément stockant les pixels d'une image (un pixel peut être représenté par un entier, un nombre flottant, un *RGB*, etc.),
— la classe *Iteration Space*, qui correspond à une région rectangulaire d'intérêt (ROI - *Region Of Interest*), sous-ensemble de l'image de sortie de l'application considérée,
— la classe *Kernel*, qui correspond effectivement au noyau de calcul à implémenter et décrit donc l'algorithme à appliquer à chaque pixel dans l'*Iteration Space*,
— la classe *Accessor* ("Accesseur"), qui décrit les pixels de l'*Image* d'entrée utilisés par un *Kernel*.

Ces éléments clés du DSL ont pour objectif principal de découpler l'ordonnancement du calcul et son exécution de l'accès aux données. La classe *Kernel* permet en effet de spécifier les méthodes d'accès aux données, tandis que les contraintes d'ordonnancement et de partitionnement de calcul sont sous la responsabilité de la classe *Iteration Space*. Les données en elles-mêmes sont encapsulées dans la classe *Image*. Cette division logique d'un problème rejoint la solution proposée par Halide pour atteindre des performances intéressantes, tout en conservant un code applicatif lisible, où les optimisations n'obscurcissent pas son utilité.

Leur travaux récents [RSH+14] se concentrent sur l'extension d'HIPA^{cc} sur d'autres cibles matérielles, à savoir des plateformes hétérogènes Android (SoC basés sur un cœur ARM et un GPU Mali partageant une zone mémoire). Cette flexibilité est possible de par leur utilisation de LLVM et de CLang pour le développement de leur chaîne de compilation. Cette approche leur permet de rajouter des *back-end* à leur outil pour les modèles de programmation qu'ils doivent supporter pour une telle cible (à savoir, Android SDK, Android NDK, Renderscript, Filterscript).

4.3.3 *Forward Financial Framework*

Le *Forward Financial Framework* (F^3) se présente comme un *framework* Python programmable pour certains types de calcul financiers [Ing]. F^3 est disponible sur github [Ing]. Comme son nom l'indique, il se concentre sur l'évaluation de produits dérivés dit *forward*, qui inclut notamment les options européennes, les options à barrière ainsi que les options asiatiques. Ce *framework* supporte une modélisation de sous-jacents suivant le modèle Black et Scholes, ou le modèle d'Heston. F^3 implémente une unique méthode de calcul pour l'évaluation de dérivés *forward* : la méthode de Monte Carlo. Ce choix s'explique aisément de par sa facilité d'implémentation sur plusieurs types d'accélérateur, ainsi que la possibilité d'obtenir aisément un jeu de paramètres optimal pour chaque architecture ciblée.

En termes de structure, F^3 est composé de plusieurs éléments qui correspondent à des concepts spécifiques à l'évaluation de produits dérivés. Pour calculer un produit dérivé, un utilisateur doit :
— créer un élément dit *Underlying*, c'est-à-dire un produit sous-jacent,
— créer un *Derivative* (produit dérivé) et ainsi spécifier quel type d'option il souhaite calculer,
— associer ce produit dérivé à un *Solver*, le moteur de calcul de Monte Carlo étant le seul disponible actuellement,
— lancer le calcul sur une platforme parmi celles disponibles (GPP, cibles compatibles OpenCL :

GPU et FPGA, cible FPGA selon le flot Maxeler FPGA).

Un utilisateur peut donc évaluer différents types de forward à l'aide de F^3 et bénéficier de certaines optimisation liées au domaine financier. Notamment, il peut configurer l'exécution du calcul pour parvenir à un compromis en termes de temps d'exécution et d'intervalle de confiance sur le résultat et, dans le cas où plusieurs produits dérivés reposent sur le même sous-jacent, il peut choisir de dupliquer les données d'entrée ou de les fusionner pour influer sur le degré de parallélisme.

Toutefois, ce framework peut difficilement être qualifié de DSL, puisqu'il ne présente pas de fonctionnalités de programmation. Si un utilisateur souhaite développer son propre modèle, ou le porter sur une nouvelle plateforme, il doit suivre un flot de développement classique puis intégrer manuellement sa solution au *framework* F^3 pour enrichir celui-ci.

4.3.4 Conclusion sur l'état de l'art

Nous avons présenté plusieurs approches de DSL dans cette section. L'implémentation d'un DSL demande des compétences à la fois dans le domaine d'utilisation, mais aussi en programmation, compilation, voire des compétences poussées en électronique numérique ou GPGPU pour l'accélération matérielle. Delite propose une approche de génération de DSL qui permet par construction d'isoler les compétences nécessaires pour obtenir un DSL accélérant les applications d'un domaine particulier sur des cibles matérielles. Leur aproche passe par une optimisation de la représentation intermédiaire d'un programme via des niveaux successifs d'optimisations, du plus spécifique au domaine jusqu'au plus spécifique à la plateforme cible.

Les approches DSL dans le domaine financier sont peu nombreuses. F^3 est mentionné comme une expérience pour prouver l'intérêt d'approches spécifiques à un domaine pour accélération matérielle, même si leur flot présente peu de degrés de libertés pour un utilisateur.

Nous avons donc illustré deux autres exemples de DSL dans un domaine distinct, à savoir le traitement d'images en HPC. De ces exemples, nous pouvons extraire des caractéristiques générales d'un DSL :

— Il est nécessaire de séparer clairement les données du reste du calcul, en termes d'encapsulation et d'accès. Cette séparation permet effectivement de raisonner sur les spécificités du domaine et d'implémenter des optimisations liées au domaine.
— L'algorithme à implémenter doit être séparé des optimisations qui lui sont appliquées, que se soit de manière explicite dans le code développé par l'utilisateur du DSL, ou de manière implicite par intégration d'étages d'optimisations dans l'outil développé.
— Dans tous les cas, un noyau parallélisable doit être isolé du reste de l'algorithme pour accélération matérielle. Cette responsabilité peut être entièrement reléguée à l'utilisateur, ou être endossée partiellement par le DSL.

4.4 Flot de développement OpenCL pour FPGA dédié à la finance

4.4.1 Introduction

Plus qu'un simple moyen d'accès à une plateforme hétérogène, un *framework* de calcul d'options doit supporter les éléments suivants :

— une plateforme physique d'accélération, calibrée pour l'hôte utilisé (que la plateforme soit partagée sur un serveur ou à utilisateur unique),
— le programme source de l'utilisateur, qui interagit avec les noyaux de calculs portés pour accélération,
— les noyaux de calculs extraits du code utilisateur, syntétisables, testables et intégrables sur une plateforme connue,
— une interface de programmation pour l'utilisateur, que ce soit pour le développement ainsi que le test, l'estimation de performance avant synthèse (parallélisation et utilisation de ressources),
— un environnement d'exécution du programme, permettant l'ordonnancement du calcul sur un ou plusieurs accélérateurs.

Nous nous concentrons plus particulièrement dans cette section à l'interface de programmation de l'utilisateur (*i.e.* un DSL et une chaîne de compilation associée), après isolation des noyaux de calculs par l'utilisateur. Les étapes précédentes, nécéssaires pour la réalisation d'un *framework* complet, sont présentées de manière succinte, mais ne feront pas l'objet d'une optimisation particulière. Classiquement, la génération d'opérateurs se heurte au problème de la granularité des calculs : plus le grain est gros, plus on peut proposer des solutions performantes, mais moins le résultat est flexible. A l'inverse, une approche à grain fin permet d'envisager des solutions très flexibles, mais souvent moins performantes. En d'autres termes, une approche à grain fin correspond à une série d'optimisations locales, optimales séparément mais formant une solution moins performante qu'une optimisation globale.

Afin de trouver une solution à ce problème, l'approche proposée ici s'appuie sur la forme classique des algorithmes du domaine de la finance. En extrayant certaines caractéristiques communes de ces algorithmes, nous proposons un langage ayant un impact minimum sur sa prise en main par un développeur, tout en permettant d'obtenir des performances intéressantes. Dans un premier temps, nous décrivons cette représentation, avant d'en tirer les différentes passes d'analyse et de transformation du code permettant d'obtenir automatiquement une implémentation efficace de l'algorithme. Une présentation de l'implémentation du *framework* et de son intégration dans le flot de développement OpenCL d'Altera est ensuite réalisée.

4.4.2 Représentation du domaine d'application

Les approches décrites dans la section 4.3 (et particulièrement sur l'approche Delite vue dans la section 4.3.1.2) ont montré certains points nécessaires pour assurer la conception d'un DSL efficace. Notamment, la séparation des données (en termes d'encapsulation et d'accès) par rapport à l'algorithme de calcul est une étape clé pour pouvoir optimiser l'algorithme et distinguer le flux de données du coeur de calcul. Cette séparation explicite facilite grandement la parallélisation du calcul, que ce soit en réplication de coeurs de calculs, ou en copie d'éléments mémoires, en fonction des ressources disponibles. De plus, l'interface de programmation doit permettre d'exprimer pleinement le problème de l'utilisateur, tel que nous l'avons déjà défini, c'est-à-dire comme un jeu *(problème, modèle mathématique, solution d'évaluation)*, par exemple {Évaluation d'options américaines, modèle de Black et Scholes, Implémentation via un Arbre Binomial}.

Dans une première approche pour définir un DSL en s'appuyant sur QuantLib et sa définition logicielle du domaine, approche présentée dans le chapitre 2, le découpage des classes élémentaires représentant l'algorithme ne permettait pas cette distinction entre les données et le contrôle. Ses éléments de base (*Instrument*, *Pricing Engine*) présentaient trop de co-dépendances. Par exemple, la classe *Instrument*, qui permet de définir un sous-jacent, comprend :

— des données (paramètres),
— des interfaces d'entrées et sortie vers des cotations (*quote*, i.e. des données du marché), mises à jour de manière paresseuse,
— des portions d'algorithmes correspondant à la mise à jour de ce sous-jacent, qui attend d'être appelé par un Pricing Engine (moteur d'évaluation).

Lorsque l'on tente d'extraire des opérateurs élémentaires avec un tel découpage, la granularité est trop fine pour obtenir des temps de calcul viables, ce qui explique d'ailleurs les faibles performances obtenues dans le chapitre 2. Une portion de code isolée pour conversion en un noyau de calcul matériel doit donc être définie à un niveau d'abstraction plus bas.

Si l'on considère les méthodes d'implémentation décrite dans le chapitre 1, on peut distinguer des caractéristiques exploitables pour définir un niveau de granularité à rechercher dans du code utilisateur.

Premièrement, plusieurs des méthodes décrites présentent une double discrétisation : en temps et par rapport à des ratios (bornés) de valeur de sous-jacent. On considère dans le raisonnement qui suit l'optimisation du calcul d'une unique option. Ce type d'optimisation est en effet bénéfique quelle que soit la charge de calcul considérée, contrairement aux optimisations demandant un nombre d'options à calculer important avant d'être performant.

C'est notamment le cas pour le calcul d'option à l'aide d'arbres (binomiaux ou trinomiaux), l'utilisation de la méthode de Monte Carlo ou de méthodes des différences finies (toute méthode confondue). Le grain défini dans ce cas correspond à une itération à une coordonnée (T, S), comportant la mise à jour du prix du sous-jacent par rapport à l'itération précédente (à $T-1$ pour un algorithme *forward* comme Monte Carlo, ou $T+1$ pour un algorithme *backward*, comme les arbres binomiaux ou les méthodes des différences finies), ainsi que la mise à jour du prix de l'option

en cours d'évaluation.

De manière générale, il est intéressant de stocker en mémoire les résultats d'un pas de discrétisation temporel complet. C'est le cas lorsque des calculs sont à réaliser entre deux pas temporels (comme par exemple le calcul de versement de dividendes au cours de la vie d'une option), ou pour le calcul d'une condition quelconque pouvant être atteinte et conduisant à un arrêt du calcul (options à barrière, précision de calcul suffisante atteinte, etc.).

Ces cas particuliers sont suffisamment communs pour orienter les choix d'implémentation de parallélisme que nous avons à notre disposition. Si on se place dans le contexte d'une génération pour OpenCL, un tropisme des boucles sur les valeurs discrètes de sous-jacent vers une implémentation sous la forme d'un *work-group*, avec un *work-item* pour chaque pas de discrétisation (ou chaque chemin de Monte Carlo) paraît adéquat. Cette correspondance relègue le parcours en temps à une boucle interne des noyaux, ce qui permet de gérer une grande partie des cas aux frontières (initialisation et finalisation de calcul), ainsi que certain cas particuliers (panier d'options corrélées ou calcul de barrière entre deux pas temporels par exemple).

L'utilisation d'une plateforme d'accélération présente un coût en latence dû aux temps de communication entre hôte et accélérateur. Cette latence est une contrainte matérielle des machines actuelles, et ne peut malheureusement pas être corrigée sans passer par une modification des machines, dépendantes des constructeurs (Xilinx ou Altera par exemple) et non plus des concepteurs. Cette latence s'applique à chaque requête : on peut donc diminuer son impact en calculant plusieurs options en même temps, suffisament pour que le gain en performance contrebalance la latence. Certaines solutions matérielles commencent à voir le jour afin de diminuer cette latence, il sera intéressant d'évaluer l'impact de ces nouvelles architectures lorsqu'elles seront disponibles.

Dans un second temps, nous devons donc considérer un cas d'utilisation où un vecteur de plusieurs jeux de *(problème, modèle mathématique, solution d'évaluation)* devraient être supportés dans un laps de temps court par une plateforme d'accélération. L'évaluation d'un unique type d'option, pour une seule modélisation de sous-jacent est en effet un cas d'utilisation peu probable. L'implémentation de différents types de modèles (par exemple : le modèle de Black et Scholes et le modèle d'Heston) présentent des différences algorithmiques importantes, qui rendent difficile leur intégration dans un unique noyau. De même, plusieurs méthodes d'implémentations (Monte Carlo, arbres binomiaux, etc.) n'ont pas assez de recoupement pour justifier leur présence dans un unique noyau, ce qui nécessiterait le partage de ressources communes et rendrait le placement sur FPGA complexe. En fonction du modèle de sous-jacent et de la méthode d'implémentation, il est toutefois envisageable de calculer plusieurs types d'options au sein d'un même noyau, en réutilisant la majeure partie du circuit de contrôle et de gestion de données.

A partir de cette analyse, il est possible de commencer à construire un *framework* adapté au domaine, permettant de rechercher des *patterns* connus dans le code utilisateur, afin d'en extraire des opérations de base qu'il est ensuite possible d'accélérer.

4.4.3 Première étape : analyse du code

La première étape lors de la génération d'une architecture matérielle vise à rechercher dans le programme utilisateur des noyaux que l'on peut potentiellement accélérer. Comme décrit dans la section précédente, un algorithme dans le domaine financier possède certaines caractéristiques qui facilitent la recherche de ces noyaux. L'analyse de code va donc chercher à extraire ces noyaux si possible, ou à proposer des approches alternatives si les caractéristiques ne sont pas retrouvées (par exemple, solution purement logicielle, permettant de garantir la fonctionnalité, ou approche purement matérielle, en créant un unique noyau au détriment des performances). Dans la version actuelle de l'outil, l'accent n'est pas mis sur l'extraction des noyaux, ceux-ci sont donc marqués par l'utilisateur à l'aide de *pragmas*. Dans une version ultérieure, une recherche de noyaux plus judicieux, permettant de soulager l'utilisateur et d'optimiser cette extraction est envisagée. Les passes d'analyse qui permettront d'extraire les noyaux répondent à plusieurs besoins. Elles doivent notamment :

— réaliser des vérifications usuelles sur le programme (syntaxe, absence de code non synthétisable, noyau bien défini sans dépendance externe, etc.),
— guider les passes d'optimisations qui suivent,
— fournir une première estimation grossière des ressources consommées, en fonction des opérateurs élémentaires utilisés,
— fournir une première estimation de la latence du noyau.

Ces étapes demandent de générer un graphe d'exécution du programme utilisateur. LLVM fournit des outils de génération de *Control Flow Graph* (CFG) sur des fonctions, via l'outil llvm-opt. Un CFG n'est toutefois pas suffisant pour notre cas d'application : les dépendances mémoires doivent en effet être explicitées, notamment en termes de nombre d'accès consécutifs, pour déterminer les zones mémoires les plus adaptées. Les données fréquemment utilisées en lecture par des *work-items*, sans mise à jour à synchroniser au sein d'un *work-group*, sont à copier en mémoire privée (registres sur FPGA). Il est aussi possible de dupliquer des calculs de complexité faible pour mettre à jour des copies locales de données. Cette optimisation est illustrée dans le chapitre 3, où la valeur du sous-jacent S est mise à jour localement dans chaque *work-item*. L'utilisation de mémoire locale dans ce cas présente en effet des inconvénients lorsque les données d'une RAM sont utilisées plusieurs fois par différents *work-items*, lors d'une itération d'un *work-group* (ou d'une boucle interne). Le réseau d'interconnexion en entrée des RAM est alors complexe et peut former un goulot d'étranglement (voir Section 4.2.3).

Les dépendances mémoires peuvent être extraites à l'aide de passes usuelles de LLVM, notamment à l'aide de passes d'analyse d'alias, comme par exemple `--a-eval` et des passes d'analyse de dépendance mémoire, comme `-da` (*Dependence Analysis*) et `-memdep` (*Memory Dependence Analysis*). Cette information peut être représentée sous la forme d'un CFG enrichi, tel qu'un Control Memory Data Flow Graph [Koc10], qui est adapté à un portage matériel, qui gère les conflits mémoires via l'utilisation directe de multiplexeurs.

Une fois le graphe généré, il est possible de déterminer le nombre minimal d'opérateurs sur le

chemin de données du noyau à générer. A l'aide de l'étude de la plateforme réalisée dans la section 4.2, on fournit une première approximation de l'utilisation en ressources du noyau à implémenter. Cette estimation permet de guider les phases d'optimisations qui suivent, en termes de copie en mémoire de résultats intermédiaires (en mémoire locale ou privée), de déroulage de boucle, de réplication de noyau et de vectorisation des opérateurs. L'estimation du chemin de donnée avant optimisation permet aussi de déterminer une borne haute sur la latence que le noyau peut atteindre, en fonction du nombre d'itérations de boucle interne nécessaire pour la complétion d'un calcul. Ces résultats sont à stocker dans une base de donnée, distincte de l'IR du noyau pour en diminuer la complexité et visible pour l'utilisateur. L'affichage d'indicateurs de performance le plus tôt possible dans le flot de compilation est une valeur ajoutée pour l'utilisateur, au vu des temps de synthèse longs qui suivent. Cette base de donnée doit présenter les informations minimales suivantes, pour chaque noyau :

- le type d'option(s) et le modèle de sous-jacent accepté, tel que défini par l'utilisateur,
- les caractéristiques de parallélisations retenues (aucune durant l'analyse),
- un numéro de version du coeur de calcul, défini à partir de la version de notre couche de compilation ainsi que du *back-end*,
- une définition des entrées (interface vers une mémoire globale, ou vers un *channel* et précision des données),
- une estimation des ressources matérielles utilisées par le noyau,
- une estimation de la latence du noyau,
- un drapeau indiquant si le noyau a déjà été synthétisé ou non.

Les entrées et sorties d'un noyau sont à spécifier par l'utilisateur ; ces spécifications doivent notamment expliciter le modèle du sous-jacent, le type d'option à évaluer et la précision du calcul (flottant simple ou double précision). De même, les opérateurs élémentaires (addition, exponentielle, maximum, etc.) dont l'implémentation matérielle est connue sont liés à des métadonnées présentes dans l'IR, selon le même modèle qu'un noyau.

4.4.4 Deuxième étape : transformation du code

Une fois l'étape d'analyse du programe utilisateur achevée, nous avons donc accès à l'IR du programme, les constructions spécifiques au domaine isolées. Un CFG enrichi avec les dépendances mémoire du programme et une base de données viennent compléter cette caractérisation du programme. Il est alors possible d'effectuer des passes d'optimisations sur l'IR, jusqu'à obtenir une version adaptée à la plateforme cible.

Les passes de transformation d'IR sont ordonnancées des plus générales (portée sur la fonction) vers les plus spécifiques (portée limitée à un *basic block*). Cela permet en effet de fixer les transformations les plus conséquentes en amont du flot et d'éviter des itérations inutiles de passes qui seraient invalidées ensuite.

Les premières transformations envisagées consistent en la scission d'un programme utilisateur en plusieurs noyaux, que ce soit pour la réutilisation de portions du calcul, ou pour une meilleure

gestion de flots de contrôle complexes (tels qu'illustrés dans le cadre de l'évaluation de volatilité implicite dans le chapitre 3). Ces transformations demandent de plus de gérer la création de de *channels* entre noyaux pour assurer un transfert de données à latence faible. Pour des transferts importants de données entre deux noyaux, nous avons vu dans la section 4.2.4 qu'une copie en mémoire globale était plus judicieuse.

Ensuite, l'organisation du noyau en termes de taille de *work-group* et correspondance avec l'algorithme à porter font l'objet d'une passe de transformation. En fonction des résultats des passes d'analyse et des indications de l'utilisateur, cette étape cherche à extraire la boucle correspondant à la discrétisation selon la valeur du sous-jacent S et à la traduire en un *work-group* de taille fixe. Lors de chaque passe de transformation, une nouvelle version de l'IR est générée, prenant en compte ces transformations et modifiant les instructions pour les adapter aux conventions suivies par AOC. Dans ce cas, cela consiste en la génération de métadonnées décrivant le noyau, ainsi que la déclaration de la fonction correspondante. De même, une mise-à-jour de la base de donnée et du graphe de contrôle enrichi est réalisée.

La passe de transformation suivante se charge de déterminer quelle type de mémoire utiliser pour stocker et transférer des résultats intermédiaires. Les entrées et sorties du noyau sont nécessairement stockées en mémoire globale, après transfert vers ou en provenance de l'hôte. Pour toutefois minimiser les accès à la mémoire globale, dont la latence est élevée, une copie locale ou privée de toute entrée est réalisée durant l'initialisation des calculs.

Les données temporaires générées à chaque pas temporel sont préférentiellement stockées en mémoire locale, avec un point de synchronisation inséré après chaque pas temporel et après initialisation et finalisation du calcul. L'utilisation de mémoire privée est réservée au stockage de données interne à un *work-item*, ou pour soulager le réseau d'interconnexion de la mémoire locale en utilisant des copies mémoires privées. Ces copies permettent de réduire le nombre de conflits mémoire, ce qui réduit la complexité du réseau d'interconnexion vers la RAM.

Tout transfert entre deux noyaux est exprimé par défaut via des *channels*. Dans le cas où le dimensionnement de ces *channel* n'est pas réussi ou dans le cas ou il existe un rebouclage, les transferts sont effectués explicitement par l'hôte, via l'utilisation de mémoire globale. Ce comportement dégénéré a peu de chance de fournir des temps de calcul acceptables, mais permet tout de même d'obtenir une implémentation fonctionnelle.

Le bon comportement de l'ensemble est assuré tant que des points de synchronisation sont placés après itération sur un pas de temps (lors de conflits mémoires entre *work-items*). Les conflits possible sont détectés à l'aide du graphe de contrôle enrichi généré durant l'étape d'analyse.

Une fois les zones mémoires définies, chaque opérateur est remplacé par une version compatible avec AOC, c'est-à-dire dont la séquence d'instruction dans l'IR est reconnue par le flot AOC.

Enfin, l'IR suit une série de passes d'optimisations, en fonction du budget en ressources que le noyau peut utiliser, en fonction des caractéristiques de la plateforme matérielle ciblée et des autres noyaux présents. Ces opérations sont similaires à celles illustrées dans le chapitre 4.2, à savoir la réplication de noyau, le déroulage de boucle internes, la vectorisation de certains opérateurs matériels,

etc. En fonction de l'utilisation faites des mémoires locales et des dépendances existantes entre *work-items*, le déroulage de boucle et la vectorisation d'opérateur ne sont pas des transformations recommandées ; la réplication de noyaux est donc favorisée si les ressources le permettent.

L'IR subit ensuite une dernière passe de transformation pour générer les dernières métadonnées qui la rend compatible avec la suite du flot AOC.

4.4.5 Implémentation du flot

Nous avons présenté un flot de compilation reposant sur LLVM et donnant un accès à de l'accélération matérielle pour le calcul d'option. Nous présentons ci-après les travaux déjà réalisés sur ce flot.

Nous avons focalisé nos efforts sur les couches basses de l'outil de compilation, c'est-à-dire les points d'interactions avec la chaîne de synthèse haut-niveau d'AOC.

En l'état, deux passes sont disponibles, pour générer les IR nécessaires à la création d'un noyau et les métadonnées associées et pour générer des mémoires locales.

Ces passes reposent bien sûr sur LLVM. La passe de création de noyau hérite de `llvm::ModulePass`. Cette passe :

- génère les métadonnées nécessaires pour AOC (*cf.* Listing 4.1 présenté dans la section 4.1.2 pour une illustration de celles-ci),
- détecte les fonctions présentes dans l'IR d'entrée qui doivent être transformées en noyaux,
- en l'état, cette passe extrait aussi des données statistiques du noyau, concernant le type d'accès aux différentes mémoires présentes dans le noyau.

Durant l'exécution de cette passe, deux conteneurs `std::map` sont créés et remplis respectivement avec des données sur les noyaux isolés et les zones mémoires détectées. Ces conteneurs incluent un `string` identifiant l'élément stocké, ainsi qu'un pointeur vers une classe *ProcessingElement* (pour les noyaux) et *MemoryElement* (pour les mémoires). Cette approche a pour but de faciliter la création de la base de données, pour estimation de performances (ressources utilisées, temps de calcul estimé). Cette passe itère sur le Module considéré (i.e. l'ensemble de l'IR d'entrée), ainsi sur les Fonctions présentes, pour détecter les noyaux. Au sein de cette double boucle, un dernier niveau d'itération sur les instructions est réalisé pour étudier les zones mémoires.

Comme perspective, ces fonctionnalités devront être extraites pour les replacer à leur place dans le flot d'analyse et d'optimisation complet, mais leur présence dans cette passe lui permet pour l'instant d'être relativement autonome.

La passe de génération de mémoire locale hérite elle aussi de `llvm::ModulePass`. Cette portée est nécessaire du fait que l'utilisation de mémoires locales nécessite de les déclarer en tant que variables globales. Pour le moment, cette passe ne supporte que la création de mémoire locale sous la forme de matrice, ce qui permet effectivement de couvrir l'ensemble des cas d'utilisation de mémoire locale (avec toutefois une contrainte en termes de déclaration).

En termes de complexité, la passe de conversion de mémoire locale itère sur les *Modules*, les *Fonctions* et les *BasicBlocks*. La passe détecte les instructions d'allocation de mémoire, vérifie que

les métadonnées de l'instruction la marque comme la création de mémoire locale puis transforme l'IR pour qu'elle soit conforme aux attentes d'AOC. Cette passe s'accompagne d'une passe classique d'élimination de code inutile (passe -dce, *DeadCodeEliminationPass*) pour supprimer les instructions restante de l'instruction d'allocation précédente.

4.5 Conclusion

Nous proposons une solution spécifique au domaine de la finance quantitative aux besoins en apparence opposés de l'accélération de calcul sur architecture hétérogène. Aisance de programmation, contraintes fortes sur les temps de calcul et solutions évolutives demandent non seulement d'employer une approche haut-niveau, mais aussi des sacrifices en termes de généralisation de l'approche. L'étude approfondie de la plateforme matérielle permet de fournir des statistiques sur les opérateurs élémentaires, les éléments mémoires utilisés et les liens de communications. Ces statistiques sont clés pour fournir à l'utilisateur et au reste de la chaîne de compilation les indices nécessaires pour déterminer les options de parallélismes les plus prometteuses. Elles donnent une estimation de l'occupation en ressource de noyaux de calculs portés sur matériel et du meilleur dimensionnement en termes de taille de *work-groups* et en termes de noyaux utilisables en parallèle (sans reconfiguration). Elles confirment de plus l'intérêt de l'utilisation de mémoire locale pour le transfert de données, ainsi que la mémoire globale pour des grands volumes de données.

L'état de l'art sur les DSL (majoritairement hors domaine financier), ainsi que les conlusions tirées de l'intégration à QuantLib (voir chapitre 2) permet d'extraire les abstractions du domaine qui peuvent être retranscrites sur matériel. Nous avons défini un étage de compilation intermédiaire acceptant du code haut-niveau en entrée et dont le produit est accepté en entrée de la chaîne de synthèse d'Altera OpenCL. Cette surcouche repose sur LLVM pour définir une série de passes permettant d'extraire du noyau défini par l'utilisateur les données nécessaires à son passage sur matériel. Ces passes permettent de définir un CFG enrichi, graphe de dépendance en données et en contrôle du noyau, ce qui guide les passes d'optimisations suivantes (parallélisation via portage sur *work-group*, déroulage de boucle pour un pas de temps fixe, etc.). La traduction des opérations élémentaires réalisées dans le noyau est ensuite effectuée à l'aide des données obtenues lors de l'étude de la plateforme. La génération de métadonnées spécifiques au flot AOC est la dernière étape avant insertion dans le flot d'Altera pour génération de bitstream pour accélération.

Conclusion et perspectives

Dans le cadre de cette thèse, nous avons étudié différentes approches permettant d'augmenter l'accessibilité de l'accélération de calcul, en nous concentrons principalement sur l'accélération matérielle. Toutes ces approches ont un point commun : elles utilisent OpenCL comme représentation de la cible finale, au lieu de travailler directement sur cette cible, afin de diminuer les temps de développements et de permettre une portabilité de ces travaux.

Le contexte de nos travaux a été présenté dans le chapitre 1. Nous avons défini le domaine d'application, à savoir le calcul d'options financières et avons défini plusieurs modèles d'évaluation de celles-ci, du plus simple (modèle de Black et Scholes) à des modèles plus complexes et prenant en compte des caractéristiques supplémentaires du marché (modèles à saut, modèles à volatilité stochastique). Nous avons ensuite introduit les plateformes de calculs susceptibles d'être utilisées pour l'implémentation des modèles de calcul d'options, dans un contexte de calcul haute performance. Plus précisément, les architectures de GPP, GPU et FPGA ont été décrites. Nous avons présenté l'intérêt de solutions matérielles hétérogènes pour diminuer les temps de calcul (à un budget énergétique identique ou inférieur), ainsi qu'OpenCL, standard de programmation pour une telle infrastructure.

Pour résoudre un problème financier concret, comme par exemple l'évaluation d'un type d'option particulier, pour une précision fixée, le choix d'un modèle n'est pas suffisant. Il faut aussi considérer son implémentation sur matériel, ce qui demande de choisir une méthode d'implémentation parmi l'état de l'art. Les méthodes d'implémentations les plus communes ont été décrite dans ce chapitre et incluent la méthode de Monte Carlo, l'utilisation d'arbres binomiaux ou trinomiaux, plusieurs méthodes des différences finies ainsi que le calcul numérique d'intégral.

Les besoins de l'industrie en termes de puissance de calcul se confrontent aux limitations des couches matérielles : les GPP traditionnellement utilisés atteignent leurs limites, tandis que la complexité en programmation des solutions à base de GPU et FPGA est pénalisante et ne leur permet pas d'atteindre des temps de mise en production compétitifs. Nous proposons donc une approche spécifique au domaine financier pour ouvrir l'accès en programmabilité à des plateformes hétérogènes. L'objectif est de fournir une solution permettant de développer rapidement des solutions de calcul d'options, qui atteignent des niveaux de performances élevés en termes d'efficacité énergétique et de rapidité.

Dans le chapitre 2, nous avons présenté une couche logicielle permettant d'intégrer des noyaux

matériels dans une librairie logicielle, de manière transparente pour un utilisateur de la librairie - une fois la plateforme matérielle installée sur la machine hôte. OpenCL est employé en tant que couche bas-niveau pour gérer les cibles matérielles et leur comportement durant l'exécution. Nous avons toutefois montré que cette démarche nécessite de réaliser des compromis coûteux. L'architecture de QuantLib, pensée pour être une librairie purement logicielle, repose sur des communications fréquentes de données entre différentes classes, ce qui implique donc de n'accélérer que des portions très restreintes d'un algorithme sur matériel (sous peine de temps de communications coûteux entre hôte et accélérateur). Déporter de plus gros grains d'un modèle sur accélérateur demande de porter également le comportement de chaque classe de QuantLib impliquée, au prix des gains espérés en temps de calcul. Les premiers résultats ont montré les limites de cette approche, dans laquelle les optimisations du logiciel deviennent des limitations de l'accélérateur. Cette première étude a également permis de montrer l'importance d'optimisations globales, et non pas partielles, afin de limiter les transferts de données et les points de rendez-vous dans le programme.

Le chapitre 3 se rapproche des couches matérielles pour présenter une étude d'architecture sur un cas concret. Nous y présentons l'évaluation d'options américaines sur cible FPGA (avec comparaison des résultats sur cible GPU). Deux optiques sont détaillées : une première implémentation repose sur un calcul en flot de donnée, tandis que la seconde se concentre sur l'optimisation locale des accès mémoires. L'implémentation a été réalisée à l'aide d'un flot haut-niveau (Compilateur Altera OpenCL) et nous avons obtenu des performances conséquentes pour la deuxième approche, avec un budget énergétique faible. Cette architecture a ensuite été étendue à un problème plus complexe - le calcul de volatilité implicite.

Les temps de développement relativement courts pour une telle architecture valident cette approche haut-niveau pour le développement d'accélérateur de calcul sur matériel. La solution d'accélération développée est toutefois très spécialisée ; sa modification pour étendre ses fonctionnalités ou la mettre à jour n'est donc pas triviale.

Un équilibre entre la stratégie de développement détaillée dans ce chapitre et l'intégration au sein d'une librairie logicielle telle que détaillée chapitre 2 est proposé dans le chapitre 4. Celui-ci repose sur une chaîne de précompilation de code spécifique au domaine pour être en adéquation avec la plateforme cible.

L'étude approfondie de la plateforme matérielle permet de fournir des statistiques sur les performances des opérateurs élémentaires, les éléments mémoires utilisés et les liens de communications. Ces statistiques sont clés pour fournir à l'utilisateur et au reste de la chaîne de compilation les indices nécessaires pour déterminer les options de parallélismes les plus prometteuses. Elles donnent une estimation de l'occupation en ressource de noyaux de calculs portés sur matériel et du meilleur dimensionnement en termes de taille de *work-groups* et en termes de noyaux utilisables en parallèle (sans reconfiguration). Elles confirment de plus l'intérêt de l'utilisation de mémoire locale pour le transfert de données, ainsi que la mémoire globale pour des grands volumes de données.

L'état de l'art sur les DSL (majoritairement hors domaine financier), ainsi que les conclusions tirées de l'intégration à QuantLib dans le chapitre 2 permettent d'extraire les abstractions du

domaine qui peuvent être retranscrites sur matériel. Nous avons défini un étage de compilation intermédiaire acceptant du code haut-niveau en entrée et dont le produit est accepté en entrée de la chaîne de synthèse d'Altera OpenCL.

Cette surcouche repose sur les librairies de compilation LLVM pour définir une série de passes, permettant d'extraire du noyau défini par l'utilisateur les données nécessaires à son passage sur matériel. Ces passes permettent de définir un CFG enrichi, graphe de dépendance en données et en contrôle du noyau, ce qui guide les passes d'optimisations suivantes (parallélisation via portage sur *work-group*, déroulage de boucle pour un pas de temps fixe, etc.). La traduction des opérations élémentaires réalisées dans le noyau est ensuite effectuée à l'aide des données obtenues lors de l'étude de la plateforme. La génération de métadonnées spécifiques au flot AOC est la dernière étape avant insertion dans le flot d'Altera pour la génération de bitstream pour accélération.

Perspectives

Les résultats obtenus laissent entrevoir différentes pistes pour continuer sur cette voie. A court terme, l'utilisation de DSL semble prometteuse, mais doit être plus complètement validée. L'intégration complète doit donc être finalisée afin de confirmer les résultats déjà obtenus. Sans prendre en compte le travail d'extraction du noyau, qui nécessite plus qu'une simple implémentation, les autres portions du flot ne sont pas encore prêtes pour l'intégration. Les phases de spécification et de conception de ces étapes sont terminées, mais l'implémentation n'est pas opérationnelle. Une fois le flot intégré, une validation sur l'application exploitée dans le chapitre 3 permettrait de comparer les deux approches, et de guider les évolutions futures.

A moyen terme, différents axes nous semblent pertinents. Le travail de caractérisation réalisé pour les cibles Altera pourrait également être fait sur d'autres cibles, afin d'élargir la portée de ce travail. LLVM est un outil largement utilisé, qui permet en particulier la génération de code pour GPP (son but premier) et GPU à partir de sa représentation intermédiaire. Une représentation normalisée d'une plateforme mériterait également d'être étudiée. Elle pourrait ensuite être utilisée par le flot pour générer une application optimisée pour différentes cibles. Porter le travail réalisé pour le FPGA sur GPU permettrait d'étendre les possibilités d'un tel flot.

Une extension du flot à d'autres cas, avec potentiellement d'autres schémas algorithmiques, est également envisagée. La double discrétisation exploitée pour simplifier la génération de l'architecture est valable pour la plupart des modèles. Elle correspond cependant à une représentation grossière, qui pourrait peut-être être affinée. Une étude plus poussée permettrait d'estimer la pertinence de cette représentation, et en particulier les variations qui sont faites par les développeurs. Le cas de la volatilité implicite est représentatif : la double discrétisation existe pour le calcul d'option, mais ne se retrouve pas dans le calcul de la volatilité elle-même. Selon les cas, une trop grande imbrication des calculs peut rendre le repérage de cette discrétisation impossible, d'où la nécessité d'affiner le modèle.

Finalement, une intégration plus fine dans OpenCL pourrait être envisagée. L'outil est prin-

cipalement utilisé pour l'instant pour la génération de noyaux, la partie contrôle sur l'hôte étant laissée telle quelle. Cependant, un travail conjoint sur le noyau et sur certaines fonctions de l'hôte (en particulier au niveau des transferts de données) pourrait avoir un intérêt. L'intégration de plusieurs cibles, choisies en fonction des paramètres, est aussi une perspective de ce travail. La génération d'un choix entre FPGA, GPP ou GPU par exemple au sein d'un même programme, sur une machine disposant de ces trois cibles, permettrait de tirer pleinement partie d'une architecture hétérogène. Ce dernier point est cependant à plus long terme.

Une dernière perspective de ce travail, plus lointaine, est l'extraction de noyaux. Cette extraction est nécessaire pour obtenir une application performante, sans intervention du développeur. Elle est cependant complexe, et pourrait faire l'objet d'une thèse à part entière. Des discussions préliminaires ont permis de visualiser différentes solutions. L'extraction de *pattern* dans un graphe est une piste qui nous semble prometteuse. L'idée serait de retrouver un *pattern* connu (le modèle de l'application, ou des portions adaptées à une accélération matérielle) dans le graphe représentant le programme. Une autre approche, plus gloutonne, pourrait utiliser les annotations issues des phases d'analyse pour mesurer les performances de l'application en fonction de la cible utilisée pour différentes opérations.

Bibliographie

[AID12] E. Atanassov, S. Ivanovska, and D. Dimitrov. Parallel implementation of option pricing methods on multiple GPUs. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 368–373, May 2012.

[Alt06] Altera. White Paper - FPGA Architecture. Technical Report v1.0, Altera Corporation, 2006.

[Alt15] Altera. Stratix v device overview. Technical Report v1.0, Altera Corporation, 2015.

[Bal] L. Ballabio. Implementing QuantLib. `http://www.implementingquantlib.com/`, visité le 29/05/2017.

[Bat96] D. S. Bates. Jumps and stochastic volatility : exchange rate processes implicit in deutsche mark options. *Review of Financial Studies*, 9(1) :69–107, January 1996.

[BDSW15] Christian Brugger, Christian De Schryver, and Norbert Wehn. *Bringing Flexibility to FPGA Based Pricing Systems*, pages 167–190. Springer International Publishing, Cham, 2015.

[boo] Boost C++ Libraries. `http://www.boost.org/`, visité le 29/05/2017.

[BS73] Fischer Black and Myron S. Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3) :637–54, 1973.

[BSL+11] K.J. Brown, A.K. Sujeeth, Hyouk Joong Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100, 2011.

[Bus] Author : Cade Metz Cade Metz Business. Google Reincarnates Dead Paper Mill as Data Center of Future. `https://www.wired.com/2012/01/google-finland/`, visité le 29/05/2017.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp : high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[CKW12] G. Chatziparaskevas, B. Kienhuis, and J. Walters. An FPGA-based parallel processor for black-scholes option pricing using finite differences schemes. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 709–714, March 2012.

[CLN+11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs : from prototyping to deployment. *IEEE Transactions on Computer Aided Design*, 30(4) :473–491, April 2011.

[CRR79] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing : A simplified approach. *Journal of Financial Economics*, 7(3) :229–263, September 1979.

[DCJ10] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. Pricing multi-asset american options on graphics processing units using a PDE approach. In *2010 IEEE Workshop on High Performance Computational Finance (WHPCF)*, pages 1–8, November 2010.

[dDP11] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4) :18–27, 2011.

[DHS07] Toby Daglish, John Hull, and Wulin Suo. Volatility surfaces : theory, rules of thumb, and empirical evidence. *Quantitative Finance*, 7(5) :507–524, 2007.

[dSSK+11] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn. An energy efficient FPGA accelerator for monte carlo option pricing with the heston model. In *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 468–474, December 2011.

[Fou] HSA Foundation. HSA foundation ARM, AMD, imagination, MediaTek, qualcomm, samsung, TI. `http://www.hsafoundation.com/`, visité le 29/05/2017.

[GLN+14] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from Domain-Specific Languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[GLS11] A. George, H. Lam, and G. Stitt. Novo-G : At the Forefront of Scalable Reconfigurable Supercomputing. *Computing in Science Engineering*, 13(1) :82–86, 2011.

[GNR+13] Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 120–127. IEEE, 2013.

[GR09] Pierre Gauthier and Pierre-Yves Rivaille. Fitting the smile, smart parameters for SABR and heston. *SSRN Electronic Journal*, (ID 1496982), October 2009.

[Gro] The Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/api/opencl/`, visité le 29/05/2017.

[Gro11] Khronos OpenCL Working Group. The OpenCL Specification. Technical report, Khronos Group, 2011.

[GS10] Manfred Gilli and Enrico Schumann. Calibrating Option Pricing Models with Heuristics. *SSRN Electronic Journal*, (ID 1566975), March 2010.

[Hes93] Steven L. Heston. A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *The Review of Financial Studies*, 6(2) :327–343, January 1993. ArticleType : research-article / Full publication date : 1993 / Copyright Â© 1993 Oxford University Press.

[HKLW02] Patrick S. Hagan, Deep Kumar, Andrew S. Lesniewski, and Diana E. Woodward. Managing smile risk. *The Best of Wilmott*, 1 :249–296, 2002.

[Hul14] J. Hull. *Options, futures, et autres actifs dérivés*. ECO GESTION. PEARSON EDUCATION, 2014.

[Ing] Gordon Inggs. ForwardFinancialFramework : F^3 is python-based framework for valuing forward looking financial products on heterogeneous parallel computing platforms. original-date : 2012-07-12T14 :56 :21Z.

[Int] Intel. Caractéristiques du produit intel® xeon® processor x5450 (12m cache, 3.00 GHz, 1333 MHz FSB). `https://ark.intel.com/fr/products/34446/Intel-Xeon-Processor-X5450-12M-Cache-3_00-GHz-1333-MHz-FSB`, visité le 29/05/2017.

[JLT11a] Qiwei Jin, W. Luk, and D.B. Thomas. On comparing financial option price solvers on FPGA. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 89–92, May 2011.

[JLT11b] Qiwei Jin, W. Luk, and D.B. Thomas. Unifying Finite Difference Option-Pricing for Hardware Acceleration. In *2011 International Conference on Field Programmable Logic and Applications (FPL)*, pages 6–9, September 2011.

[JT] Richard Jordan and Charles Tier. Asymptotic approximations to deterministic and stochastic volatility models. *SIAM Journal on Financial Mathematics*, 2(1) :935–964.

[JTLC08] Qiwei Jin, David B. Thomas, Wayne Luk, and Benjamin Cope. Exploring reconfigurable architectures for binomial-tree pricing models. In *Reconfigurable Computing : Architectures, Tools and Applications*, pages 245–255. Springer, 2008.

[kin13] Memory module specifications - kvr16s11s6/2. Technical Report VALUERAM1370-001.A00, Kingston Technology, 2013.

[Koc10] Andreas Koch. Adaptive computing systems and their design tools. In *Dynamically Reconfigurable Systems*, pages 117–138. Springer, 2010.

[LL96] D. Lamberton and B. Lapeyre. *Introduction to Stochastic Calculus Applied to Finance, Second Edition*. Chapman & Hall/CRC Financial Mathematics. Chapman & Hall, 1996.

[LLVa] LLVM. "clang" c language family frontend for LLVM. `http://clang.llvm.org/`, visité le 29/05/2017.

[LLVb] LLVM. The LLVM compiler infrastructure project. `http://llvm.org/`, visité le 29/05/2017.

[MBN10] Dariusz Murakowski, William Brouwer, and V. Natoli. CUDA implementation of barrier option valuation with jump-diffusion process and brownian bridge. In *2010 IEEE Workshop on High Performance Computational Finance (WHPCF)*, pages 1–4, November 2010.

[McW05] Nairn McWilliams. Pricing American options using Monte Carlo simulation. White Paper, 2005.

[Mer76] Robert C. Merton. Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, 3(1–2) :125–144, January 1976.

[MHT+12] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Korner, and Wieland Eckert. Generating device-specific GPU code for local operators in medical imaging. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 569–581. IEEE, 2012.

[Mit09] Sovan Mitra. A Review of Volatility and Option Pricing. April 2009.

[MKK+11] Henning Marxen, Anton Kostiuk, Ralf Korn, Christian de Schryver, Stephan Wurm, Ivan Shcherbakov, and Norbert Wehn. Algorithmic complexity in the heston model : an implementation view. In *Proceedings of the fourth workshop on High performance computational finance*, WHPCF '11, pages 5–12, New York, NY, USA, 2011. ACM.

[NU12] Q. Nasar-Ullah. GPU acceleration for the pricing of the CMS spread option. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[NVIa] NVIDIA. Carte graphique GeForce GTX 660 ti avec technologie kepler|NVIDIA. `http://www.nvidia.fr/object/geforce-gtx-660ti-fr.html`, visité le 29/05/2017.

[NVIb] NVIDIA. Parallel programming and computing platform | CUDA | NVIDIA | NVIDIA. `http://www.nvidia.com/object/cuda_home_new.html`, visité le 29/05/2017.

[NVI12] NVIDIA. Technology overview - nvidia geforce gtx 680. Technical Report v1.0, NVIDIA Corporation, 2012.

[NVi13] NVidia. Nvidia tesla - gpu accelerators. Technical report, NVIDIA Corporation, 2013.

[ORS+13] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in scala : towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th international conference on Generative programming : concepts and experiences*, pages 125–134. ACM Press, 2013.

[PCC+14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth

Gopal, and Simon Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[PF05] M. Pharr and R. Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[Qua] QuantLib. QuantLib : a free/open-source library for quantitative finance. `http://quantlib.org/index.shtml`, visité le 29/05/2017.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide : a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6) :519–530, 2013.

[RO10] Tiark Rompf and Martin Odersky. Lightweight modular staging : a pragmatic approach to runtime code generation and compiled DSLs. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[RSH+14] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2014.

[SCH+12] R. Sridharan, G. Cooke, K. Hill, H. Lam, and A. George. FPGA-Based reconfigurable computing for pricing multi-asset barrier options. In *2012 Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, pages 34–43, July 2012.

[swi] Simplified Wrapper and Interface Generator. `http://www.swig.org/`, visité le 29/05/2017.

[Tec] Terasic Technologies. Terasic - DE main boards - stratix - DE5-net FPGA development kit. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=164&No=526`, visité le 29/05/2017.

[tim] tim.lewis. OpenMP - enabling HPC since 1997. `http://www.openmp.org/`, visité le 29/05/2017.

[Top] Top500. November 2016 | TOP500 supercomputer sites. `https://www.top500.org/lists/2016/11/`, visité le 29/05/2017.

[TT14] A. Tavakkoli and D.B. Thomas. Low-latency option pricing using systolic binomial trees. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 44–51, 2014.

[TTL12] A.H.T. Tse, D. Thomas, and W. Luk. Design Exploration of Quadrature Methods in Option Pricing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(5) :818–826, May 2012.

[TTTL10a] A.H.T. Tse, D.B. Thomas, K.H. Tsoi, and W. Luk. Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters. In *2010 International Conference on Field-Programmable Technology (FPT)*, pages 233–240, December 2010.

[TTTL10b] A.H.T. Tse, D.B. Thomas, K.H. Tsoi, and W. Luk. Reconfigurable Control Variate Monte-Carlo Designs for Pricing Exotic Options. In *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pages 364–367, September 2010.

[TZKH10] Yu Tian, Zili Zhu, Fima C. Klebaner, and Kais Hamza. Option pricing with the SABR model on the GPU. In *2010 IEEE Workshop on High Performance Computational Finance (WHPCF)*, pages 1–8, November 2010.

[VBT+15] Javier Alejandro Varela, Christian Brugger, Songyin Tang, Norbert Wehn, and Ralf Korn. Pricing high-dimensional american options on hybrid CPU/FPGA systems. In Christian De Schryver, editor, *FPGA Based Accelerators for Financial Applications*, pages 143–166. Springer International Publishing, 2015.

[vis] VisualHDL. `http://visualhdl.sysprogs.org/`, visité le 29/05/2017.

[War08] Pete Warren. City business races the Games for power. *The Guardian*, May 2008.

[Web] Halide.

[WMI09] Chris Wynnyk and Malik Magdon-Ismail. Pricing the American Option Using Reconfigurable Hardware. In *International Conference on Computational Science and Engineering (CSE)*, pages 532–536. IEEE, 2009.

[WMPW03] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement c-slow retiming for the xilinx virtex FPGA. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, pages 185–194. ACM, 2003.

[Xil14] Xilinx. Vivado Design Suite User Guide - High-Level Synthesis. Technical Report UG902, Xilinx Corporation, 2014.

Liste des publications

Publications en conférence

[1] Valentin Mena Morales, Erik Hochapfel, Pierre-Henri Horrein, Sandrine Vaton, Amer Baghdadi. Étude d'architectures pour l'évaluation d'options américaines. In *Colloque national du groupe de recherches System-on-Chip - System-in-Package (GDR SoC-SiP)*, Lyon, France, 2013.

[2] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Erik Hochapfel, Sandrine Vaton. Energy-Efficient FPGA Implementation for Binomial Option Pricing Using OpenCL. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, Dresden, Germany. 2014.

[3] Valentin Mena Morales, Yahia Brakni, Pierre-Henri Horrein, Amer Baghdadi. CAASPER : Providing Accessible FPGA-acceleration over the Network. In *Proceedings of the 26th IEEE International Symposium on Rapid System Prototyping (RSP)*, Amsterdam, Netherlands, 2015.

Publications en revue

[4] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Thomas Jourdan. Accurate and energy efficient FPGA implementation for financial applications using OpenCL. *Submitted to ACM Transactions on Reconfigurable Technology and Systems*, 2016.

Résumé

Les applications de calcul haute-performance (HPC) nécessitent des capacités de calcul conséquentes, qui sont généralement atteintes à l'aide de fermes de serveurs au détriment de la consommation énergétique d'une telle solution. L'accélération d'applications sur des plateformes hétérogènes, comme par exemple des FPGA ou des GPU, permet de réduire la consommation énergétique et correspond donc à un compromis architectural plus séduisant. Elle s'accompagne cependant d'un changement de paradigme de programmation et les plateformes hétérogènes sont plus complexes à prendre en main pour des experts logiciels. C'est particulièrement le cas des développeurs de produits financiers en finance quantitative. De plus, les applications financières évoluent continuellement pour s'adapter aux demandes législatives et concurrentielles du domaine, ce qui renforce les contraintes de programmabilité de solutions d'accélérations. Dans ce contexte, l'utilisation de flots haut-niveaux tels que la synthèse haut-niveau (HLS) pour programmer des accélérateurs FPGA n'est pas suffisante. Une approche spécifique au domaine peut fournir une réponse à la demande en performance, sans que la programmabilité d'applications accélérées ne soit compromise.

Nous proposons dans cette thèse une approche de conception haut-niveau reposant sur le standard de programmation hétérogène OpenCL. Cette approche repose notamment sur la nouvelle implémentation d'OpenCL pour FPGA introduite récemment par Altera. Quatre contributions principales sont apportées : (1) une étude initiale d'intégration de cœurs de calculs matériels à une librairie logicielle de calcul financier (QuantLib), (2) une exploration d'architectures et de leur performances respectives, ainsi que la conception d'une architecture dédiée pour l'évaluation d'option américaine et l'évaluation de volatilité implicite à partir d'un flot haut-niveau de conception, (3) la caractérisation détaillée d'une plateforme Altera OpenCL, des opérateurs élémentaires, des surcouches de contrôle et des liens de communication qui la compose, (4) une proposition d'un flot de compilation spécifique au domaine financier, reposant sur cette dernière caractérisation, ainsi que sur une description des applications financières considérées, à savoir l'évaluation d'options.

Mots clef : Conception haut-niveau, OpenCL, FPGA, GPU, Finance, Accélération matérielle, HPC, HLS, Prototypage

Abstract

The need for resources in High Performance Computing (HPC) is generally met by scaling up server farms, to the detriment of the energy consumption of such a solution. Accelerating HPC application on heterogeneous platforms, such as FPGAs or GPUs, offers a better architectural compromise as they can reduce the energy consumption of a deployed system. Therefore, a change of programming paradigm is needed to support this heterogeneous acceleration, which trickles down to an increased level of programming complexity tackled by software experts. This is most notably the case for developers in quantitative finance. Applications in this field are constantly evolving and increasing in complexity to stay competitive and comply with legislative changes. This puts even more pressure on the programmability of acceleration solutions. In this context, the use of high-level development and design flows, such as High-Level Synthesis (HLS) for programming FPGAs, is not enough. A domain-specific approach can help to reach performance requirements, without impairing the programmability of accelerated applications.

We propose in this thesis a high-level design approach that relies on OpenCL, as a heterogeneous programming standard. More precisely, a recent implementation of OpenCL for Altera FPGA is used. In this context, four main contributions are proposed in this thesis: (1) an initial study of the integration of hardware computing cores to a software library for quantitative finance (QuantLib), (2) an exploration of different architectures and their respective performances, as well as the design of a dedicated architecture for the pricing of American options and their implied volatility, based on a high-level design flow, (3) a detailed characterization of an Altera OpenCL platform, from elemental operators, memory accesses, control overlays, and up to the communication links it is made of, (4) a proposed compilation flow that is specific to the quantitative finance domain, and relying on the aforementioned characterization and on the description of the considered financial applications (option pricing).

Keywords: High-level design, OpenCL, FPGA, GPU, quantitative finance, hardware acceleration, HPC, HLS, prototyping

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

N° d'ordre : 2017IMTA0018

IMT Atlantique Bretagne-Pays de la Loire - www.imt-atlantique.fr

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 03
T +33 (0)2 29 00 11 11
F +33 (0)2 29 00 10 00

Campus de Nantes
4, rue Alfred Kastler - La Chantrerie
CS 20722
44307 Nantes Cedex 3
T +33 (0)2 51 85 81 00
F +33 (0)2 51 85 81 99

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
T +33 (0)2 99 12 70 00
F +33 (0)2 99 12 70 08